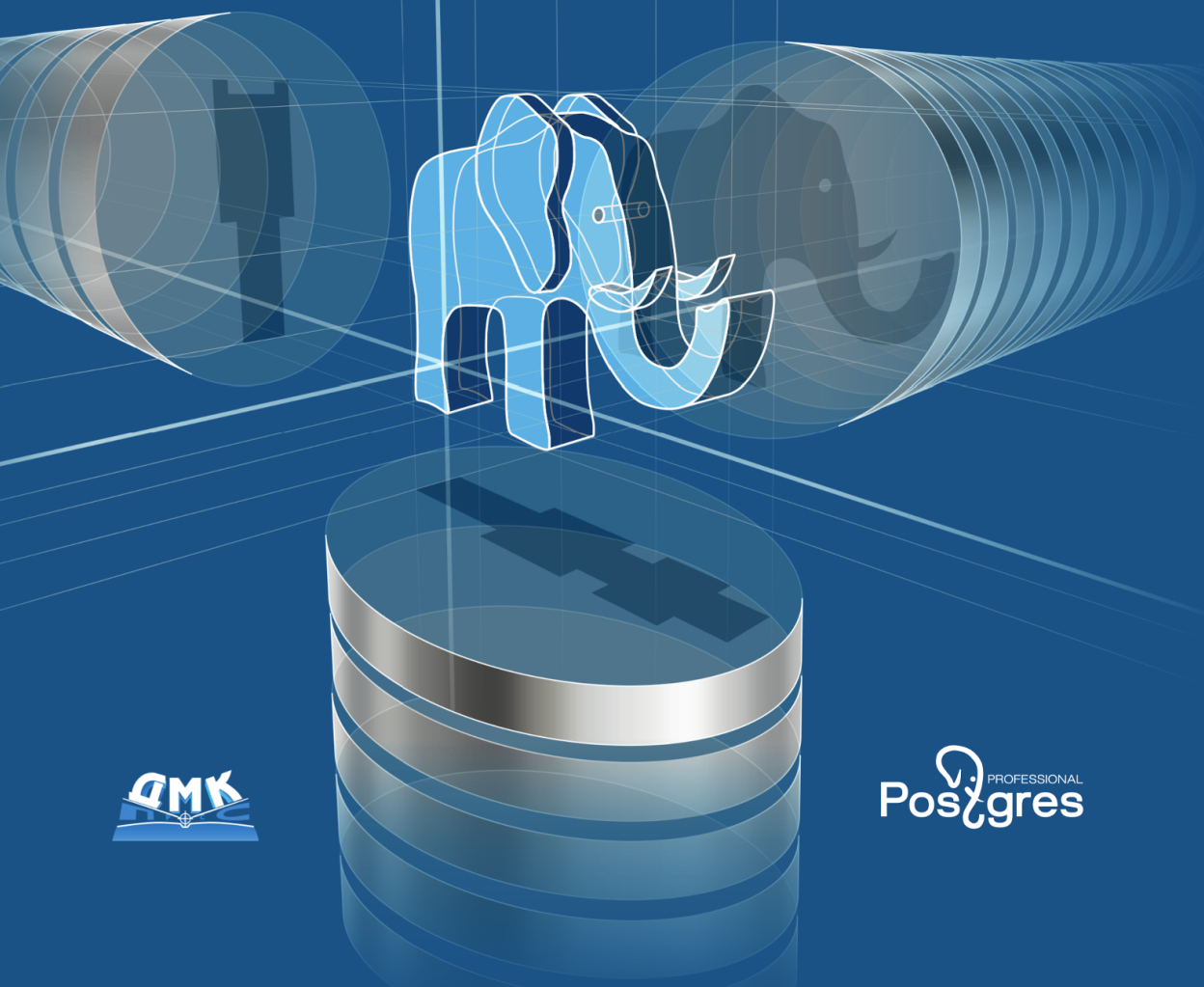
Б. А. Новиков,
Е. А. Горшкова, Н. Г. Графеева

ОСНОВЫ ТЕХНОЛОГИЙ БАЗ ДАННЫХ

второе издание



ДМК

PROFESSIONAL Postgres

Компания Postgres Professional

Б. А. Новиков,
Е. А. Горшкова, Н. Г. Графеева

Основы технологий баз данных

Второе издание

ДМК ПРЕСС
издательство

Москва, 2020

УДК 004.65
ББК 32.972.134
Н73

Новиков Б. А.

Н73 Основы технологий баз данных: учебное пособие / Б. А. Новиков, Е. А. Горшкова, Н. Г. Графеева; под ред. Е. В. Рогова. — 2-е изд. — М.: ДМК Пресс, 2020. — 582 с.

ISBN 978-5-97060-841-8

Материал первой части учебного пособия составляет основу для базового курса и содержит краткий обзор требований и критериев оценки СУБД и баз данных, теоретическую реляционную модель данных, основные конструкции языка запросов SQL, организацию доступа к базе данных PostgreSQL, вопросы проектирования приложений и основные расширения, доступные в системе PostgreSQL.

Вторая часть, добавленная в настоящем издании, содержит материал, который будет полезен разработчикам баз данных и СУБД. В ней подробно рассматриваются структуры хранения, методы выполнения и оптимизации запросов, дополнительные возможности языка SQL, средства поддержки согласованности и надежности. Рассмотрены средства программирования серверов баз данных, средства расширения функциональности PostgreSQL, вопросы создания систем с репликацией, параллельных и распределенных систем баз данных.

Сайт книги: https://postgrespro.ru/education/books/dbtech.

Для программистов и студентов

УДК 004.655
ББК 32.973.134

ISBN 978-5-6041193-5-8
ISBN 978-5-97060-841-8

Оглавление

О курсе

На кого ориентирован курс

Курс рассчитан на студентов классических и технических университетов и других вузов, имеющих базовую подготовку по программированию и продолжающих специализироваться в областях, близких к программированию.

Какие знания будут получены

В курсе подробно рассматриваются основные понятия, устройство и принципы работы СУБД, а также технологии (архитектура, алгоритмы, структуры данных), лежащие в их основе.

Прослушавшие курс получат уверенные знания и практические навыки по следующим вопросам:

- устройство и принципы работы СУБД;
- проектирование баз данных;
- работа с SQL — составление и оптимизация запросов;
- разработка серверных приложений;
- использование различных типов индексов;
- обработка транзакций и одновременный доступ;
- основы эксплуатации баз данных;
- обеспечение надежности хранения, отказоустойчивости и высокой доступности;
- принципы организации и работы параллельных и распределенных СУБД;
- работа со слабоструктурированными данными (JSON, XML).

Такая подготовка позволит на старших курсах (в магистратуре) специализироваться на разработке и настройке приложений баз данных либо в областях проектирования и разработки СУБД.

Структура курса

Курс состоит из двух частей.

Первая часть рассчитана на студентов младших курсов бакалавриата. В ней рассматриваются основные сведения о базах данных и системах управления базами данных: реляционная модель данных, язык SQL, обработка транзакций.

Вторая часть курса может читаться на последних курсах бакалавриата или для студентов магистратуры. Во второй части подробно рассматриваются технологии, лежащие в основе функционирования СУБД, а также современные направления и тенденции развития СУБД, основные аспекты их практического применения. При этом некоторые темы, рассмотренные в первой части, изучаются повторно на более глубоком уровне.

Курс в основном касается классических реляционных и объектно-реляционных СУБД, но затрагивает также тематику неклассических СУБД.

Практические занятия не только помогают закрепить пройденный на лекциях материал. Они содержат много дополнительной информации, закрепляющей и расширяющей знания, изложенные в теоретической части. В качестве СУБД для практических занятий используется PostgreSQL.

Как первая, так и вторая части курса могут быть выделены в самостоятельные курсы. Отдельные разделы курса могут быть скомбинированы так, чтобы получить более практическую или более фундаментальную направленность либо адаптировать курс к конкретному учебному плану вуза.

Программные средства, используемые в курсе

Для эффективного освоения материала курса и для выполнения упражнений необходимо установить на компьютере ряд программных продуктов. Набор этих продуктов может зависеть от используемой операционной системы и от других обстоятельств, но в любом случае понадобятся:

- система управления базами данных PostgreSQL;
- демонстрационная база данных, которая используется в большинстве примеров;
- текстовый редактор для подготовки запросов на языке SQL.

Установка PostgreSQL и демобазы рассматривается в главе 3.

Для выполнения упражнений по созданию и редактированию моделей данных может потребоваться инструмент для редактирования диаграмм.

Для разработки приложений на императивных языках программирования (C, C++, Java, Python и др.) потребуются соответствующие среды разработки, однако такие упражнения не включены в состав этого курса.

При работе с PostgreSQL можно использовать ряд приложений, предоставляющих графические интерфейсы для работы с базами данных. Многие из этих приложений могут работать с различными СУБД. Как правило, в таких системах ограничены возможности работы с особенностями любой конкретной СУБД. При освоении материала этого курса целесообразнее использовать средства самой системы PostgreSQL или другие программы, спроектированные специально для работы с PostgreSQL.

Благодарности

Подготовка этого курса была бы невозможна без активной поддержки со стороны компании Postgres Professional и ее руководства, в частности О. Бартунова и И. Панченко. Качество материала существенно улучшилось благодаря усилиям Е. Рогова, взявшего на себя огромный труд по редактированию курса.

Главы 7 и 9 написаны совместно Е. Горшковой и Б. Новиковым, глава 20 — совместно Н. Графеевой и Б. Новиковым. В подготовке упражнений принимали участие В. Бусаров, К. Секереш, Г. Шалыгина и Е. Михайлова.

Часть I

От теории к практике

Глава 1

Введение

Компьютеры используются повсеместно: невозможно найти предприятие или учреждение, которые не применяли бы их для решения производственных или управленческих задач. Подобные высказывания не слишком заметны в средствах массовой информации, потому что они уже давно не являются новостью. Профессионалы, однако, понимают, что на самом деле важны не компьютеры, а информационные системы, которые на них работают, а в центре любой информационной системы находятся данные.

Эта книга о том, как хранить данные, обеспечивать их корректность и сохранность и как их обрабатывать эффективно.

1.1. Базы данных и СУБД

Появление и относительно широкое распространение в начале 1960-х гг. запоминающих устройств достаточно большой емкости с возможностью доступа к произвольным участкам памяти — магнитных дисков — открыло широкие возможности для создания сложных структур долговременно хранимых данных. Высокая скорость обновления небольших объемов данных (доли секунды) создала условия для создания приложений, способных функционировать в режиме оперативной работы (online). В отличие от систем предшествующих поколений время ответа стало измеряться не сутками, а секундами или долями секунды.

Эти возможности, однако, привели к существенному усложнению кода приложений и, как следствие, к удорожанию их разработки и снижению надежности. В связи с этим появилась идея централизации функций управления данными, которая привела к появлению систем, предоставляющих приложениям услуги по обработке данных. Такие системы получили название систем управления базами данных (СУБД).

Поскольку СУБД используются многими приложениями, ожидается, что они могут обеспечивать более высокие значения эксплуатационных характеристик, таких как надежность хранения и эффективность обработки, недостижимые при индивидуальной разработке средств управления данными для каждого приложения.

Важно отметить, что многие особенности и характеристики, присущие ранним системам управления базами данных, связаны с требованиями тех областей применения и классов приложений, которые были наиболее актуальны в то время. В первую очередь это приложения, работающие в режиме оперативной обработки (online transaction processing, OLTP) в банковской и финансовой сферах.

Прежде чем обсуждать, каким образом эти области применения повлияли на характеристики СУБД, уточним значение некоторых терминов, которые будут использоваться в дальнейшем.

Система управления базами данных (СУБД) — это программный комплекс, обеспечивающий централизованное хранение данных и предоставляющий приложениям услуги по обработке данных.

Совокупность данных, хранимых под управлением СУБД, называется *базой данных*. Оригинальное английское словосочетание data base дословно переводится как «основание, состоящее из данных». В русском словосочетании «база данных» этот смысл несколько искажается. На самом деле это — фундамент, на котором строятся приложения и который состоит из данных. Действительно, данные (а следовательно, база данных) являются очень существенной частью практически любой информационной системы.

Система управления базами данных, находящаяся в фазе выполнения, связанная с некоторой конкретной базой данных и готовая выполнять запросы на обработку этой базы данных, называется *экземпляром* (instance) или *сервером базы данных*. На самом деле экземпляр и сервер — разные понятия: один сервер баз данных может управлять несколькими экземплярами баз данных, однако это различие станет важным только начиная с главы 5.

1.2. Требования к СУБД

Ранние системы управления данными очень сильно различались как по своей внутренней организации, так и по предоставляемым возможностям. Потребо-

валось несколько лет, для того чтобы определить, каковы основные функции систем управления базами данных и какие требования следует предъявлять к таким системам.

Основные требования к системам управления базами данных были сформулированы в документе, опубликованном в 1971 г. комитетом по системам и языкам обработки данных (CODASYL) [28], русский перевод которого издан в 1975 г. [64]. Основой для этих требований послужил анализ систем, применявшихся в период подготовки отчета, и особенностей прикладных областей, в которых эти системы использовались.

В дальнейшем круг областей применения СУБД непрерывно расширялся, появлялись новые системы и уходили старые, однако многие из этих требований остались актуальными и сегодня, и большинство современных СУБД в той или иной форме реализует значительную часть этих требований. Однако далеко не все классы приложений, в которых используются современные системы, предъявляют те же требования к обработке данных, поэтому и системы реализуются иначе.

Приложения, относящиеся к классу оперативной обработки (OLTP), характеризуются тем, что:

- каждое выполнение приложения занимает мало времени (в идеале — не больше долей секунды);
- данные используются совместно многими приложениями;
- при каждом выполнении приложение использует ничтожную долю общего объема хранимых данных, и обычно количество используемых данных не зависит от общего объема базы.

Важно также отметить, что процессы обработки данных и структуры данных в тех областях, в которых использовались ранние СУБД, фактически были формализованы задолго до появления электронных вычислительных систем. Так, правила бухгалтерского учета в основном сложились в XIV веке и мало изменились в последующем. Возможно, это привело к тому, что СУБД, как правило, ориентированы на обработку структурированных данных.

Перечислим основные требования к системам управления базами данных.

Разделение программ и данных. Описание структуры данных должно быть отделено от кода приложений, и система должна допускать независимое

изменение структуры данных и кода приложения. В документе [64] использовался термин «независимость» (independence), однако «разделение» лучше отражает существо этого требования.

Высокоуровневый язык запросов. Система должна предоставлять средства для обработки данных, не включенные в какое-либо приложение.

Целостность. Система должна предотвращать запись данных, нарушающих заранее специфицированные ограничения.

Согласованность. Система должна предотвращать появление некорректностей в данных вследствие параллельной или псевдопараллельной работы нескольких приложений.

Отказоустойчивость. СУБД не должна допускать потери данных даже в случае отказов оборудования.

Защита и разграничение доступа. Система должна предотвращать несанкционированный доступ к данным и предоставлять каждому пользователю (или приложению) доступ только к части данных в соответствии с правами этого пользователя.

Заметим, что в ранние годы существования СУБД предполагалось, что все данные, необходимые для информационных систем предприятия, будут храниться в единой базе данных. Безусловно, это очень хорошая идея, поскольку при этом создаются возможности для исключения избыточности данных, проверки их корректности и предотвращается рассогласованность данных, используемых различными подразделениями предприятия. Почти все учебники по базам данных написаны с учетом этого предположения.

Однако на практике это никогда не было реализовано: как правило, для каждого приложения или небольшой группы приложений создается отдельная база данных. Основная причина, по-видимому, состоит в том, что структуры данных, необходимые для обеспечения информационных потребностей предприятия, слишком сложны для того, чтобы их можно было описать в рамках одной базы данных. Это обстоятельство существенно влияет на значение и использование как перечисленных требований, так и других особенностей систем управления базами данных.

Далее в этой главе данные требования и возможности СУБД обсуждаются более детально, а в последующих главах показано, как эти возможности реализуются в современных системах и, в частности, в системе PostgreSQL.

1.3. Разделение данных и программ

Каждая система управления данными создает некоторый уровень абстракции для других программных компонент, которые используют ее услуги. Для того чтобы реализовать определенный уровень абстракции данных, необходимо, чтобы система могла работать с описанием данных, соответствующих требуемому уровню абстракции.

Так, современные файловые системы представляют файлы как последовательности байтов. Соответственно, операции над содержимым файлов выражаются в терминах позиций байтов внутри файла. Никакие более сложные операции не могут быть определены, потому что файловая система не имеет информации о структуре данных внутри файла.

Поскольку ожидается, что системы управления базами данных предоставляют операции над данными сложной структуры, необходимо, чтобы описание этой структуры было доступно СУБД и было бы общим для всех программ (приложений), использующих эти данные. Это приводит к идее отделения описания структуры данных от программ. Такое описание хранится в самой базе данных и называется *схемой базы данных*.

Для определения схем используются языки описания данных. Чем больше возможностей у такого языка, тем больше услуг может предоставить система управления базами данных. В то же время необходимость подготовки детализированного описания данных на фазе проектирования прикладной системы может в некоторых случаях оказаться чрезмерно трудоемкой.

Само по себе отделение описания данных от приложений не дает достаточной гибкости. Для того чтобы в полной мере реализовать идею разделения данных и программ, в 1975 г. (тем же комитетом CODASYL) была подготовлена обобщенная модель языка описания данных. Описание этой модели стало известно под названием «модель данных ANSI/SPARC», так как предполагалось, что эта модель будет иметь статус стандарта. Схематически основные компоненты этой модели представлены на рис. 1.3.1.

Модель включает:

внешнюю схему, содержащую описание данных в таком виде, в котором они будут использоваться приложением (отдельно для каждого приложения), а также отображение логической структуры данных во внешнюю схему;

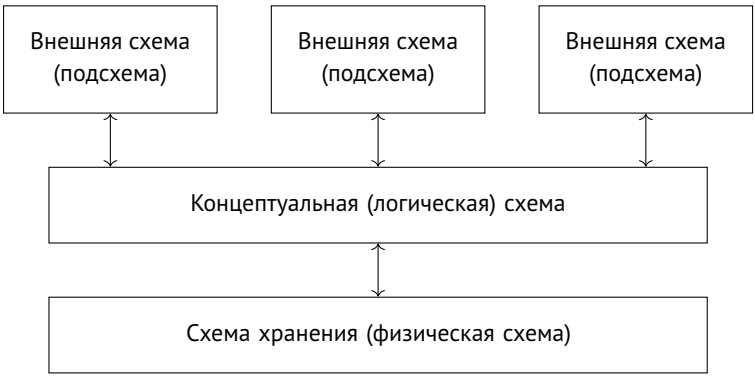


Рис. 1.3.1. Трехуровневая модель данных ANSI/SPARC

концептуальную схему, содержащую полное описание логической структуры данных, доступное СУБД (этот уровень описания было бы правильнее называть логической схемой базы данных);

схему хранения, описывающую, как организовано хранение логических структур данных.

Фактически эта модель никогда не была полностью реализована ни в одной системе, однако ее удобно использовать, для того чтобы определять назначение тех или иных составляющих языка описания данных.

В идеале трехуровневая модель обеспечивает возможности относительно независимой эволюции приложений и системы в целом. Так, при появлении новых приложений, использующих те же данные, достаточно определить новую внешнюю схему. В результате внедрение нового приложения не повлияет на работу других приложений.

Если новая версия приложения использует дополнительные элементы или структуры данных, достаточно определить новую внешнюю схему. Тогда старая и новая версии приложения смогут сосуществовать, что значительно упрощает постепенный и безопасный переход на новую версию.

Если для работы нового приложения требуются дополнительные структуры данных, эти структуры могут быть добавлены в концептуальную схему, что теоретически не повлияет на работу других приложений, так как их внешние схемы не будут содержать новых элементов данных.

Конечно, такая идеальная картина не всегда может реализоваться. Далеко не все изменения логической схемы могут остаться незаметными даже для приложений, которым не нужны измененные части, потому что изменения могут влиять не только на элементы данных, но и на взаимосвязи между ними.

В реальности эти потенциальные возможности могут реализоваться только в том случае, если разработчики как базы данных, так и приложений их тщательно учитывают. Эволюция базы данных имеет смысл в тех случаях, когда ценность накопленных данных высока. В других случаях создание новой базы данных может оказаться более оправданным решением. Далее в данном курсе будут рассмотрены примеры, иллюстрирующие оба подхода.

Развитые СУБД, в том числе PostgreSQL, предоставляют большое разнообразие методов хранения и поиска данных. Выбор этих методов влияет на производительность приложений, но не влияет на логическую структуру и на результаты выполняемых операций. Поэтому изменение структуры хранения можно использовать для того, чтобы повлиять на характеристики производительности отдельных приложений и системы в целом. Процесс внесения изменений, направленных на улучшение характеристик производительности и не затрагивающих логику работы приложений, называется *настройкой*.

В последние годы значение разделения описаний данных и программ зачастую недооценивается. Многие методологии разработки приложений предполагают генерацию схемы базы данных на основе объектной модели приложения. В частности, это характерно для методологий быстрой разработки прототипов, не предполагающих проведения тщательного предварительного анализа предметной области. Конечно, сложность при этом не исчезает, и задачи, связанные с проектированием схемы базы данных, все равно приходится решать, даже если они не выделены как отдельная единица работы.

Особенности проектирования схемы базы данных существенно зависят от применяемой модели данных. Различные типы моделей обсуждаются в главе 2. Здесь мы только отметим, что сложность проектирования может по-разному распределяться между базой данных и приложением: чем беднее модель данных, тем больше требуется делать на уровне приложения.

Наконец, заметим, что в некоторых подходах описание данных не оформляется как отдельная единица хранения (схема). Вместо этого описание данных может храниться вместе с самими данными (например, при использовании XML или JSON).

1.4. Языки запросов

Наличие описания логической структуры данных открывает возможности для выполнения достаточно сложных операций манипулирования данными внутри СУБД. Такие операции записываются на *языке запросов*.

Входящие в состав современных СУБД языки запросов являются декларативными, т. е. позволяют описать требуемый результат вычислений, но не способ выполнения этих вычислений. Благодаря этому СУБД может выбрать наиболее эффективные (по некоторому критерию производительности) алгоритмы получения результата. Это оказывается особенно полезным при массовой обработке данных, так как, с одной стороны, позволяет исключить передачу промежуточных результатов между сервером базы данных и приложением и, с другой стороны, выбрать оптимальный способ выполнения вычислений с учетом характеристик фактически хранимых данных, что недостижимо при программировании эквивалентных операций в коде приложения.

Мощные средства обработки декларативных запросов, предоставляемые современными СУБД, в том числе PostgreSQL, зачастую используются недостаточно. Это происходит по многим причинам, среди которых можно выделить плохую совместимость декларативных средств языков запросов с императивными средствами массово применяемых языков программирования.

Практика разработки приложений без использования возможностей языков запросов привела к появлению ряда систем, не предоставляющих такие средства (так называемые NoSQL-системы). При использовании подобных систем для хранения данных, очевидно, часть функций СУБД переносится в приложение. Потенциально это может приводить к увеличению сложности приложения и стоимости его разработки либо к снижению качества — что во многих случаях оказывается допустимым.

1.5. Целостность и согласованность

В состав логической схемы базы данных могут включаться не только описания структур данных и зависимостей между ними, но и дополнительные условия, которым хранимые данные обязательно должны удовлетворять. Такие условия называются *ограничениями целостности* (integrity constraints). Система управления базами данных проверяет ограничения целостности при выполнении

любых изменений хранимых данных и не допускает выполнения операций, нарушающих эти ограничения.

Поддержка ограничений целостности на уровне СУБД позволяет существенно упростить разработку приложений и одновременно улучшить их качество, так как исключает необходимость обработки некорректных данных. Такие данные просто не могут быть записаны в базу данных и, следовательно, никогда не будут выданы в качестве ответа на запрос приложения.

В качестве ограничений целостности можно задавать только условия, которые не могут нарушаться ни при каких обстоятельствах. Существуют, однако, условия корректности другого типа, обычно связывающие значения нескольких элементов данных. В качестве примера таких условий чаще всего приводится правило, согласно которому суммарный баланс при переводе средств с одного бухгалтерского счета на другой не может измениться. Такие условия могут нарушаться для отдельных операций, однако удовлетворяются для набора из нескольких операций.

Состояния базы данных, в которых выполняются подобные условия, называются *согласованными* (consistent), а сами правила — условиями согласованности. Конечный набор операций, который переводит согласованное состояние в другое согласованное, называется *транзакцией*. Несмотря на то что каждое приложение обеспечивает согласованность при выполнении своих транзакций, при неконтролируемом параллельном или псевдопараллельном выполнении нескольких транзакций согласованность может нарушаться. Одной из важных функций СУБД является предотвращение нарушений согласованности при одновременной работе многих приложений (или одного и того же приложения от имени разных пользователей).

Различие между целостностью и согласованностью можно пояснить следующим образом. Ограничения целостности описываются условиями в базе данных, и СУБД отвечает за то, чтобы эти ограничения выполнялись. Условия согласованности определяются приложением и не могут быть проверены СУБД, однако она гарантирует, что результаты выполнения приложения (транзакции) не будут зависеть от факторов, находящихся вне контроля приложения, в том числе от работы других приложений, сбоев и отказов вычислительной системы.

Литература по базам данных изобилует упрощенными примерами из финансовой области приложений, однако ни в коем случае не следует отождествлять понятия транзакции в базах данных с банковскими транзакциями или другими транзакциями в смысле прикладных предметных областей. В реальности

даже самые простые банковские транзакции реализуются в информационных системах как комбинации из нескольких транзакций в базах данных.

В русскоязычной литературе зачастую термин *consistency* переводится как «целостность», что приводит к путанице, так как термин *integrity* переводится точно так же. В этой книге слово *целостность* всегда относится к ограничениям целостности (integrity constraints), а термин *согласованность* обозначает понятие, выражаемое термином *consistency*.

1.6. Отказоустойчивость

В наши дни информационные системы используются повсеместно: едва ли найдется предприятие или организация, в которой они бы не применялись. Во многих случаях работа информационной системы стала жизненно важной для основных производственных процессов: отказы системы приводят к остановке бизнес-процессов, а потеря данных приводит к катастрофическим последствиям (зачастую не только для производственных функций, но и для жизни людей или состояния окружающей среды).

Поэтому при разработке систем управления базами данных с самого начала очень большое внимание уделялось средствам, обеспечивающим отказоустойчивость данных и их выживаемость. В результате длительного развития технологий, связанных с базами данных, эта цель была достигнута.

Современные системы при соответствующей конфигурации могут гарантировать полную сохранность данных и восстановление после отказов оборудования в корректном (согласованном) состоянии. При необходимости система может быть организована таким образом, чтобы восстановление занимало доли секунды. Другими словами, СУБД способны обеспечить значительно более высокую надежность и доступность данных, чем надежность или доступность оборудования, на котором эти данные хранятся и на котором работают эти системы.

Однако создание высоконадежных систем неизбежно оказывается весьма дорогостоящим. Это связано с необходимостью многократного дублирования используемых средств на всех уровнях, начиная от оборудования, что приводит к существенному усложнению системы. В случае использования внешних сервисов, например при размещении данных в облачной среде, необходимо иметь запасные ресурсы, способные обеспечить выживание при отказе этой среды,

даже если она позиционируется как высоконадежная. Поэтому при проектировании системы следует выбрать такой уровень защищенности от отказов, который для нее действительно необходим.

Технологических ограничений, которые не позволяли бы в достаточной мере защитить данные, не существует; все случаи, когда потеря данных происходила, связаны с недооценкой рисков при проектировании системы.

1.7. Безопасность и разграничение доступа

Данные нуждаются в защите не только от отказов оборудования и природных явлений, но и от несанкционированного доступа. Все развитые системы содержат средства как для предотвращения доступа к базе данных от имени лиц, не имеющих на это права, так и для разграничения доступа к данным тех, кто такое право имеет. Допускается обработка (чтение или модификация) только тех данных, для которых соответствующие операции разрешены лицу, от имени которого они выполняются.

Подобные средства защиты реализуются не только на уровне СУБД: защитой приходится заниматься практически на всех уровнях и во всех компонентах информационной системы. Во многих простых системах средства защиты, предоставляемые СУБД, вообще не используются. Однако по-настоящему надежная защита должна быть многоуровневой, а некоторые ее виды вообще невозможно реализовать без использования СУБД.

1.8. Производительность

Сравнение различных систем управления базами данных и оценка их применимости невозможны без учета их производительности. Для того чтобы такой учет был по возможности объективным, необходимы количественные метрики для измерения производительности.

Наиболее важными интегральными (т. е. учитывающими несколько различных факторов) метриками являются *пропускная способность* и *время отклика* системы. Обе характеристики измеряются на определенной нагрузке системы. Обе метрики имеют смысл для очень широкого класса систем; уточнение

того, какие нагрузки имеет смысл рассматривать, зависит, конечно, от класса системы и от требований к ней. Для систем управления базами данных это может быть некоторая смесь запросов или других действий разной сложности. Когда такая нагрузка определена, можно измерить среднее количество подобных действий, выполняемых за единицу времени (пропускная способность), или среднее время выполнения одного действия (время отклика). Во многих случаях имеет смысл оценивать время отклика отдельно для каждого класса действий (в зависимости от их сложности для системы).

Необходимо подчеркнуть, что для достижения высокой пропускной способности может потребоваться конфигурация системы, отличающаяся от конфигурации, необходимой для достижения низкого времени отклика, и улучшение одной из этих характеристик совсем не обязательно приводит к улучшению другой. Этот факт можно пояснить следующим образом. Для получения высокой пропускной способности следует максимально использовать имеющиеся вычислительные ресурсы, что может приводить к росту очередей заданий, ожидающих выполнения, и, следовательно, к увеличению времени отклика за счет ожидания в очереди. С другой стороны, для сокращения времени отклика необходимо сократить время ожидания, в том числе в очередях, поэтому вычислительные ресурсы должны работать с неполной нагрузкой.

Для времени отклика важно также различать измерение на стороне клиента и на стороне сервера. Время выполнения не очень сложного запроса на сервере может оказаться значительно меньше времени пересылки текста запроса и возврата результата по вычислительной сети. В связи с этим может быть важно оценивать время отклика не для отдельных запросов или других операций с базой данных, а для каких-либо функций приложения в целом. Например, для веб-приложений наиболее важным вариантом времени отклика является время, необходимое для генерации HTML-страницы в ответ на запрос пользователя.

Для параллельных систем наиболее важной характеристикой является *масштабируемость* (scalability). В действительности масштабируемость не является отдельной характеристикой, а показывает, как изменяется другая характеристика при изменении нагрузки и размеров вычислительной системы. Можно говорить о масштабируемости пропускной способности или масштабируемости времени отклика, но не о масштабируемости вообще. Для того чтобы оценить масштабируемость, необходимо оценить какую-либо характеристику для системы, состоящей из одного вычислителя, на определенном объеме базы данных и определенной нагрузке и ту же самую характеристику для системы,

в которой количество вычислителей, количество данных и нагрузка (число запросов, например) — все увеличены в N раз. Тогда масштабируемость выражается отношением второго значения к первому. Поведение масштабируемости по пропускной способности показано на рис. 1.8.1.

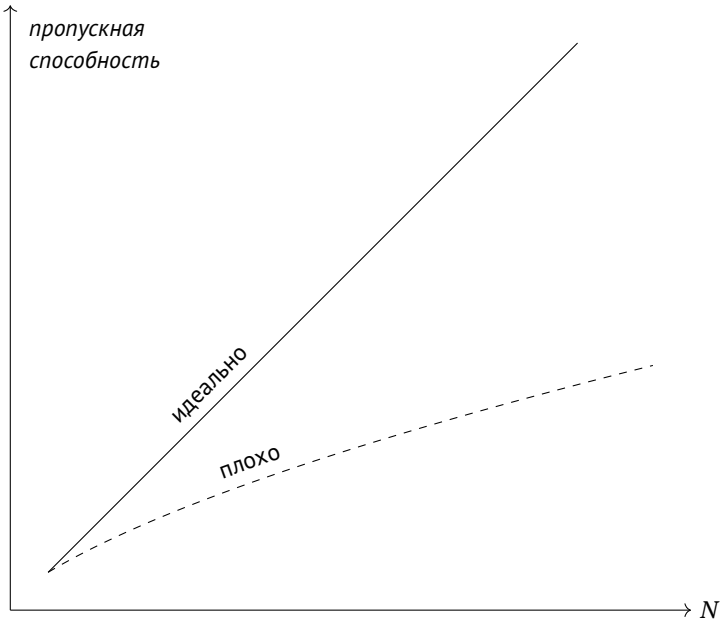


Рис. 1.8.1. Масштабируемость пропускной способности

Идеальное поведение масштабируемости по пропускной способности может быть представлено линейной зависимостью от N: система, содержащая в N раз больше оборудования и данных, способна обрабатывать в N раз больше запросов. В реальности такая масштабируемость недостижима, так как некоторая часть вычислительных ресурсов необходима для синхронизации работы параллельных вычислителей.

Для масштабируемости по времени отклика идеальная зависимость представляется константой: при увеличении количества вычислителей, объема данных и потока запросов время отклика не возрастает. Так же как и для пропускной способности, идеальная масштабируемость по времени отклика практически недостижима. Поведение масштабируемости по времени отклика иллюстрируется рис. 1.8.2.

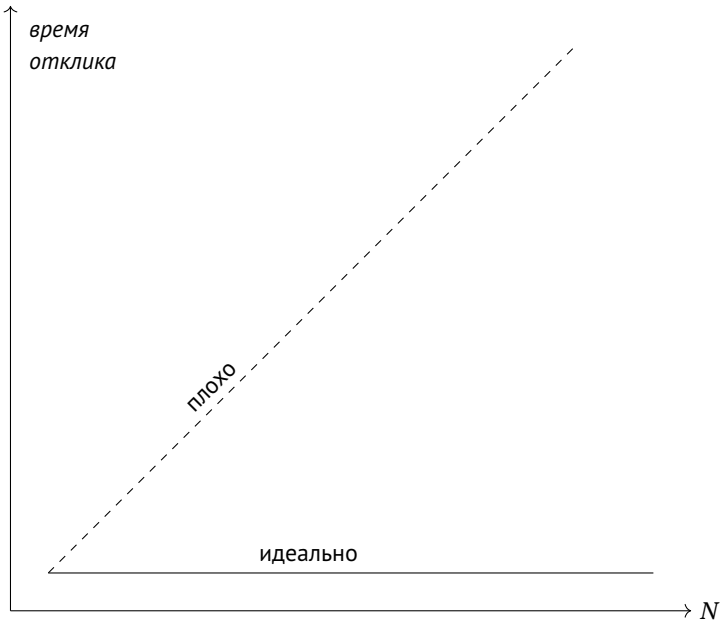


Рис. 1.8.2. Масштабируемость времени отклика

Еще одной характеристикой параллельных систем является *ускорение*, которое измеряется как отношение времени отклика на системе с одним вычислителем ко времени отклика на системе из N вычислителей.

Кроме характеристик производительности, имеется ряд других характеристик, из которых наиболее важной является *доступность* (availability). Доступность определяется как отношение времени нормальной работы системы к длине интервала времени, в течение которого измеряется доступность. Одновременное достижение высоких значений всех характеристик обычно требует значительного усложнения и удорожания системы.

Интегральные метрики, рассмотренные выше, полезны для оценки работы системы в целом, однако они мало полезны для оценки эффективности отдельных операций или запросов приложения. Поэтому в самой СУБД применяются метрики, значения которых можно предсказывать на основе информации об алгоритмах выполнения операций и статистических характеристик хранимых данных. Значением таких метрик является количество некоторых вычислительных ресурсов, необходимых для выполнения операции, например процессорное время или количество операций обмена с внешними устройствами,

а в случае параллельных или распределенных систем — еще и нагрузка на вычислительную сеть, которая может выражаться количеством сообщений, объемом передаваемых данных, количеством синхронных сообщений (с ожиданием ответа) и т. п.

Во многих случаях запросы приложений, сформулированные на высокоуровневом декларативном языке, могут быть выполнены многими различными способами, эквивалентными по результату, но использующими разное количество вычислительных ресурсов. Высокопроизводительные СУБД в таких случаях выбирают способ выполнения (план), для которого необходимое количество ресурсов минимально.

1.9. Создание приложений, взаимодействующих с базой данных

Современные СУБД поставляются с инструментами для создания и ведения баз данных. Эти инструменты предназначены для администраторов баз данных и помогают решать типовые задачи, такие как создание и изменение таблиц, редактирование записей, разграничение доступа и управление резервными копиями. Несмотря на то что эти программы обладают графическим интерфейсом, большинство задач удобнее решать, используя команды языка SQL.

Разумеется, такой интерфейс не подходит для бизнес-приложений. Система, ориентированная на массового пользователя, должна быть настолько понятной, чтобы пользователь, хорошо разбирающийся в предметной области, мог без всякого обучения начать с ней работать. Такие системы скрывают доступ к данным за графическим интерфейсом, который позволяет эффективно решить задачу и максимально ограждает пользователя от ошибок.

Для создания бизнес-приложений используется клиент-серверная архитектура. Ядро СУБД работает на сервере, а прикладная программа — на клиенте, причем сам клиент может быть сложным и состоять из нескольких уровней. Бизнес-логика может быть реализована как на сервере в виде хранимых процедур, так и на клиенте с использованием высокоуровневого языка программирования. Поскольку данные на клиенте и на сервере представлены по-разному, возникает проблема *потери соответствия* (impedance mismatch).

В случае объектно-ориентированных языков программирования и реляционных баз данных такая проблема называется объектно-реляционной потерей

соответствия. Реляционные базы данных не поддерживают основных концепций объектно-ориентированной парадигмы. Наибольшие затруднения при трансформации объектов в таблицы вызывает несоответствие системы типов, отображение наследования и многозначных связей, а также поддержка навигации между объектами.

Объектно-ориентированные языки пытаются решить проблемы при помощи каркасов объектно-реляционных отображений (object-relational mapping frameworks). Такие каркасы представляют собой библиотеки, написанные на объектно-ориентированном языке программирования. Разработчик приложения работает с привычными объектными моделями, которые автоматически преобразуются в таблицы и наоборот. Например, при операции сохранения каркас получает объект, отображает его в соответствующие таблицы и формирует SQL-запрос для вставки записи. При операции чтения каркас получает идентификатор объекта, формирует SQL-запрос для выборки данных, отображает найденные записи в объекты и возвращает их приложению.

Использование каркасов объектно-реляционных отображений ускоряет разработку, поскольку программисту не требуется глубоко знать ни SQL, ни реляционную теорию. Однако такие каркасы практически не используют специфические особенности конкретной СУБД, из-за чего тонкая настройка запросов становится невозможна. По-видимому, хорошим решением является использование каркасов для стандартных операций и чистого SQL для сложных запросов.

Попытка решить проблему несоответствия стала одной из причин создания баз данных NoSQL, которые представляют собой альтернативу реляционным СУБД. Такие базы данных не имеют структурированной схемы и работают напрямую с объектами. Преимущества и недостатки баз данных NoSQL рассматриваются в главе 9.

1.10. Итоги главы

В этой главе определены основные понятия, связанные с системами управления базами данных, обсуждены основные требования к таким системам, как они были определены исторически и как они эволюционировали на протяжении десятилетий существования СУБД. Более детально многие из этих тем разбираются в последующих главах.

1.11. Контрольные вопросы

Вопрос 1.1. Какие основные требования предъявляются к системам управления базами данных?

Вопрос 1.2. Какие основные компоненты содержит обобщенная трехуровневая модель данных ANSI/SPARC?

Вопрос 1.3. Каковы основные характеристики языков запросов в современных СУБД?

Вопрос 1.4. Что означает термин «независимость данных»?

Вопрос 1.5. Какие преимущества возникают при использовании независимости данных?

Вопрос 1.6. Что означает термин «согласованность данных»?

Вопрос 1.7. Что понимается под ограничением целостности в системах управления базами данных?

Вопрос 1.8. Как трактуются понятия безопасности и разграничения доступа в современных системах управления данными?

Вопрос 1.9. Какие основные метрики используются для оценки производительности?

Вопрос 1.10. Что такое архитектура клиент — сервер? Как распределяются программные компоненты?

Вопрос 1.11. Что называют объектно-реляционной потерей соответствия?

Глава 2

Теоретические основы БД

2.1. Модели данных

Будем называть *моделью данных* систему взаимосвязанных понятий и правил, предназначенную для описания структур и свойств данных, используемых (хранимых и обрабатываемых) в информационной системе. В этом курсе предполагается, что такая информационная система пользуется услугами некоторой СУБД. Поэтому можно также сказать, что модель данных задает способ описания схемы базы данных.

В состав развитой модели данных входят:

- способы описания данных: какие базовые (примитивные) типы можно использовать, каким образом строить сложные структуры данных из более простых;
- способы описания взаимосвязей между объектами данных;
- средства задания ограничений целостности;
- способы конструирования операций, которые можно использовать в рамках модели данных.

Не все модели данных, используемые в современных СУБД, содержат все перечисленные составляющие в развитой форме, однако все эти составляющие модели данных необходимы, для того чтобы удовлетворить основные требования, в частности рассмотренные в главе 1. Отсутствие каких-либо элементов или средств в модели данных, реализуемой в СУБД, обычно означает, что эти элементы должны быть реализованы на уровне приложений, использующих такую СУБД.

Например, если в модели данных СУБД отсутствует возможность описания сложных структур данных (скажем, любое значение рассматривается как последовательность байтов), то фактическая структура таких значений должна быть определена в приложении. Обычно такие данные трудно использовать

в других приложениях, и возникает необходимость преобразования в иную модель данных.

2.1.1. Идентификация и изменяемость

Многие свойства и особенности модели данных определяются тем, каким образом в рамках этой модели различаются используемые в ней объекты данных. Некоторые модели предусматривают наличие выделенного способа идентификации (например, объектного идентификатора в объектно-ориентированных моделях данных). В этом случае объекты данных считаются совпадающими, если у них совпадают идентификаторы. Другие элементы данных, входящие в тот же объект, образуют его состояние, которое может быть различным, и, следовательно, в таких моделях объекты обладают изменяемостью. Далее мы покажем, что не все модели данных обладают такой возможностью.

Многообразие методов идентификации можно условно разделить на следующие категории:

По естественным признакам объекта реального мира, который описывается объектом данных. Примером естественных идентификаторов может служить трехбуквенный код аэропорта, номер бронирования или биометрические свойства.

По искусственному значению (суррогату), которое генерируется информационной системой. Заметим, что суррогатный идентификатор, созданный в одной системе, может использоваться как естественный идентификатор в другой. В качестве примера можно назвать номер документа, идентифицирующего личность.

По связи объекта с другим, уже идентифицированным объектом. Этот метод полезен, для того чтобы различить объекты, которые в рамках модели невозможно различить другим способом. Например, «правое переднее колесо легкового автомобиля» идентифицирует колесо по его месту. Поскольку колеса взаимозаменяемы, в рамках модели они могут описываться совпадающими значениями всех атрибутов.

Другой пример такого типа идентификации можно получить, если требуется, чтобы представление некоторого объекта не зависело от того, в каком месте в памяти компьютера данный объект находится. В этом случае различные копии объекта можно различать по адресу в памяти, где находится копия.

Выбор метода идентификации особенно важен для проектирования базы данных и информационной системы, ее использующей. Для моделей данных важно наличие какого-либо идентификатора, а не его природа.

Если же способ идентификации не выделен, то единственным способом проверки на равенство является проверка на совпадение значений всех элементов данных. В таких моделях данных объекты, различающиеся значением хотя бы одного элемента, считаются различными. Следовательно, в таких моделях объекты изменяться не могут. Можно заменить объект на другой, но изменить невозможно.

Среди рассматриваемых в этой главе моделей данных в теоретической реляционной модели данных применяется идентификация по значениям атрибутов, а в объектных моделях данных и в модели данных «сущность — связь» предполагают наличие выделенных для идентификации атрибутов объектов.

Не следует думать, что поддержка изменяемости всегда является достоинством модели. Далее в книге мы покажем, что наиболее широко используемая модель данных — реляционная — не поддерживает изменяемость. Это утверждение может показаться парадоксальным. Читателю, у которого оно вызывает протест, придется потерпеть до раздела 2.2, в котором приведено развернутое объяснение.

Вследствие неизменяемости в рамках реляционной модели данных удается построить мощные средства выполнения запросов. Создать аналогичные средства для моделей, поддерживающих изменяемость, не удается, несмотря на многолетние усилия исследователей и практиков. В некоторых моделях данных строятся две системы типов: отдельно для изменяемых и для неизменяемых объектов, в частности это было сделано в проекте стандарта объектных баз данных [13].

В некоторых случаях идентификаторы, явно не требуемые в рамках модели данных, могут быть выявлены на основе ограничений целостности. Это используется в теоретической реляционной модели данных для определения понятия ключа.

В реальности, конечно, практически все объекты обладают изменяемостью, поэтому при проектировании системы очень важно правильно выбрать способ идентификации, позволяющий взаимно-однозначно сопоставлять объекты реального мира с их представлениями в информационной системе.

2.1.2. Навигация и поиск по значениям

Поиск данных для обработки выполняется в любой системе управления базами данных, однако в рамках разных моделей данных он осуществляется поразному. Можно выделить следующие основные классы поисковых операций:

Навигационный доступ (переходы от одного объекта к другим происходят с помощью ссылок). Примером, не связанным непосредственно с базами данных, может служить использование URL для навигации в интернете.

Поиск по значениям (ассоциативный поиск). Примером, также непосредственно не связанным с базами данных, могут быть поисковые средства Яндекс или Google, хотя ассоциативный поиск вовсе не ограничивается теми видами текстового поиска, которые реализованы в этих системах.

Навигационный доступ использует явно заданные связи между объектами — например от текущего обрабатываемого объекта можно перейти к другому объекту по идентификатору, значение которого содержится в одном из атрибутов текущего объекта. Этот способ поиска характерен для объектно-ориентированных моделей данных, в том числе для объектно-ориентированных языков программирования.

Результатом поиска по значениям обычно является набор объектов данных, удовлетворяющих условиям поиска. Такой поиск может не использовать заранее заданные связи, и часто результат содержит несколько объектов данных. Этот способ часто используется в декларативных языках запросов, в том числе и в теоретической реляционной модели, и в используемых на практике системах управления базами данных, в которых реализован язык запросов SQL (обсуждаемый в главе 4).

Отметим, что навигация возможна, даже если модель данных не предусматривает явную идентификацию объектов данных. Например, можно по текущему элементу обрабатываемой коллекции найти коллекцию взаимосвязанных с ним объектов данных другого типа.

Не существует четкого различия между способами поиска, и многие модели допускают использование операций того и другого класса. Поэтому выбор способа поиска является не только свойством модели, но и отражает стиль программирования приложений.

2.1.3. Объекты и коллекции объектов

Гранулярность доступа определяет, что является единицей обмена при доступе к данным. Операции, определенные в модели данных, могут быть ориентированы на обработку отдельных объектов данных или на массовую обработку. Как правило (хотя и не обязательно) навигационный доступ предполагает обработку отдельных объектов данных, а поиск по значениям чаще всего предполагает массовую обработку.

Свойство гранулярности модели данных определяет, каким образом происходит взаимодействие между сервером базы данных и приложением и, следовательно, должно учитываться при проектировании приложения.

На уровне моделей данных невозможно четко отделить модели одного типа от моделей другого, однако в реализациях это различие становится очень существенным. Системы, ориентированные на обработку отдельных объектов данных, как правило, оказываются неэффективными при массовой обработке, и наоборот, системы, достигающие очень высокой производительности при массовой обработке, могут быть крайне неэффективны, если они используются для обработки большого количества отдельных объектов.

Среди моделей, рассматриваемых в этой главе, как теоретическая реляционная модель, так и ее практически реализованные варианты ориентированы в первую очередь на массовую обработку. Системы, основанные на реляционных моделях данных, в том числе PostgreSQL, обеспечивают высокую эффективность массового извлечения данных, но могут оказаться значительно менее эффективными при обработке отдельных объектов данных.

2.1.4. Свойства моделей данных

Свойства моделей данных, перечисленные в этом разделе, могут показаться абстрактными, однако в действительности учет этих свойств оказывает очень существенное влияние на качество проектируемых систем. Более того, эти свойства важны для сравнительной оценки достоинств и недостатков той или иной модели данных.

Мы не рассматриваем детально ранние модели данных (иерархическую и сетевую), потому что по своим характеристикам и свойствам они похожи на объектные модели данных. Тем не менее заметим, что для всех таких моделей

(включая объектную) характерны навигационный способ доступа и ориентация на обработку отдельных объектов. В соответствии с этим как критерии эффективности, так и особенности реализации были ориентированы именно на эти свойства. Появление в начале 1970-х гг. реляционной модели данных (рассматриваемой в следующем разделе) вызвало негативную реакцию многих практиков, так как в терминах общепринятых в то время критериев реляционная модель не могла обеспечить приемлемую эффективность реализации.

Спустя два десятилетия стало очевидно, что ориентация на обработку отдельных записей и навигацию не может обеспечить эффективную массовую обработку данных, и появились технологии и реализации реляционных систем, значительно превосходящие по производительности ранние системы, но в терминах новых критериев.

Распространение в последние годы моделей данных с ограниченными функциями (часто объединяемых под зонтиком NoSQL) связано с отказом от массовой обработки объектов на уровне базы данных и, соответственно, с изменением требований к модели данных и критериев эффективности.

2.2. Реляционная модель данных

Обсуждение реляционной модели данных невозможно без небольшого исторического введения. Реляционная модель появилась в начале 1970-х гг. в работах Э. Кодда (Edgar Codd) и ряда других исследователей. Большую роль в популяризации идей реляционной модели данных сыграли работы К. Дейта (Christopher Date). В течение почти десятилетия модель разрабатывалась как чисто теоретический способ описания свойств коллекций данных и языков запросов. При этом теоретики подчеркивали, что реляционная модель данных не решает вопросы организации хранения данных, а практики уверенно писали о том, что эффективная реализация реляционной модели данных невозможна.

В последующее десятилетие, однако, были исследованы методы и структуры хранения и поиска, а также способы оптимизации запросов, позволившие создать системы управления базами данных, основанные на вариантах реляционной модели и обеспечивающие высокую эффективность массовой обработки. Фактически системы такого типа широко используются до настоящего времени. К этому же классу систем относится и PostgreSQL.

Необходимо подчеркнуть, что радикальное изменение взглядов на возможность создания высокоэффективных систем на основе реляционной модели

в значительной мере определяется не технологиями хранения и индексирования данных, а изменением критериев эффективности и требований к системам: в 70-е гг., говоря об эффективности, подразумевали эффективность доступа к отдельным объектам данных и навигации между ними, а начиная с 80-х гг., наиболее важными стали эффективность массовой обработки и поиск по значениям атрибутов (т. е. ассоциативный поиск). Именно поэтому абстрактные свойства моделей данных, рассмотренные выше в этой главе, представляются весьма важными.

Далее в этом разделе мы рассмотрим теоретическую реляционную модель данных и кратко обсудим отличия практических реализаций этой модели в системах, основанных на языке SQL. Более подробное изложение модели данных языка SQL отложим до главы 4.

2.2.1. Основные понятия реляционной модели данных

Домены

Неформально домен можно описать как множество значений, обладающих некоторыми общими свойствами. Например, можно говорить о доменах целых или вещественных чисел, домене длин, домене текстов и др.

В теоретической реляционной модели данных доменами называются множества некоторых значений. При этом предполагается, что все значения, находящиеся в доменах, являются скалярными константами, т. е. структура значений, даже если она существует, не рассматривается в рамках модели, и любые изменения приводят просто к новым значениям, а не к измененным старым. Кроме этого, требуется, чтобы на любом домене было определено бинарное отношение равенства. Это значит, что для любых двух значений из домена можно (с помощью этого отношения) определить, совпадают эти значения или нет. Необходимость в такой проверке возникает, когда значения приходят из разных источников, например значение, хранимое в базе данных, может сравниваться с константой в запросе.

Кроме этого, на доменах могут быть определены другие отношения, операции и функции. Например, на доменах, содержащих числовые значения, можно рассматривать:

- отношения упорядочивания $<$, $\leqslant$, $>$, $\geqslant$;
- арифметические операции, например $+$, $-$, $\times$, $\div$;

- функции, принимающие скалярные значения, например тригонометрические и другие.

Могут также быть определены функции, принимающие значения в другом домене, или функции нескольких аргументов из разных доменов. Например, на домене текстов можно рассматривать функцию, возвращающую длину текста, и функцию, выделяющую подстроку указанной длины, начиная с указанной позиции.

Любые такие дополнительные отношения, функции и операции могут использоваться в рамках реляционной модели для вычисления новых значений при описании запросов, однако необходимым для самой реляционной модели является только отношение равенства.

Можно сказать, что в языках программирования понятию домена соответствует понятие абстрактного типа данных: как и абстрактный тип данных, домен определяет операции, которые можно выполнять над входящими в него значениями. Отличие состоит в том, что (как уже указано выше) значения, входящие в домен, считаются скалярными, т. е. не могут быть, например, массивами.

Домены могут быть взаимосвязаны. Например, домен целых чисел является подмножеством домена рациональных, или, в терминах операций и функций, один домен может быть специализацией другого (т. е. соответствовать подтипу абстрактного типа). Однако в реляционной теории все домены рассматриваются как независимые. Конечно, можно определить функции, преобразующие значения из одного домена в другой, например числовые значения можно записывать как текстовые строки, но в теоретической реляционной модели данных такие преобразования не нужны.

Ранние реализации реляционной модели допускали использование только ограниченного набора доменов, включающего числовые домены, домены дат и времени, домен литерных строк и т. п. Такое ограниченное понимание доменов вызвало критику и утверждения об ограниченности реляционной модели. Современные реализации реляционной модели, в том числе в PostgreSQL, предусматривают использование любых доменов, как это и предполагалось реляционной теорией с самого начала.

Например, можно определить домены длин и весов. Значения в этих доменах, разумеется, будут числовыми, однако наборы операций будут отличаться: длины можно складывать и вычитать, но результат перемножения длин будет уже в другом домене, возможно, в домене площадей. Умножать длины можно только на значение из числового домена, при этом результат тоже будет длиной.

Точно так же можно складывать значения веса, но вряд ли целесообразно определять операцию сложения веса с длиной.

Ничто не мешает использовать домен, содержащий точки плоскости, при этом координаты точки (декартовы или полярные) будут вычисляться с помощью функции, принимающей значения в подходящем домене.

По историческим причинам возможности определения доменов рассматриваются как расширения реализаций реляционной модели. Возможности, предоставляемые системой PostgreSQL, обсуждаются в главе 8.

Отношения, атрибуты и кортежи

Перейдем к обсуждению центрального понятия теоретической реляционной модели — понятия *отношения*.

Формально отношение определяется как n-местный (или n-арный) предикат, т. е. функция с n аргументами, принимающая булево значение: истина (true) или ложь (false). При этом предполагается, что для каждого аргумента задан домен, из которого могут выбираться значения этого аргумента.

Напомним, что в математике определение функции не предполагает наличия какой-либо формулы или алгоритма, позволяющего вычислить значение этой функции по значениям аргументов. Это справедливо и для предикатов, рассматриваемых в реляционной теории.

В отличие от обычной математической нотации в реляционной теории используются именованные, а не позиционные (т. е. занумерованные натуральными числами) аргументы. Именованные аргументы отношения называются *атрибутами отношения*. Использование имен позволяет при необходимости записывать атрибуты в произвольном порядке.

Имена атрибутов одного отношения обязательно должны быть попарно различными. С каждым атрибутом связан некоторый домен, из которого выбираются значения этого атрибута (это уже было отмечено выше), при этом один и тот же домен может быть связан с несколькими различными атрибутами одного отношения.

Множество всех атрибутов отношения называется *схемой отношения*.

Совокупность из n значений, по одному значению (из соответствующего домена) для каждого атрибута, называется *кортежем*. Можно сказать, что кортеж —

это набор значений аргументов, для которого можно вычислить предикат, соответствующий отношению. Говорят, что кортеж принадлежит отношению, если предикат принимает истинное значение на этом кортеже.

Семантика отношений состоит в том, что они описывают зависимости между своими атрибутами. Считается, что значения в кортеже взаимосвязаны, если этот кортеж принадлежит отношению. Если атрибуты соответствуют свойствам некоторого класса объектов реального мира, то принадлежность кортежа отношению может означать, что объект с такими значениями свойств существует в реальности. Другими словами, каждый кортеж, принадлежащий отношению, выражает истинность некоторого факта.

Количество кортежей в каком-либо отношении называется *кардинальностью* этого отношения.

Проиллюстрируем приведенные выше определения примерами.

Предположим, что отношение exams имеет атрибуты {name, course, grade}, принимающие значения в соответствующих доменах. Тогда истинное значение предиката на следующих кортежах указывает на то, что упомянутые студенты получили указанные оценки по этим курсам.

exams (name := 'Анна', course := 'Базы данных', grade := 5)

exams (name := 'Анна', course := 'Анализ данных', grade := 5)

exams (course := 'Анализ данных', grade := 4, name := 'Виктор')

exams (name := 'Нина', grade := 5, course := 'Базы данных')

Напомним, что предикаты (отношения) обычно не задаются какими-либо формулами. Поэтому, для того чтобы записать предикат, используется принятый в математике табличный способ представления функций: для каждого значения аргумента выписывается соответствующее ему значение функции. В нашем случае функция (предикат) имеет только два значения, поэтому достаточно выписать только те значения аргументов (кортежи), которые входят в отношение (или, что эквивалентно по приведенному выше определению, предикат принимает истинное значение).

Каждый кортеж состоит из n значений, поэтому отношения принято записывать в виде таблиц, содержащих n колонок, соответствующих атрибутам отношения (и озаглавленных именами атрибутов), а каждая строка таблицы соответствует одному кортежу. При этом порядок, в котором расположены кортежи в таблице, не имеет никакого значения.

Поскольку каждый кортеж представляет истинность некоторого факта, повторное включение любого кортежа не дает никакой новой информации. В теоретической реляционной модели данных предполагается, что все кортежи, входящие в отношение, различны, и поэтому совокупность кортежей отношения является множеством (в математическом смысле).

В табличном представлении отношение exams, приведенное выше, может выглядеть, как показано на рис. 2.2.1.

exams

name	course	grade
Анна	Базы данных	5
Анна	Анализ данных	5
Виктор	Анализ данных	4
Нина	Базы данных	5

Рис. 2.2.1. Пример отношения в табличной записи

Отметим, что представление отношений таблицами возможно только для отношений, содержащих конечное число кортежей.

Во многих реализациях реляционной модели данных отношения называются *таблицами,* атрибуты — *колонками* или *столбцами*, а кортежи — *строками.* В дальнейшем термины «колонка» и «столбец» будут использоваться как синонимы.

В отличие от теоретической реляционной модели данных, табличные реализации этой модели в системах управления базами данных допускают хранение совпадающих строк в таблицах, если уникальность не задана как ограничение целостности в схеме таблицы.

2.2.2. Реляционная алгебра

Успех реляционной модели данных, интенсивно используемой на протяжении десятилетий, обусловлен тем, что в рамках этой модели определены мощные и выразительные языки манипулирования данными. Один из таких языков строится на основе операций, позволяющих определять новые отношения из уже имеющихся. Это делает возможным использование результатов выполнения операций в качестве аргументов для выполнения следующих операций, т. е. задание требуемых вычислений в форме выражений.

В математике подобные системы операций (определенных на некотором множестве) называются *алгебрами*. Операции могут быть частичными, например операция деления чисел определена, только если делитель отличается от нуля. Набор операций, рассматриваемых в этом подразделе, называется *реляционной алгеброй*. Эти операции определены на множестве всех возможных конечных отношений.

Почти все операции реляционной алгебры являются частичными, т. е. они определены не для любых аргументов или их комбинаций (для бинарных операций). В некоторых случаях, для того чтобы операция стала выполнимой, достаточно выполнить переименование атрибутов. Для этой цели можно ввести операцию, которая изменяет имена атрибутов в схеме отношения, но не меняет никаких значений в кортежах, входящих в отношение, и не меняет доменов, связанных с атрибутами.

Теоретико-множественные операции

Поскольку кортежи, входящие в отношение, образуют множества, можно применять обычные теоретико-множественные операции к любой паре отношений, имеющих одинаковые схемы. Эти операции тесно связаны с логическими операторами булевой алгебры.

В реляционной алгебре используются следующие операции:

Объединение UNION: в результат включаются кортежи, входящие в первый *или* во второй аргумент.

Пересечение INTERSECT: результат содержит только кортежи, входящие в первый *и* второй аргументы.

Разность EXCEPT: результат содержит кортежи, входящие в первый *и не* входящие во второй аргумент. В некоторых вариантах алгебры эта операция называется MINUS.

Используя эти операции, можно строить производные, определяемые алгебраическими выражениями.

Например, операция симметричной разности отношений R и S, соответствующая логической операции XOR (exclusive or, исключающее или), может быть определена как

(R UNION S) EXCEPT (R INTERSECT S)

или как

(R EXCEPT S) UNION (S EXCEPT R).

Заметим, что в реляционной алгебре нет теоретико-множественной операции дополнения. Это связано с тем, что отношения должны быть конечными множествами, а многие домены бесконечны.

Унарные операции

Операции с одним аргументом (унарные операции) позволяют выделить из отношения ту информацию, которая нужна данному приложению при данном выполнении.

Операция проекции PROJ включает в результат подмножество атрибутов отношения, переданного в качестве аргумента. На рис. 2.2.2 показан результат проекции отношения exams на подмножество атрибутов {name, course}. Такой результат показывает только факт сдачи экзамена, но не оценку.

PROJ [name, course] exams

name	course
Анна	Базы данных
Анна	Анализ данных
Виктор	Анализ данных
Нина	Базы данных

Рис. 2.2.2. Реляционная операция проекции

В результате исключения части атрибутов при выполнении операции проекции может оказаться, что различные кортежи исходного отношения станут совпадать по значению. Реляционная операция проекции обязательно исключает возникшие дубликаты. На рис. 2.2.3 проекция на атрибуты {course, grade} содержит меньшее количество кортежей, чем исходное отношение.

Реализации реляционной модели выполняют удаление дубликатов, только если оно явно задано в запросе.

Операция фильтрации FILTER строит новое отношение, включая в него строки исходного отношения, удовлетворяющие условию, выраженному логической

PROJ [course, grade] exams

course	grade
Базы данных	5
Анализ данных	5
Анализ данных	4

Рис. 2.2.3. Проекция с удалением дубликатов

формулой. В первых работах по реляционной модели данных эта операция называлась ограничением, потому что она соответствует понятию ограничения функции (в этом случае отношение рассматривается как предикат, т. е. функция). Зачастую эта операция называется селекцией, однако такое наименование создает ложную связь с оператором SELECT языка запросов SQL, который обладает значительно более мощными возможностями.

Условие в операции фильтрации строится по следующим правилам:

- Простое условие представляется атрибутом, который сравнивается с константой или с другим атрибутом того же отношения с помощью бинарного отношения равенства, определенного на домене этого атрибута (или атрибутов). Поскольку такое отношение определено на любом домене, любой атрибут можно сравнивать с константой из того же домена.
- Ранее определенные по этим правилам условия можно соединять логическими операторами AND, OR, NOT.
- Условие может быть заключено в круглые скобки.

На рис. 2.2.4 показан результат выполнения операции фильтрации. Заметим, что результатом выполнения фильтрации может быть пустое отношение.

FILTER [course='Анализ данных' AND grade=5] exams

name	course	grade
Анна	Анализ данных	5

Рис. 2.2.4. Операция фильтрации

Произведение и соединение

Операция прямого (или декартова) произведения PROD строит все пары кортежей из первого и второго аргументов и по каждой такой паре создает кортеж результата, содержащий все значения атрибутов кортежей, входящих в эту пару. Необходимо, чтобы все имена атрибутов аргументов данной операции были различными. Для этого перед выполнением операции надо переименовать атрибуты с совпадающими именами в одном из отношений.

На рис. 2.2.5 показано отношение courses, содержащее информацию о зачетных единицах, которые можно получить за курс, а на рис. 2.2.6 представлено произведение отношений.

courses

title	credits
Базы данных	5
Анализ данных	10

Рис. 2.2.5. Отношение courses

exams PROD courses

name	course	grade	title	credits
Анна	Базы данных	5	Базы данных	5
Анна	Анализ данных	5	Базы данных	5
Виктор	Анализ данных	4	Базы данных	5
Нина	Базы данных	5	Базы данных	5
Анна	Базы данных	5	Анализ данных	10
Анна	Анализ данных	5	Анализ данных	10
Виктор	Анализ данных	4	Анализ данных	10
Нина	Базы данных	5	Анализ данных	10

Рис. 2.2.6. Прямое произведение отношений

Очевидно, что кардинальность произведения отношений равна произведению кардинальностей аргументов. Этот факт используется для оценки стоимости вычислений алгебраических выражений.

Само по себе произведение нельзя считать полезной операцией, потому что результат содержит кортежи, отражающие истинные, но не обязательно связанные между собой факты. Намного более полезный результат дает операция соединения JOIN, представляющая собой произведение с последующей фильтрацией. Поскольку результатом произведения является одно отношение, условие фильтрации может в этом случае содержать и сопоставлять атрибуты из разных аргументов, выделяя таким образом взаимосвязанные кортежи из разных отношений.

В алгебраических выражениях для обозначения операции соединения наряду с JOIN используется знак ⋈.

Для соединений общего вида (часто называемых тета-соединениями) условие фильтрации (часто обозначаемое буквой θ) может быть любым, однако обычно используются условия, представляющие собой конъюнкцию условий на равенство значений пар атрибутов. Такие соединения называются эквисоединениями. Поскольку значения сравниваемых атрибутов в получаемом отношении будут одинаковыми, целесообразно выполнить проекцию, которая исключит дублирующие значения.

Пример соединения отношений exams и courses показан на рис. 2.2.7. Поскольку условие соединения состоит в проверке равенства значений атрибутов, это соединение является эквисоединением.

exams JOIN [course=title] courses

name	course	grade	title	duration
Анна	Базы данных	5	Базы данных	5
Нина	Базы данных	5	Базы данных	5
Анна	Анализ данных	5	Анализ данных	10
Виктор	Анализ данных	4	Анализ данных	10

Рис. 2.2.7. Соединение отношений

По определению операция соединения является производной, так как она выражается через произведение и фильтрацию. Несмотря на это, операция соединения рассматривается отдельно, потому что она очень важна для приложений. Важно и то, что для операции соединения существуют намного более эффективные алгоритмы, чем алгоритмы, основанные на вычислении прямого произведения.

Существует еще несколько вариантов операции соединения; в этой книге они обсуждаются в контексте языка запросов SQL.

Свойства операций реляционной алгебры

Перечислим основные алгебраические тождества, справедливые для операций реляционной алгебры.

Коммутативность. Операции объединения, пересечения, произведения и соединения коммутативны.

Например:

R UNION $S = S$ UNION R,

R JOIN $S = S$ JOIN R.

Ассоциативность. Операции пересечения, объединения, произведения и соединения ассоциативны.

Например:

R UNION (S UNION T) = (R UNION S) UNION T,

R JOIN (S JOIN T) = (R JOIN S) JOIN T.

Дистрибутивность. Пары операций объединения, пересечения, произведения (или соединения) подчиняются дистрибутивным законам.

Пересечение дистрибутивно относительно объединения:

R INTERSECT (S UNION T) = (R INTERSECT S) UNION (R INTERSECT T).

Объединение дистрибутивно относительно пересечения:

R UNION (S INTERSECT T) = (R UNION S) INTERSECT (R UNION T).

Произведение и соединение дистрибутивны относительно операций объединения и пересечения:

R PROD (S UNION T) = (R PROD S) UNION (R PROD T);

R PROD (S INTERSECT T) = (R PROD S) INTERSECT (R PROD T).

Кроме этого, имеются тождества, связывающие унарные операции друг с другом и с бинарными: можно заменять одну операцию несколькими более простыми и обратно, заменять фильтрации теоретико-множественными операциями и др.

Существование алгебраических тождеств позволяет преобразовывать алгебраические выражения в эквивалентные. Результат вычисления эквивалентных выражений будет, конечно, одинаковым, однако сложность вычисления может отличаться очень значительно (на несколько порядков). Чтобы выполнить запрос эффективно, СУБД может выбрать среди эквивалентных способов записи запроса в виде выражения такой, выполнение которого требует меньшего количества вычислительных ресурсов (таких как процессорное время или количество операций обмена данными).

2.2.3. Другие языки запросов

Одним из основных требований к системам управления базами данных, перечисленных в главе 1, является наличие языка запросов. Желательно, чтобы такой язык был декларативным, т. е. позволял описывать, какой требуется результат, не указывая способа его вычисления. Реляционная алгебра не может считаться декларативным языком, так как алгебраическое выражение определяет порядок выполнения операций.

Существуют языки запросов более высокого уровня, позволяющие записать требования к результату выполнения запроса в виде набора логических условий. Такие языки называются исчислениями. Запрос в исчислении представляется набором правил.

В левой части правила (которая называется *головой* правила) обычно определяется схема отношения, которое является результатом вычисления правила, а в правой (в *теле* правила) — условие, которому должны удовлетворять кортежи, включаемые в результат вычислений.

Условие, размещаемое в теле правила, формируется следующим образом:

- простые условия задаются предикатами, определенными на доменах; в качестве аргументов предикатов могут использоваться переменные или константы;
- условие может быть заключено в скобки;

- условие может быть построено из более простых условий с помощью логических операций $\wedge$, $\vee$, $\neg$;
- условие, содержащее свободные переменные, может быть замкнуто с помощью кванторов всеобщности $\forall$ или существования $\exists$;
- свободные переменные могут использоваться в голове правила в качестве атрибутов результирующего отношения.

Различают исчисления с переменными на кортежах и на доменах. Для переменных на кортежах указывается принадлежность переменной к отношению; при этом значения атрибутов обозначаются именем переменной, за которым следует имя атрибута, отделенное точкой. Голова и тело правила разделяются знаком $:-$.

Например, соединение двух отношений, представленное на рис. 2.2.7, может быть записано следующей формулой в исчислении:

$$\begin{gathered}\mathsf{JoinResult}\ (\mathsf{x.name},\ \mathsf{x.course},\ \mathsf{x.grade},\ \mathsf{y.title},\ \mathsf{y.credits})\ :- \\ \mathsf{x} \in \mathsf{exams}\ \wedge\ \mathsf{y} \in \mathsf{courses}\ \wedge\ \mathsf{x.course} = \mathsf{y.title}\end{gathered}$$

В исчислении на доменах переменные принимают значения в доменах. Из таких переменных и констант формируются кортежи, принадлежность которых к отношениям базы данных становится одним из простых условий в выражении, описывающем запрос.

Например, тело правила

$$\{\mathsf{name},\ \mathsf{course},\ \mathsf{grade}\} \in \mathsf{exams}\ \wedge\ \mathsf{grade} < 5$$

описывает запрос на выборку результатов экзаменов с оценкой ниже отличной.

В отличие от исчислений общего вида в реляционном исчислении не допускается рекурсия, т. е. отношение, находящееся в голове правила, не может использоваться в его теле, а если правил несколько, то не допускается также взаимная рекурсия.

Существует несколько вариантов реляционных исчислений, однако можно доказать, что все они эквивалентны друг другу и каждый из них эквивалентен реляционной алгебре. В данном случае эквивалентность означает, что любой

запрос, который можно записать на одном из этих языков, можно также выразить и на другом эквивалентном языке. Ни исчисления, ни доказательство эквивалентности не рассматриваются в этой главе.

Применяемый в реализациях язык запросов SQL занимает промежуточное положение между алгеброй и исчислением и позволяет использовать формы записи запросов, близкие как к алгебраическим выражениям, так и к формулам в исчислении. Важно отметить, что независимо от выбранного способа записи при подготовке запроса к выполнению он переводится в алгебраическое выражение, при этом среди эквивалентных выбирается такое выражение, выполнение которого будет вычислительно эффективно.

2.2.4. Особенности реляционной модели данных

Сопоставим свойства реляционной модели данных с характеристическими особенностями других моделей данных, которые обсуждаются в начале этой главы.

Семантика реляционной модели определяется тем, что кортежи соответствуют фактам, точнее, утверждениям об объектах реального мира. Роль объектов данных выполняют кортежи, которые в реляционной модели не имеют никакой специально выделенной идентификации. Поэтому формально кортежи не могут быть изменяемыми в рамках теоретической реляционной модели. Можно заменить одно значение на другое, но это будет другой кортеж, а не модификация старого.

Все операции реляционной алгебры в качестве результата создают новое отношение, поэтому реляционная модель, несмотря на свой достаточно абстрактный характер, ориентирована на массовую обработку данных.

Наконец, в реляционной алгебре предусмотрены только критерии поиска, использующие значения атрибутов. Более того, взаимосвязи между кортежами различных отношений также устанавливаются в операциях соединения на основе значений атрибутов. Поэтому организация навигационного доступа в этой модели требует дополнительных соглашений об использовании атрибутов, обычно выходящих за рамки этой модели.

Можно, однако, используя ограничения целостности (которые обсуждаются ниже), организовать и навигацию, и даже обработку отдельных кортежей.

2.2.5. Нормальные формы

Функциональные зависимости и ключи

В представленном выше изложении реляционной модели данных неявно предполагалось, что атрибуты отношения независимы или зависимости между атрибутами неизвестны. Однако значительная часть реляционной теории анализирует различные типы зависимостей, часть из которых рассматривается в этом подразделе.

Пусть X и Y — некоторые множества атрибутов одного отношения. Говорят, что Y *функционально зависит* от X, и пишут $X \to Y$, если для любой комбинации значений атрибутов из X может существовать только одна комбинация значений Y, входящая в отношение.

Другими словами, функциональная зависимость просто означает, что существует некоторая функция с областью определения X и принимающая значения в Y, которая определяет значения Y для любого кортежа, который может входить в рассматриваемое отношение. Подчеркнем, что для использования в реляционной теории важно только существование такой функции, но не способ ее вычисления.

Заметим, что если $Y \subset X$, то $X \to Y$, потому что любая комбинация значений атрибутов однозначно определяет сама себя. Такие зависимости называются *тривиальными* и существуют для любого множества атрибутов. В частности, все атрибуты отношения зависят от схемы этого отношения и каждый атрибут зависит сам от себя.

Нетривиальные зависимости не могут быть выведены формально, они должны отражать закономерности, которым подчиняются свойства объектов реального мира, представленных в СУБД. Подчеркнем, что такие закономерности должны быть справедливы для любого состояния базы данных, т. е. должны выполняться для всех без исключения объектов реального мира.

Например, в отношении exams оценка (grade) функционально зависит от пары атрибутов {name, course}, потому что каждый студент может иметь только одну итоговую оценку по любой дисциплине, хотя мы и не можем вычислить эту оценку по значениям атрибутов, от которых она зависит. В этом же отношении, очевидно, нет зависимости между атрибутами name и course, потому что любой курс может сдавать несколько студентов и любой студент может сдавать несколько курсов. Очевидно также, что ни name, ни course не зависят от оценки даже в комбинации с другим атрибутом. Поэтому {name, course} $\to$ {grade}

является единственной нетривиальной функциональной зависимостью в этом отношении.

Множество атрибутов, от которого функционально зависят все атрибуты отношения, называется *возможным ключом* отношения.

В любом отношении есть по крайней мере один возможный ключ, потому что любой атрибут тривиально зависит от всех атрибутов отношения. Заметим, что любые два кортежа отношения различаются по значениям атрибутов возможного ключа. Действительно, если значения атрибутов возможного ключа совпадают, то остальные тоже должны совпадать, потому что они функционально зависят от возможного ключа.

Возможный ключ называется *минимальным ключом*, если после исключения из него любого атрибута оставшееся множество атрибутов не является возможным ключом.

В одном отношении может быть несколько минимальных ключей. Один из минимальных ключей выбирается в качестве *первичного ключа* отношения. Понятие первичного ключа не имеет значения для реляционной теории, однако оно важно для применения этой теории. Значения первичного ключа, как и любого возможного ключа, уникальны и могут поэтому использоваться для идентификации объектов реального мира, описываемых кортежами отношения.

Мы по-прежнему не можем говорить об изменяемости кортежей, но можем говорить, что разные кортежи с одинаковыми значениями первичного ключа описывают разные состояния одного реального объекта. Конечно, эти кортежи хранятся в различных состояниях отношения (теоретически — в разных отношениях, описывающих состояние реальности в разные моменты времени), т. к. сосуществовать в одном отношении они не могут.

Например, в отношении exams единственным минимальным возможным ключом является {name, course}. Эта пара атрибутов составляет первичный ключ данного отношения.

Нормализация

Наличие нетривиальных функциональных зависимостей может приводить к нежелательным эффектам, которые принято называть *аномалиями*.

Рассмотрим отношение, приведенное выше на рис. 2.2.7.

Кроме уже упомянутой зависимости {name, course} → {grade}, в нем имеется зависимость {course} → {credits}. Вследствие этой зависимости в отношении присутствует избыточность: зачетные единицы указаны столько раз, сколько раз встречается курс. Более существенный недостаток состоит в том, что в таком отношении невозможно хранить информацию о курсе, который не сдавал ни один студент. Включить информацию о курсе можно только вместе с оценкой и именем студента, а удаление всех оценок по курсу приводит к потере информации о курсе.

Причина состоит в том, что атрибут credits зависит только от части первичного ключа. Для устранения аномалий в этом случае необходимо вместо такого отношения использовать две его проекции, совпадающие с нашими отношениями exams и courses. Это не приводит к потере информации, потому что исходное отношение может быть получено как результат соединения. В нашем случае, конечно, оно и было результатом соединения, но в теоретической реляционной модели хранимые и вычисленные отношения неразличимы.

По историческим причинам отношения, в которых все атрибуты имеют скалярные значения, называются отношениями в *первой нормальной форме* (1NF). В нашем изложении теории все отношения находятся в 1NF.

Отношения, в которых отсутствуют зависимости от неполного ключа, называются отношениями во *второй нормальной форме* (2NF).

Аномалии могут быть вызваны также транзитивными зависимостями. Если имеются функциональные зависимости $X \to Y$ и $Y \to Z$, то существует еще и зависимость $X \to Z$, которая является суперпозицией первых двух. Такие комбинации функциональных зависимостей приводят к аномалиям, потому что каждый комплект значений атрибутов из Z будет повторен вместе с соответствующими значениями атрибутов из Y.

Например, в отношении со схемой {emp, dept, mgr}, для каждого сотрудника указан его отдел и менеджер. В таком отношении имеется избыточность, потому что у всех сотрудников одного отдела менеджер один и тот же.

Для устранения таких аномалий исходное отношение заменяется на его проекции таким образом, что зависимости, являющиеся транзитивными, оказываются в разных отношениях. В нашем примере такими отношениями могут быть {emp, dept} и {dept, mgr}. Эквисоединение этих проекций восстанавливает исходное отношение, поэтому при создании проекций потери информации не происходит. Неформально устранение транзитивных зависимостей переводит отношение в *третью нормальную форму* (3NF).

Существует два неэквивалентных определения третьей нормальной формы. Использованный выше вариант принято обозначать 3NF. Говорят, что отношение находится в *нормальной форме Бойса — Кодда* (BCNF), если для любой нетривиальной функциональной зависимости $X \to Y$ между атрибутами этого отношения множество X содержит некоторый ключ этого отношения. Различие между 3NF и BCNF для нас несущественно.

Построение схемы реляционной базы данных (т. е. набора схем отношений) можно начинать не с определения отношений, а с определения перечня атрибутов и функциональных зависимостей. Известны алгоритмы, обладающие полиномиальной сложностью, которые на основе этой информации строят схему базы данных, состоящую из отношений в третьей нормальной форме (или в BCNF, в зависимости от алгоритма).

Практическое значение нормальных форм и нормализации состоит в том, что они дают критерии, по которым можно оценивать качество логической структуры базы данных. Подчеркнем, что речь идет именно о логической структуре базы данных, на основе которой строятся представления структуры данных для приложений. При этом структуры хранения могут отличаться от логической структуры, поскольку при их проектировании учитываются и другие критерии.

Непосредственное хранение логической схемы может привести к необходимости частого выполнения вычислительно сложных операций соединения во многих запросах. Для того чтобы исключить излишние вычисления, в подобных случаях целесообразно организовать хранение ненормализованных отношений (на уровне схемы хранения, а не на логическом уровне).

В некоторых случаях нормализацией называют проектные решения, никакого отношения к нормализации не имеющие. Типичным примером является замена значений атрибутов (например, строковых) на суррогатные идентификаторы с последующим вынесением строковых значений в отдельное отношение. Такое проектное решение может иметь некоторые основания, однако связь с нормализацией состоит только в том, что оно вводит искусственные транзитивные зависимости. Как указывалось выше, в реляционной модели данных все значения являются константами, они идентифицируют сами себя, и поэтому в рамках этой модели никакой необходимости в дополнительных суррогатных идентификаторах нет. Отметим, что зачастую именно такие проектные решения приводят к излишнему усложнению многих запросов.

Другие зависимости и нормальные формы

В теоретической реляционной модели кроме функциональных зависимостей рассматривается целый ряд других классов зависимостей и связанных с ними нормальных форм.

Так, *многозначные зависимости* определяют соответствие между группами значений атрибутов. Устранение нежелательных многозначных зависимостей приводит к четвертой нормальной форме.

Мы не будем детально рассматривать эти типы зависимостей и соответствующие нормальные формы, потому что их практическое значение невелико.

2.2.6. Практические варианты реляционной модели данных

Большинство используемых в настоящее время систем управления базами данных основано на реляционной модели данных, однако все эти реализации обладают существенными отличиями от этой теоретической модели.

Для того чтобы не отпугнуть потребителей, не имеющих соответствующей подготовки, математические термины заменяются другими: вместо отношений говорят о таблицах, атрибуты называются колонками, а кортежи — строками таблиц. (В некоторых публикациях на русском языке строки таблиц принято называть рядами.)

Далее мы рассмотрим более существенные отличия практических вариантов реляционной модели данных от теоретической.

Неопределенные значения

Одним из важнейших отличий практических реализаций является возможность использования неопределенных значений атрибутов, обычно обозначаемых ключевым словом NULL. Предполагается, что неопределенные значения можно задавать в тех случаях, когда значение атрибута не известно, не определено или не имеет смысла в сочетании со значениями других атрибутов той же строки.

Использование неопределенных значений зачастую упрощает проектирование схемы базы данных за счет некоторого усложнения кода приложений. Так,

при отображении в базу данных иерархии классов все объекты могут быть отображены в одну таблицу; при этом атрибуты, имеющиеся только у объектов подкласса, получают неопределенные значения для объектов суперкласса. В этом случае принадлежность объекта к определенному классу должна устанавливаться в коде приложения, чтобы предотвратить некорректное использование атрибутов, имеющих неопределенные значения.

Довольно часто неопределенные значения используются в тех случаях, когда по каким-либо причинам значение не является обязательным. Например, для пассажира, не имеющего карты постоянного клиента, значение соответствующего атрибута может быть неопределенным.

В теоретической реляционной модели данных неопределенные значения не допускаются, потому что их использование существенно усложняет определение основных операций и приводит к появлению неустранимых парадоксов.

Например, операции над атрибутами, значениями которых являются неопределенные значения, в результате дают неопределенные значения, однако операция фильтрации интерпретирует неопределенное значение логического предиката как ложное. Более детально нелогичности и парадоксы, связанные с использованием NULL, обсуждаются в главе 4.

При проектировании баз данных хорошей практикой считается запрет на использование неопределенных значений для всех атрибутов, кроме тех, для которых неопределенные значения необходимы. Запрет на использование неопределенных значений является ограничением целостности.

Дубликаты

В отличие от теоретической реляционной модели практические реализации допускают появление идентичных строк как в хранимых таблицах, так и в результатах выполнения запросов. Использование дубликатов разрешается стандартом, потому что в ранних системах устранение дубликатов или проверка их отсутствия в хранимых таблицах оказывались вычислительно сложными и могли приводить к существенному снижению производительности системы.

Конечно, для удаления дубликатов аналогичные вычисления требуются и в современных условиях, однако их влияние на общую производительность систем стало менее существенным вследствие увеличения мощности вычислительных систем на несколько порядков, и, с другой стороны, усложнение приложений повысило значение логической корректности.

Наличие дубликатов в хранимых таблицах чаще всего является следствием ошибки разработчиков приложения.

Дополнительные операции

Как возможность использования неопределенных значений, так и допущение дубликатов приводят к изменению семантики и алгебраических свойств теоретических реляционных операций в промышленных реализациях СУБД.

Так, допущение дубликатов приводит к необходимости различать теоретико-множественные операции, исключающие дубликаты из результата и оставляющие их. Соответственно, каждая из операций объединения, пересечения и разности существует в двух вариантах, и появляется дополнительная операция устранения дубликатов. Подчеркнем еще раз, что алгебраические свойства операций, допускающих дубликаты, отличаются от свойств обычных реляционных операций, поэтому нецелесообразно исключать последние из алгебры.

Практические реализации предусматривают также операции внешнего соединения, результаты которых могут содержать неопределенные значения. Например, операция левого внешнего соединения включает в результат те кортежи первого аргумента, для которых не нашлось пары во втором аргументе, дополняя эти кортежи неопределенными значениями для атрибутов второго операнда. Подробнее операции внешнего соединения рассмотрены в главе 4. Здесь мы только обратим внимание на то, что алгебраические свойства этих операций отличаются от свойств обычной операции соединения. Например, левое внешнее соединение некоммутативно.

2.3. Средства концептуального моделирования

Проектирование информационных систем и лежащих в их основе баз данных является довольно сложной задачей. Описание всех используемых в системе или взаимодействующих с ней типов объектов реального мира может насчитывать сотни и тысячи единиц, соотношения и взаимосвязи между ними зачастую оказываются нетривиальными и требуют глубоких знаний процессов, происходящих в реальном мире.

Чтобы упростить взаимопонимание между специалистами предметной области и разработчиками информационных систем, используются разнообразные

языки моделирования, среди которых широкой известностью пользуются унифицированный язык объектного моделирования UML и модель данных концептуального уровня «сущность — связь» (Entity — Relationship, ER). Различие между этими инструментами моделирования в том, что UML позволяет описывать высокоуровневую объектную модель всей системы, в том числе поведение, т. е. описывать функционирование разрабатываемой системы, в то время как ER в основном описывает свойства данных, используемых в системе. Существуют и другие средства моделирования, описывающие отдельные стороны ее функционирования, например языки моделирования бизнес-процессов.

В этом разделе обсуждается главным образом модель данных «сущность — связь», так как ее развитые возможности позволяют описывать самые разнообразные свойства данных, далеко не всегда легко представляемые средствами более общих языков моделирования, в том числе UML. В рамках этого курса будут рассмотрены только основные особенности и возможности модели «сущность — связь»; для более детального освоения модели и процесса проектирования следует обратиться к специальной литературе и документации по системам проектирования баз данных с использованием этой модели.

2.3.1. Модель данных «сущность — связь»

Основные понятия

Одним из основных понятий модели является понятие *сущности*. По определению сущность представляет собой описание некоторого объекта реального мира, который может быть четко отделен от других объектов (возможно, также представленных сущностями в модели), и его описание однозначно связано с этим реальным объектом. Такое, на вид размытое и нечеткое, определение на самом деле задает важное свойство модели: сущности должны быть различимы. Другими словами, с самого начала в модели постулируется наличие некоторого способа идентификации, позволяющего сопоставить объекты реального мира с их представлениями в базе данных. То есть предполагается идентификация сущностей по их естественным признакам (а не по суррогатным идентификаторам).

Кроме этого, требуется, чтобы описания сущностей в модели были связаны с описываемыми объектами реального мира. Как отмечено выше, из этого следует, что сущности могут быть изменяемыми, т. е. могут иметь дополнительные свойства (называемые атрибутами сущности), значения которых могут из-

меняться. Поскольку при этом привязка сущности к объекту реального мира изменяться не должна, сущности в модели должны иметь некоторый неизменяемый идентификатор. На практике в качестве такого идентификатора можно использовать набор из нескольких свойств реального объекта, которые позволяют его определить.

В модели «сущность — связь» предполагается, что все атрибуты сущностей имеют скалярные значения (т. е. эти значения не структурируются в рамках данной модели) и для идентификации атрибутов используются имена.

Совокупность сущностей, имеющих совпадающие (по именам) наборы атрибутов, называется *множеством сущностей*. Понятие множества сущностей является аналогом понятия класса в некоторых объектных моделях. В некоторых источниках множества сущностей называются сущностями, а их элементы — экземплярами сущностей.

Может показаться, что такое формальное (на основе списка атрибутов) определение множеств сущностей может привести к тому, что в одном множестве окажутся сущности, представляющие реальные объекты совершенно различных типов. Однако в рамках понятий, имеющихся в модели, различить типы объектов, имеющих одинаковые наборы атрибутов, невозможно. Если различие между типами объектов существенно для решаемых задач, то оно должно быть отражено различием в списке атрибутов. Проиллюстрируем это упрощенными примерами.

Пусть в системе хранится информация о производителе и модели для автомобилей и самолетов. Это объекты разного типа, и, очевидно, можно определить по значению атрибута «производитель», что Ford — автомобиль, а Airbus — самолет. Однако, если производитель — SAAB, подобный вывод сделать будет сложнее. В действительности такой набор атрибутов описывает не автомобили или самолеты, а любые транспортные средства, поэтому различия между конкретными реальными объектами несущественны. Но если включить, например, атрибут «максимальная высота полета», то различие становится видимым на уровне модели данных.

Рассмотрим другой пример. Пусть в информационной системе необходимо хранить информацию о поставщиках и потребителях продукции предприятия. По-видимому, поставщики и потребители будут характеризоваться одним и тем же набором атрибутов, описывающим (юридические или физические) лица. Немного более тщательное проектирование показывает, что одно и то же лицо может быть как поставщиком, так и потребителем, поэтому объединение всех партнеров в одном множестве сущностей вполне оправдано.

Другим ключевым понятием модели является понятие *связи*. По определению связь представляет собой упорядоченную последовательность сущностей, имеет свою идентификацию и может иметь свои собственные атрибуты. Связи между сущностями из одних и тех же множеств, имеющие идентификацию одного типа и совпадающие (по именам) наборы атрибутов, составляют множество связей.

Например, сущности студент и дисциплина могут быть связаны связью экзамен. При этом упорядоченным набором сущностей будет (студент, дисциплина). Эта же пара будет идентифицировать связь, а дополнительными атрибутами связи могут быть оценка и дата ее получения.

Структуры данных, описываемые в рамках модели ER, принято представлять двумерными диаграммами. В классической модели ER сущности изображаются прямоугольниками, связи — ромбами, атрибуты — овалами. Такие диаграммы, однако, получаются слишком громоздкими, поэтому описания атрибутов обычно включаются в тот элемент диаграммы, к которому они относятся, или вовсе не включаются, в особенности на начальных фазах проектирования.

Простейший пример диаграммы «сущность — связь» с использованием ромба для обозначения связи показан на рис. 2.3.1.

Рис. 2.3.1. Простейшая диаграмма «сущность — связь»

Ограничения целостности

Наиболее распространенным видом связей являются бинарные, т. е. такие, которые связывают две сущности. Для них можно определить ограничения, показывающие, как именно сущности могут быть взаимосвязаны. Используются следующие символы:

1 — в связи может участвовать и должна участвовать одна сущность;

0 — в связи может участвовать не больше, чем одна сущность;

m, n — в связи может участвовать ноль или несколько сущностей из одного множества.

Ограничения целостности описываются парой таких символов, разделенных двоеточием.

Например, если каждый сотрудник некоторого предприятия обязательно является сотрудником одного отдела, такое ограничение будет выражено записью 1 : n (читается «один ко многим»). При этом n обозначает, что несколько сотрудников могут быть связаны с одним отделом, а 1 — что каждый сотрудник обязательно связан с каким-нибудь отделом и только с одним.

Точно так же связь между аэропортом отправления и рейсом подчиняется ограничению 1 : n, потому что каждый рейс обязательно отправляется из какого-нибудь одного аэропорта. С другой стороны, связь между пассажиром и постоянным клиентом подчиняется ограничению 0 : n, потому что постоянный клиент может быть пассажиром нескольких рейсов (все-таки он — постоянный клиент и, скорее всего, пользуется услугами авиаперевозчика неоднократно), но пассажир не обязательно должен быть связан с постоянным клиентом.

Наиболее сложным типом бинарных связей являются связи типа m : n (читается «многие ко многим»). Например, сотрудник может участвовать в нескольких проектах, и в каждом проекте может участвовать несколько сотрудников.

Наследование

Объекты реального мира могут одновременно входить в состав нескольких множеств. Так, любой человек может быть (а может не быть) работником какого-либо предприятия. Подобные ситуации представляются в модели «сущность — связь» с помощью понятия наследования, которое представляется связью *is a*. Эта связь всегда является бинарной связью 1 : 0, и дополнительно требуется, чтобы идентификаторы связанных сущностей совпадали. Например, если каким-либо способом в базе данных найден студент, то во множестве персон обязательно должна быть сущность с таким же идентификатором и обязательно связанная с найденной сущностью во множестве студентов. Неформально можно это выразить утверждением: *студент является человеком*. Заметим, что во многих объектных моделях данных такое требование отсутствует.

Модель «сущность — связь» предписывает также, что при наследовании атрибуты более широкого множества не дублируются в сущностях более узкого. Например, если сущность персона имеет атрибут дата рождения, то такого атрибута не должно быть у сущности студент, так как его можно получить из сущности более широкого множества.

Это правило, на первый взгляд, определяет требования к организации хранения, что несколько странно для модели концептуального уровня. Однако смысл данного ограничения состоит в том, что значения атрибутов могут наследоваться, но не могут изменяться при наследовании. Например, рост и вес человека будут одинаковыми, независимо от того, рассматриваем мы этого человека как студента или нет.

Условие на тождественность идентификаторов сущностей накладывает ограничение на множественное наследование: оно возможно только в том случае, если имеется общее множество, из которого наследуют оба наследуемых множества (правило «ромба»). Например, множество работающих студентов может наследовать свойства студентов и свойства сотрудников, потому что и студенты, и сотрудники являются специализациями более общей сущности «человек» и, следовательно, все четыре множества могут иметь общий идентификатор.

Далее в главе 8 мы увидим, что реализация наследования в системе PostgreSQL отличается от наследования в модели данных «сущность — связь».

Отображение в реляционную модель

В модели «сущность — связь» не предусмотрены операции манипулирования данными. Для того чтобы описание данных можно было использовать, необходимо на его основе построить описание данных в другой модели.

Отображение в реляционную модель данных строится следующим образом:

1. Для каждого множества сущностей строится отношение, атрибутами которого становятся идентификатор и все атрибуты, имеющиеся у сущностей, входящих в это множество.
2. Определяются функциональные зависимости атрибутов от идентификатора для каждого построенного отношения.
3. Для каждого множества связей строится отношение, атрибутами которого становятся идентификатор и все атрибуты связи. Напомним, что идентификатор связи содержит все идентификаторы связываемых сущностей, поэтому в отношении идентификатору связи будет соответствовать несколько атрибутов.
4. Определяются функциональные зависимости атрибутов, полученных из атрибутов связи, от атрибутов, полученных из идентификатора связи.

Если другие функциональные зависимости не определяются, то полученные отношения будут в 3NF. Если же дополнительные функциональные зависимости удалось обнаружить, то полученная схема не будет нормализованной и, возможно, понадобится дополнительная нормализация. Заметим, однако, что наличие таких функциональных зависимостей может указывать на ошибки в проектировании модели «сущность — связь» (возможно, некоторые сущности на самом деле являются агрегатами, состоящими из более мелких сущностей).

Рассмотрим пример.

При перевозке грузов с каждым заказом на перевозку связаны отправитель, получатель, плательщик и, возможно, еще несколько юридических или физических лиц. Поскольку каждое из этих лиц может быть как отправителем, так и получателем (для разных грузов), все сущности, которые могут выступать в одном из этих качеств, объединяются в одно множество сущностей, которое будет называться party. Конечно, эти сущности имеют много разнообразных атрибутов, но пока мы ограничимся только идентификатором и именем.

Из атрибутов заказа (order) пока возьмем только вес. И наконец, атрибутом связи между заказом и партией будет, кроме идентификаторов связываемых сущностей, роль партии в этой связи (отправитель, получатель и т. п.).

Диаграмма для этого примера показана на рис. 2.3.2.

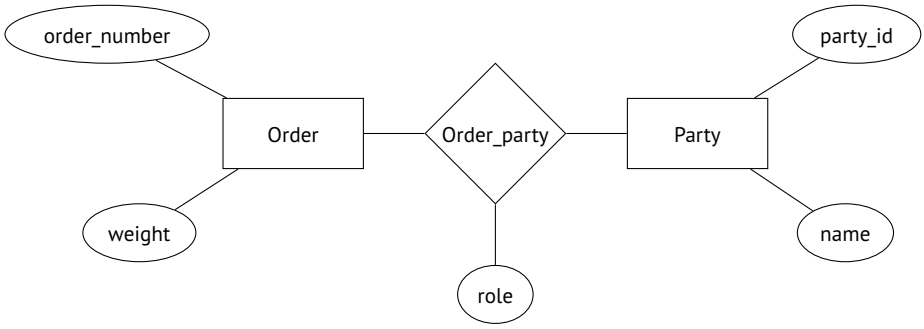


Рис. 2.3.2. Связь груза с партиями

В результате отображения будет получена схема, показанная на рис. 2.3.3.

При этом ключами в отношениях order и party будут order_number и party_id соответственно, а ключом отношения order_party, представляющего связь между партиями и заказами, будет совокупность всех трех его атрибутов, поскольку

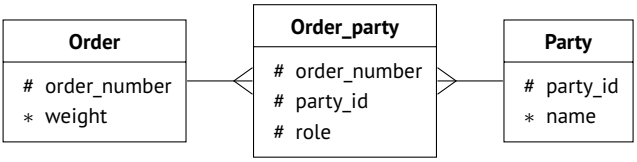


Рис. 2.3.3. Представление связи m : n

одна и та же партия может выступать по отношению к грузу в нескольких ролях одновременно: например, отправитель может быть плательщиком.

Множество связей, на которые наложено ограничение 1 : n, можно отобразить более простым способом. Дело в том, что ограничение такого вида определяет функциональную зависимость идентификатора одной сущности от идентификатора другой. Действительно, пусть множества сущностей R_1 и R_n с ключами k_1 и k_n связаны связью E, которая подчиняется ограничению 1 : n. Ключом для связи будет пара ключей связываемых отношений, однако ключ k_n однозначно определяет ключ k_1 в силу ограничения целостности.

Поэтому атрибуты связи могут зависеть только от идентификатора второй сущности k_n и могут быть включены в отношение, представляющее при отображении множество сущностей R_n вместе с идентификатором из первого множества R_1, и для представления связи отдельная таблица не нужна.

В практически используемых инструментах для построения диаграмм модели данных «сущность — связь» не предусмотрена возможность определения связей m : n, поэтому для представления таких связей необходимо определять дополнительные сущности и две связи типа 1 : n.

Ключ другого отношения, функционально зависящий от ключа рассматриваемого отношения и включенный в состав его атрибутов, называется *внешним ключом* (foreign key).

2.3.2. Концептуальные объектные модели

Современные варианты модели «сущность — связь» содержат большое количество разнообразных возможностей для описания структур данных и особенностей их взаимосвязей, однако не позволяют описывать какие-либо операции над этими структурами данных. По-видимому, эта особенность модели связана с тем, что в период ее создания предполагалось, что операции обработки

данных, выполняемые в приложении, должны быть отделены от схемы базы данных и поэтому их следует проектировать отдельно другими средствами.

Зачастую, однако, эта особенность модели рассматривается как недостаток, который преодолевается в рамках объектно-ориентированного подхода к проектированию систем. В настоящее время доминирующим средством объектно-ориентированного проектирования является унифицированный язык моделирования (Unified Modeling Language, UML).

В процессе моделирования с использованием UML одновременно создаются диаграмма классов приложения и соответствующая ей модель базы данных. Хотя формально никаких ограничений на структуры данных этот язык сам по себе не накладывает, но на практике зачастую получаются схемы баз данных, несколько уступающие по качеству схемам, получаемым при использовании модели «сущность — связь». Это вызвано в первую очередь необходимостью компромиссов для упрощения отображения схемы базы данных в структуры данных приложения (и обратно). Некоторые дополнительные ограничения на структуры базы данных возникают при использовании систем, реализующих объектно-реляционные отображения (ORM), или специализированных систем генерации приложений (frameworks).

2.4. Объектные и объектно-реляционные модели данных

В связи с расширением спектра областей применения систем управления базами данных предположения, в которых разрабатывались ранние реляционные СУБД, оказались слишком ограничительными. Поэтому, начиная с середины 80-х гг. исследуются различные расширения модели, позволяющие более непосредственно отобразить специализированные структуры хранения на логическом уровне. В рамках этого направления рассматривались, например, отношения не в первой нормальной форме (Non-First-Normal-Form, NFNF, NF^2, nested relations) и другие варианты структурирования данных, допускающие более сложные типы атрибутов, чем скаляры.

В дальнейшем это направление привело к созданию языков программирования с постоянным хранением данных (persistent programming languages) и объектных моделей баз данных. И в том и в другом случае в базе данных размещаются объекты, определяемые в рамках объектной модели языка программирования. Этот подход позволил добиться очень высокой эффективности доступа

в режиме навигации: в лучших системах время доступа к постоянно хранимым объектам только в 3–4 раза превышало время доступа к оперативной памяти.

Ожидалось, что объектные базы данных вытеснят все остальные классы моделей данных, однако по ряду причин этого не произошло:

- база данных может использоваться только клиентами, написанными на одном языке программирования;
- не удается создать высокоуровневый декларативный язык запросов;
- не удается обеспечить высокую производительность при массовой выборке данных.

В итоге системы, основанные на чисто объектных моделях данных, заняли относительно небольшой сегмент в многообразии применений СУБД, но некоторые объектные средства стали составной частью систем управления базами данных общего назначения. В настоящее время все высокопроизводительные системы, в том числе PostgreSQL, реализуют объектные расширения, и поэтому их принято называть объектно-реляционными.

Наиболее важными видами объектных расширений можно считать:

- возможность определения пользовательских типов данных, в том числе структурных;
- использование коллекций объектов.

Средства определения типов данных, по сути, заложены в концепции абстрактного домена, поэтому возможность определения пользовательских типов данных скорее снимает ограничения ранних реализаций, чем расширяет теоретическую реляционную модель данных, по крайней мере, если ограничиваться скалярными типами. В дополнение к скалярным пользовательским типам обычно объектные расширения включают возможности создания структурных типов. Примерами структурных типов могут быть геометрические объекты (точки, прямые и т. д.).

Коллекцией называется набор объектов определенного типа. Различают следующие разновидности коллекций:

set — набор объектов, не содержащий дубликатов, т. е. аналог реляционного отношения;

bag — неупорядоченный набор объектов, в котором могут быть дубликаты;

list — упорядоченный список объектов;

array — набор объектов с доступом по индексу или индексам.

Любая коллекция может быть значением атрибута. Существуют функции, преобразующие коллекции в виртуальные таблицы, а также позволяющие записывать результаты выполнения запросов в качестве значений коллекций.

Возможности определения и использования коллекций в системе PostgreSQL обсуждаются далее в главе 8.

2.5. Другие модели данных

2.5.1. Слабоструктурированные модели данных

В некоторых классах приложений отделение описания структуры данных (т. е. схемы) от самих данных оказывается нежелательным. Например, при передаче документов по сети целесообразно определение структуры документа пересылать вместе с документом. Поскольку в подобных случаях значительная часть данных представляет собой текст на естественном языке, такие данные принято называть *слабоструктурированными* (semi-structured; некоторые авторы упрямо называют такие данные полуструктурированными). Наиболее широко используемым форматом для представления слабоструктурированных данных является XML, довольно часто употребляется также JSON, популярность которого стремительно растет в последние годы.

Хотя ни XML, ни JSON не предполагалось использовать для хранения данных, оба этих формата можно рассматривать как модели данных. Широко распространено мнение о том, что эти модели предоставляют возможности для более гибкого описания и представления данных, чем реляционная модель, однако на самом деле схемы XML (XSD) предоставляют средства для описания не менее жестких ограничений целостности, чем реляционная модель данных.

В объектно-реляционных базах слабоструктурированные данные могут храниться как значения атрибутов отношений (таблиц). В системе PostgreSQL для этого используются типы данных xml, json и jsonb, более детально обсуждаемые в главе 8. Как и для типов коллекций, имеются встроенные функции, позволяющие формировать значения слабоструктурированных типов из табличных данных и, наоборот, извлекать элементы слабоструктурированных значений

в виде коллекций. Это дает возможность сочетать реляционные и слабоструктурированные языки запросов (в частности, XPath и XQuery, а также SQL/JSON path, введенный в стандарте SQL 2016).

2.5.2. Модели для представления знаний

В системах представления знаний (таких как семантические сети и онтологии), а также для представления графов часто используется тернарная модель данных, в которой элементарной структурой является тройка вида *(объект, атрибут, значение)*. При этом для записи каждого объекта необходимо столько строк, сколько атрибутов имеет этот объект.

В таком представлении схема (перечень атрибутов) неотделима от данных, что обеспечивает возможность хранения любых объектов без какого-либо предварительного описания их структуры. Это дает очень большую гибкость представления, которая привлекает многих разработчиков.

С другой стороны, применение тернарного представления существенно усложняет функции приложения и очень существенно влияет на его производительность.

2.5.3. Ключ — значение

В последнее десятилетие получили довольно широкую известность системы, предназначенные для хранения пар ключ — значение. При этом предполагается, что поиск данных возможен только по (первичному) ключу, а интерпретация значения выполняется в приложении. Привлекательной стороной таких систем является простота начального запуска, однако практически все функции, обычно выполняемые системами управления базами данных, в том числе

- сложные структуры данных,
- взаимосвязи между объектами данных,
- ограничения целостности,
- высокоуровневые декларативные запросы,
- поиск по значениям неключевых атрибутов

и другие, должны реализовываться в коде приложения, что существенно усложняет разработку приложений или приводит к снижению качества.

Многие системы, первоначально позиционировавшиеся как системы этого типа, эволюционируют в направлении включения более сложных возможностей, приближающих их к более развитым СУБД, вплоть до реализации полноценных декларативных языков запросов.

2.5.4. Устаревшие модели данных

Во многих учебниках по базам данных до сих пор рассматриваются ранние модели данных, использовавшиеся в 70–80-е гг. Наиболее известной из этих моделей является сетевая, язык описания данных для которой разрабатывался комитетом CODASYL [21], а наиболее широко применявшейся была иерархическая модель данных, реализованная в системе IMS компании IBM. Обе эти модели предоставляют возможности навигационного доступа к отдельным объектам и возможности описания взаимосвязей между ними, однако в иерархической модели представление для клиентской программы всегда является деревом (хотя в самой базе данных возможно описание и хранение любых видов взаимосвязей).

Возможности сетевой модели данных полностью перекрываются объектными и объектно-реляционными моделями данных, а возможности иерархической — средствами XML и JSON.

2.6. Примеры проектирования схемы в модели «сущность — связь»

Варианты модели данных «сущность — связь», используемые на практике и реализованные во многих инструментах проектирования баз данных, значительно отличаются от представления модели, излагаемого в учебниках (в том числе от варианта, кратко представленного выше).

Наиболее важное отличие состоит в том, что исключаются любые связи между более чем двумя сущностями, а также связи типа «многие ко многим». Поскольку для бинарных связей типа 1 : n или 0 : n атрибуты связи можно перенести в одну из связываемых сущностей (выше показано, как это делать, при

описании отображения в реляционную модель) и поскольку нет необходимости в явной идентификации связей, обозначения связей ромбами становятся ненужными. Связи обозначаются линиями, соединяющими сущности (прямоугольники).

Для описания в базе данных ситуаций, в которых требуются связи «многие ко многим», необходимо вводить дополнительные сущности, выполняющие роль таких связей.

На рис. 2.3.3 показан фрагмент подобной диаграммы, на котором множественная связь заменена на сущность. Аналогично множественная связь экзамен между множествами курсов и студентов может быть заменена на сущность.

На концах линии, представляющей бинарную связь, указывается кратность связи. Существуют различные системы нотации для кратностей, при этом многие инструментальные средства предоставляют возможность выбора нотации.

Развитые реализации модели «сущность — связь» предоставляют целый ряд различных дополнительных понятий, позволяющих более детально и более точно описывать взаимосвязи между сущностями.

Одним из таких понятий является понятие *слабой сущности*. В отличие от обычных (сильных) сущностей слабая сущность может существовать в базе данных, только если она связана с некоторой *сильной сущностью*. Например, в базе данных предприятия может содержаться информация о детях сотрудников, необходимая для расчета каких-либо льгот или выплат, предусмотренных законодательством, или, может быть, просто для рассылки новогодних подарков. Однако эти сущности (дети сотрудников) не имеют смысла для предприятия, если сотрудники увольняются. Возможно, конечно, что такая информация должна сохраняться в качестве архивной вместе с информацией о лицах, ранее работавших на предприятии, однако это совсем другой аспект проектирования базы данных.

Рассмотрим фрагмент схемы базы данных, показанный на рис. 2.6.1 и предназначенный для хранения информации, имеющейся на авиабилетах. Конечно, этот фрагмент значительно проще, чем реальные схемы, пригодные для данной цели. Многие важные стороны процесса перевозки пассажиров в этом варианте учесть невозможно, однако даже такая упрощенная схема дает возможность показать использование различных типов связей.

Основной сущностью в этой схеме является *бронирование* (bookings). В момент приема заказа создается уникальный номер бронирования book_ref, который никогда не изменяется и поэтому может служить в качестве идентификатора.

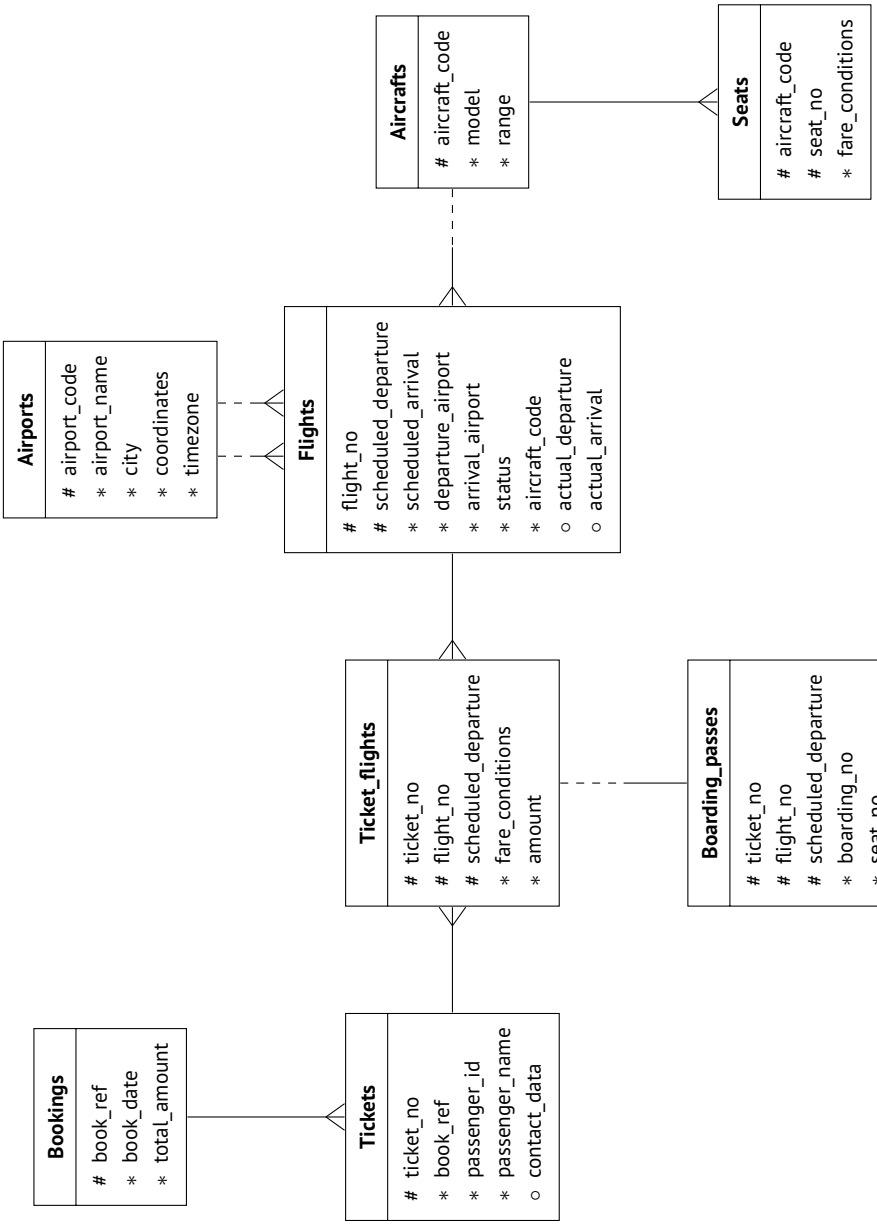


Рис. 2.6.1. Фрагмент схемы базы данных для хранения авиабилетов

В каждое бронирование можно включить несколько пассажиров, для каждого из которых создается отдельный *билет* (tickets). Пассажир не выделен в отдельную сущность; его идентификационная информация (номер документа и имя) записывается в билет. Наша схема не дает возможности узнать, являются ли разные пассажиры одним физическим лицом.

Билет может содержать несколько *перелетов* (ticket_flights), соответствующих прямому и обратному рейсам, а кроме того, перемещение в каждом направлении может включать пересадки. Каждый перелет связан с бронированием и с *рейсом* (flights), поэтому в качестве идентификатора перелета используется комбинация из идентификатора бронирования и идентификатора рейса.

Рейс идентифицируется своим номером и датой вылета по расписанию. Он связан с двумя *аэропортами* (airports) отправления и прибытия. Каждый аэропорт имеет уникальный трехбуквенный код (например, SVO или LED).

При регистрации на рейс каждому пассажиру выдается *посадочный талон* (boarding_passes), содержащий номер места в качестве атрибута (в дополнение к необходимой идентификации).

Количество *мест* (seats) в кабине зависит от модели *самолета* (aircrafts), совершающего рейс. Предполагается, что каждая модель имеет только одну компоновку салона.

Заметим, что в этом примере использованы только естественные ключи, т. е. все значения, использованные для идентификации, применяются в реальных бизнес-процессах и никак не зависят от информационной системы. Такое проектное решение обеспечивается тем, что в реальности фактически применяются неизменяемые идентификаторы, обеспечивающие уникальность, поэтому введение дополнительных суррогатных ключей привело бы к некоторой избыточности данных. С другой стороны, это приводит к появлению составных первичных ключей: в нашем примере ключи перелета и посадочного талона содержат по три атрибута.

Отметим также, что на самом деле наши таблицы не нормализованы. Так, в таблице рейсов аэропорты отправления и прибытия зависят только от номера рейса, но не зависят от даты и времени вылета, т. е. в этой таблице имеется зависимость от неполного ключа, и поэтому она не находится во второй нормальной форме. Конечно, этот недостаток модели легко устранить, выделив еще одну сущность — *маршрут*, который идентифицируется номером рейса. Однако особенности использования этой таблицы и ее небольшие размеры делают влияние аномалий ненормализованных схем не очень существенным.

Поскольку таблица невелика по размеру, избыточность тоже невелика, коды аэропортов практически никогда не изменяются, поэтому аномалии обновления также не могут повлиять на работу системы.

На рис. 2.6.2 показан вариант схемы базы данных авиабилетов, в котором во всех таблицах использованы суррогатные ключи.

Можно заметить, что эта схема обладает некоторыми недостатками, по сравнению со схемой, показанной на рис. 2.6.1. Так, при извлечении данных из таблицы рейсов для получения кодов аэропортов, которые понятны любому специалисту без какой-либо дополнительной расшифровки, необходим доступ к таблице аэропортов. Зависимость от неполного ключа превратилась в транзитивную зависимость (идентификаторы аэропортов отправления и прибытия зависят от номера рейса, однако для нормализации потребуется ввести еще один суррогат для маршрута). Поскольку коды аэропортов и номера бронирования являются в действительности уникальными и неизменяемыми, использование суррогатов в этих таблицах создает транзитивные зависимости. Устранение этих зависимостей привело бы к появлению сущностей, которым не соответствуют никакие объекты реального мира.

Необходимо, однако, заметить, что в подобных случаях транзитивные зависимости не могут вызвать отрицательные эффекты, связанные с аномалиями (кроме избыточности), потому что такие атрибуты, как номер бронирования, обладают уникальностью и неизменяемостью.

При проектировании схемы базы данных необходимо учитывать ограничения, накладываемые на структуру базы данных средой разработки приложений. При этом неизбежны компромиссы, которые приводят к ухудшению качества схемы базы данных. Одним из часто встречающихся компромиссов такого рода является использование искусственных (суррогатных) ключей для всех таблиц базы данных. Это может быть необходимо, если среда разработки приложения не допускает использование никаких идентификаторов объектов, кроме целочисленных. Повсеместное применение суррогатов может приводить к появлению фактических дубликатов (различающихся только искусственными идентификаторами) и, как следствие, к трудно обнаруживаемым ошибкам в работе приложения.

Сторонники тотального использования суррогатных ключей указывают на следующие преимущества этого подхода к проектированию схемы:

- гарантируется уникальность идентификаторов объектов, так как они генерируются в системе;

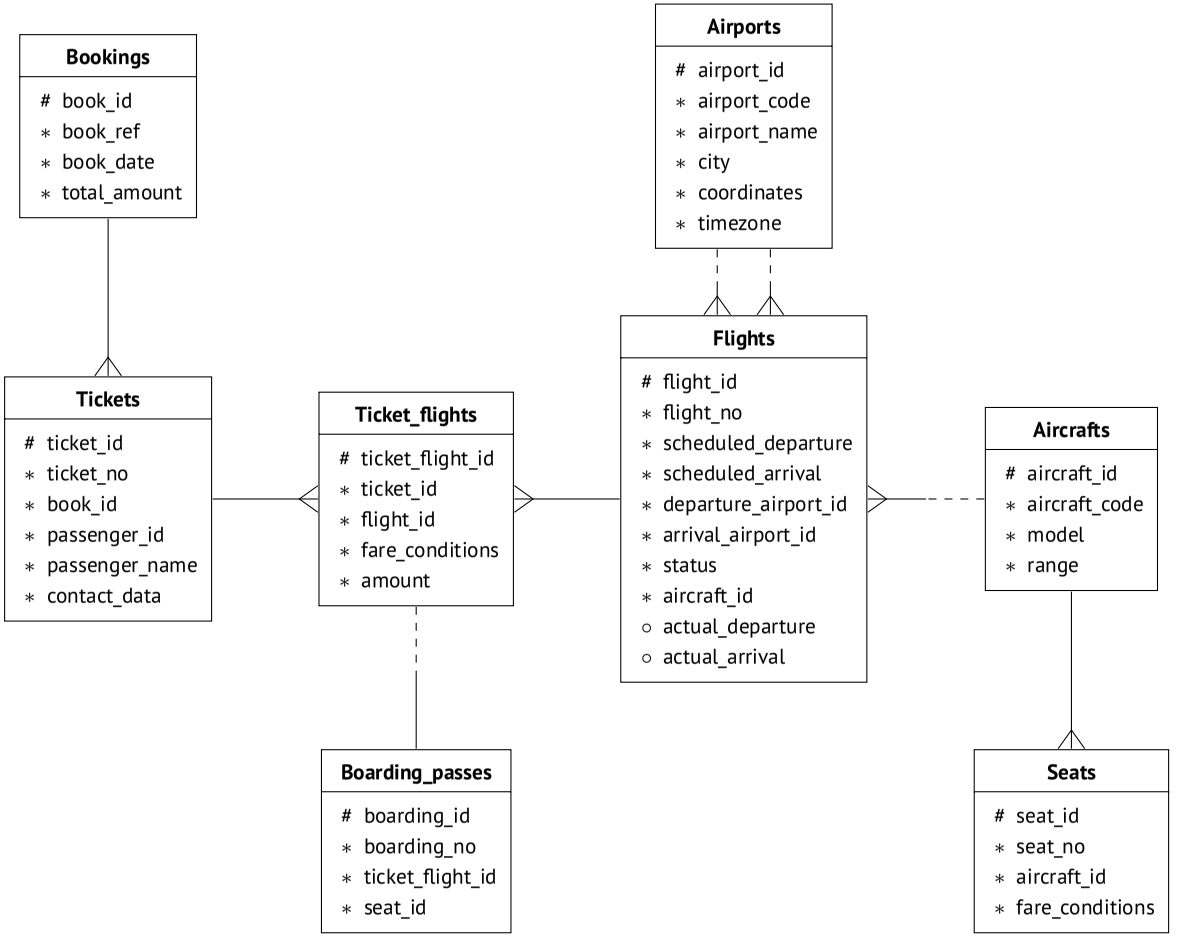


Рис. 2.6.2. Фрагмент схемы базы данных для хранения авиабилетов с суррогатными ключами

- гарантируется неизменяемость идентификаторов, нет необходимости в тщательном анализе предметной области для выявления неизменяемых естественных идентификаторов;
- исключается необходимость использования составных ключей.

На рис. 2.6.3 показана схема демонстрационной базы данных, которая будет использоваться в большинстве примеров и упражнений в этом курсе. При проектировании этой схемы были приняты компромиссные решения: в большинстве отношений использованы естественные ключи, но для некоторых оказалось целесообразно определить суррогатные.

Кроме хранимых отношений, эта схема содержит представление для маршрутов, которое можно использовать в запросах наравне с хранимыми отношениями и которое содержит данные, вычисляемые на основе данных из хранимых отношений.

2.7. Библиографические комментарии

Понятия, связанные с навигационной обработкой данных, и основы сетевой модели данных собраны в статье [7], автор которой был позже удостоен премии Тьюринга (русский перевод в [71]). Опыт использования иерархической системы управления базами данных IMS/360 обобщен в [20].

Основы реляционной модели данных представлены широкой аудитории в [18], наиболее часто цитируемой статье в компьютерной литературе. Несколько более сложный вариант этой модели данных представлен в [19].

Различные подходы к нормализации реляционных схем обсуждаются и сопоставляются в [26]; работа [9] описывает алгоритмы синтеза реляционных схем по функциональным зависимостям. Обстоятельная книга [44] (русский перевод [66]) содержит изложение теории реляционных баз данных. Все статьи и книга, перечисленные в этом разделе, требуют достаточно серьезной математической подготовки.

Модель данных «сущность — связь» впервые представлена в [17]. За этой статьей последовало огромное количество исследований и практических разработок методов и инструментов проектирования схем реляционных баз данных.

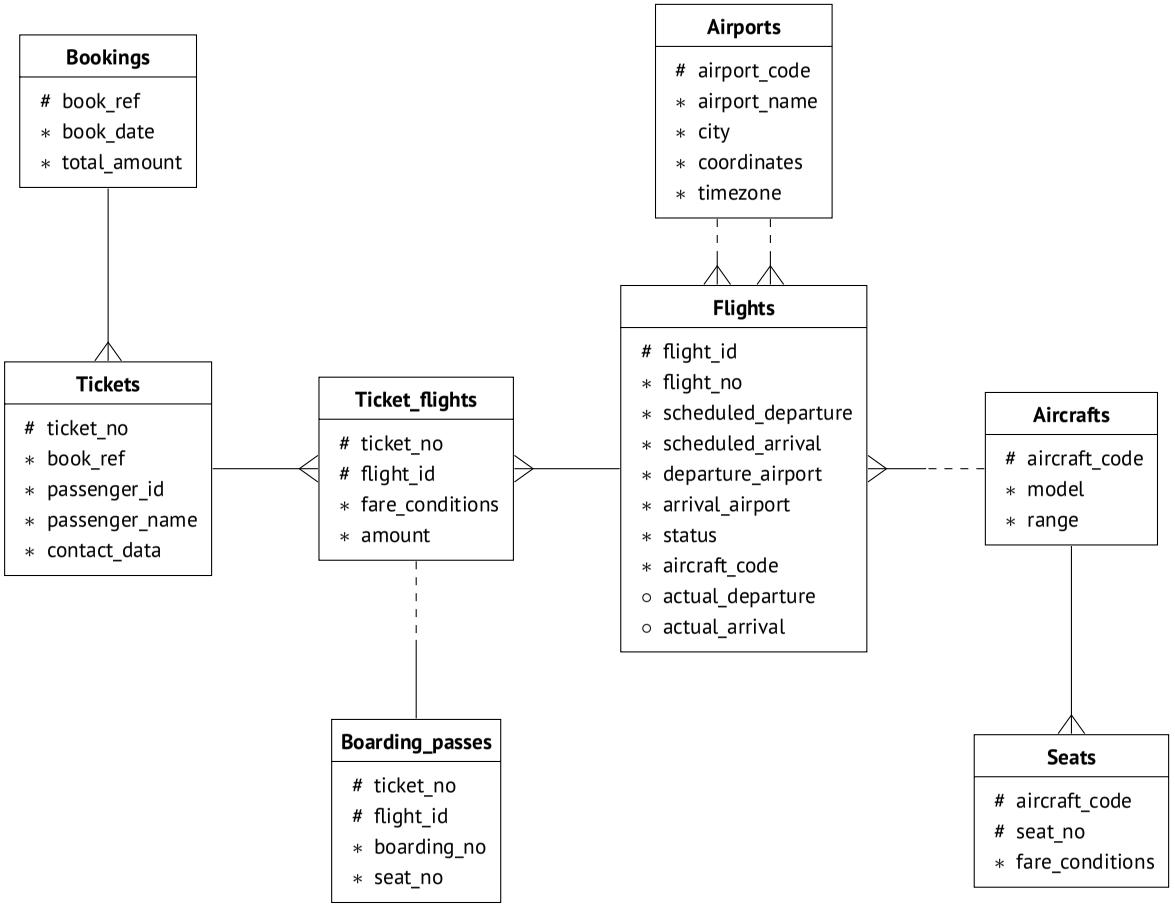


Рис. 2.6.3. Схема демонстрационной базы данных

2.8. Упражнения

Упражнение 2.1. Докажите, что операции UNION, INTERSECT, PROD, JOIN коммутативны.

Упражнение 2.2. Докажите, что операции UNION, INTERSECT, PROD, JOIN ассоциативны.

Упражнение 2.3. Докажите, что выполняются дистрибутивные законы для PROD и JOIN относительно UNION и INTERSECT, а также для UNION относительно INTERSECT, INTERSECT относительно UNION.

Упражнение 2.4. В схеме с курсами и студентами предусмотрите возможность ведения занятий по курсу несколькими преподавателями. Экзамен может сдаваться не тому преподавателю, который вел занятия.

Упражнение 2.5. Укажите отклонения от третьей нормальной формы в демонстрационной базе данных.

Упражнение 2.6. В схемах с авиаперевозками аэропорт может иметь несколько терминалов, названия терминалов уникальны внутри аэропорта. Каждый рейс отправляется с определенного терминала аэропорта отправления и прибывает на определенный терминал аэропорта прибытия. Внесите дополнения в схему, позволяющие хранить эту информацию. Во многих аэропортах имеется только один терминал, это не должно быть особым случаем при выполнении запросов.

Упражнение 2.7. Авиаперевозчики могут регистрировать постоянных клиентов. Если при бронировании указывается карточка постоянного клиента, информация о нем может быть скопирована в бронирование, но может быть и изменена. Добавьте в схему демонстрационной базы возможность хранения данных о постоянных клиентах.

Упражнение 2.8. Багаж пассажира может опаздывать на рейс при пересадке и доставляться отдельно от пассажира другим рейсом. Добавьте отношения для представления информации о багаже.

Упражнение 2.9. Аэропорт может быть основным аэропортом для нескольких близлежащих населенных пунктов. Внесите изменения в схему, позволяющие описать такие ситуации.

Упражнение 2.10. Создайте схему базы, которую можно использовать для хранения исторических данных о погоде (температура, влажность, скорость ветра, время суток и т. п.).

Упражнение 2.11. Создайте схему базы с информацией о наличии автомобилей в автомобильных салонах. Необходимо учитывать марку автомобиля, модель, год выпуска, адрес салона и т. п.

Глава 3

Знакомимся с базой данных

3.1. Установка базы данных

Для того чтобы продолжить освоение материала, представленного в этой книге, необходимо установить систему управления базами данных PostgreSQL (или получить доступ к уже установленной). Система PostgreSQL может работать под управлением любой распространенной операционной системы практически на любых компьютерах.

Действия, необходимые для установки, зависят от операционной системы и детально описаны на сайте PostgreSQL https://www.postgresql.org/download.

Также можно воспользоваться специально подготовленным образом операционной системы Linux с предустановленной СУБД PostgreSQL. Образ доступен по адресу https://edu.postgrespro.ru/DBTECH-student.ova и может быть импортирован в систему виртуализации, такую как VirtualBox или VMWare.

Почти все примеры и упражнения в первой части этой книги используют *демонстрационную базу данных* системы PostgreSQL. В подготовленный образ эта база уже включена, а в случае самостоятельной установки нужно дополнительно скачать один из доступных вариантов демобазы и загрузить его в свою систему. Как это сделать, детально описано на странице сайта компании Postgres Professional https://postgrespro.ru/education/demodb. Там же вы найдете и полное описание схемы этой базы данных.

3.2. Подключение к серверу базы данных

Напомним, что базой данных (БД) называется совокупность данных, доступная для приложения через сервер базы данных. В этой главе мы будем предполагать, что приложение выступает в роли клиента и взаимодействует непосредственно с сервером базы данных. В более сложных (многослойных) архитектурах эту роль выполняют компоненты, непосредственно взаимодействующие

с СУБД (например, часть системы, работающая на сервере приложений), но в этой главе рассматривается только взаимодействие с сервером базы данных непосредственно.

Для того чтобы выполнять какие-либо действия над данными, находящимися в БД, приложение должно установить соединение с сервером базы данных. Различные клиенты (приложения) могут устанавливать соединение по-разному, однако обычно для этого требуется следующая информация:

- сетевой адрес сервера базы данных (может быть указан именем или IP-адресом);
- номер порта, через который производится соединение;
- имя базы данных;
- имя пользователя базы данных.

Значения этих параметров могут быть записаны в конфигурационных файлах клиента и поэтому не обязательно указываются явно, однако они необходимы в любом случае.

Сетевой адрес сервера и порт определяют, куда передаются сообщения, адресованные серверу базы данных. Большинство СУБД имеет зафиксированный предпочтительный номер порта, который используется, если администратор не изменит его при создании сервера. Для СУБД PostgreSQL это порт 5432. Использование другого номера порта необходимо, если на одном компьютере одновременно работает несколько серверов баз данных. Конечно, сервер и клиент базы данных тоже могут работать на одном и том же компьютере, однако для способа установки соединения это не имеет значения, потому что и в этом случае используется сетевой протокол (исключение составляют локальные подключения через сокеты домена Unix, поддерживаемые некоторыми операционными системами).

Один сервер баз данных может обрабатывать запросы к нескольким базам данных. Совокупность баз данных, обслуживаемых одним сервером, в системе PostgreSQL называется *кластером*. (Термин «кластер» используется также во многих других контекстах и может обозначать совокупность компьютеров, совокупность близких точек в методах анализа данных, способ размещения коллекций данных на носителях и имеет ряд других значений, непосредственно с компьютерами не связанных.)

Некоторые объекты в системе PostgreSQL, например пользователи, определены для всего кластера, однако в рамках одного соединения клиентская программа может работать только с одной базой данных. Имя этой базы указывается при установлении соединения. В простых конфигурациях (например, для серверов баз данных, запускаемых на компьютере клиента) часто достаточно одной базы данных. Как мы увидим далее, ничто не мешает хранить данные нескольких независимых приложений в одной базе. Это упрощает совместное использование данных разных приложений в рамках одного сеанса. В то же время практика создания отдельных баз данных для каждого небольшого приложения широко распространена.

Понятие *пользователь* (user) имеет много различных значений. Нам будет нужно различать пользователя операционных систем на компьютерах сервера и клиента, пользователя базы данных и пользователя приложения. Эти понятия часто существенно отличаются. Например, если приложение работает как веб-приложение (такое как интернет-магазин), то его пользователем становится любой покупатель, однако такой пользователь не будет зарегистрирован как пользователь базы данных.

В некоторых случаях может быть удобно использовать одинаковые имена для обозначения разных объектов, например пользователи операционной системы или домена могут быть зарегистрированы как пользователи базы данных. Но в любом случае эти сущности остаются различными.

Понятие пользователя базы данных важно для реализации разграничения доступа и защиты данных на уровне БД. Права доступа как к базе данных в целом, так и к отдельным объектам определяются именно для пользователей БД. Управление доступом к объектам базы данных более детально обсуждается в главе 5.

3.3. Простой клиент: psql

Для того чтобы работать с базой данных, необходима программа, выполняющая функции клиента. В примерах этой книги в качестве такой программы будет использоваться `psql`. Эта программа входит в состав любого комплекта PostgreSQL в любой операционной системе. Для запуска программы `psql` нужно с командной строки операционной системы ввести команду

```
psql -d имя_бд -U имя_пользователя -h сервер -p порт
```

Возможно, в ответ на подсказку системы потребуется ввести пароль. В некоторых комплектах эту программу можно запускать и другим способом.

Часто для соединения с локальным сервером базы данных (т. е. работающим на том же компьютере, что и psql) не требуется указывать вообще никаких параметров. Например, при использовании образа ОС с предустановленной системой PostgreSQL команда подключения к демобазе (которая называется demo) выглядит как

```
psql -d demo
```

Если не указать и имя базы данных, то в установленном соединении будет использоваться БД, имеющая имя postgres.

Программа psql предоставляет интерфейс типа командной строки. Существуют другие клиентские программы, предоставляющие более развитый экранный интерфейс, который дает некоторые преимущества для быстрой ориентации в структуре базы данных, однако для использования сколько-нибудь сложных конструкций SQL в любом случае необходимо текстовое представление.

В командной строке psql можно вводить операторы SQL, которые необходимо завершать точкой с запятой «;», и команды программы psql, начинающиеся с символа обратной косой черты «\». Список команд с краткими описаниями того, что они делают, можно получить по команде \?.

Язык SQL, предназначенный для работы с базой данных и предоставляющий мощные средства для выполнения любых операций в базе данных, обсуждается в главе 4. Поэтому здесь мы рассмотрим только некоторые из команд psql, которые позволяют бегло ознакомиться с тем, что находится в базе данных.

Каждый объект базы данных создается в некоторой схеме. Для небольших баз данных можно использовать всего одну схему, которую можно не указывать при ссылках на объекты базы данных при выполнении операций, однако такая практика считается недопустимой в серьезных промышленных разработках. Для баз данных сложной структуры схемы позволяют объединить объекты (например, относящиеся к одному приложению) и дают ряд дополнительных возможностей, в основном связанных с использованием имен объектов. Схемы можно также применять для разграничения доступа. Такие возможности обсуждаются в главе 5.

Основным объектом логического уровня в базах данных, построенных на основе реляционной модели данных, являются двумерные *таблицы*. При этом

строки (кортежи) описывают значения различных атрибутов одной сущности, а *колонки* — значения одного атрибута, принадлежащие разным сущностям.

Кроме таблиц в базе данных могут быть определены другие объекты логического уровня: типы данных, домены, представления (view), процедуры, триггеры и др. Многие из таких объектов не содержат данных.

В программе psql имеется большой набор команд, позволяющих вывести информацию об объектах базы данных. Обычно в качестве параметра таких команд можно указать имя или шаблон, которому должны удовлетворять имена объектов, обрабатываемые такой командой.

Приведем в качестве примера результат выполнения команды \dt, которая выводит список таблиц в схеме bookings демонстрационной базы данных:

```
demo=# \dt bookings.*
              List of relations
  Schema  |      Name       | Type  |  Owner
----------+-----------------+-------+---------
 bookings | aircrafts_data  | table | student
 bookings | airports_data   | table | student
 bookings | boarding_passes | table | student
 bookings | bookings        | table | student
 bookings | flights         | table | student
 bookings | seats           | table | student
 bookings | ticket_flights  | table | student
 bookings | tickets         | table | student
(8 rows)
```

Более детальную информацию об объектах базы данных можно получить с помощью команды \d:

```
demo=# \d bookings.aircrafts_data
                Table "bookings.aircrafts_data"
    Column     |     Type     | Collation | Nullable | Default
---------------+--------------+-----------+----------+---------
 aircraft_code | character(3) |           | not null |
 model         | jsonb        |           | not null |
 range         | integer      |           | not null |
Indexes:
    "aircrafts_pkey" PRIMARY KEY, btree (aircraft_code)
Check constraints:
    "aircrafts_range_check" CHECK (range > 0)
Referenced by:
    TABLE "flights" CONSTRAINT "flights_aircraft_code_fkey"
      FOREIGN KEY (aircraft_code) REFERENCES
        aircrafts_data(aircraft_code)
    TABLE "seats" CONSTRAINT "seats_aircraft_code_fkey"
      FOREIGN KEY (aircraft_code) REFERENCES
        aircrafts_data(aircraft_code) ON DELETE CASCADE
```

Для таблиц команда \d выводит список столбцов (атрибутов) вместе с их типами и ограничением NOT NULL, а также другие ограничения целостности, определенные для этой таблицы.

Существуют другие программы-клиенты, большинство из которых использует графические средства для отображения содержимого базы данных на экране: деревья для представления структуры вложенных объектов, решетки (таблицы) и т. п. В качестве примера такой программы можно назвать pgAdmin.

Тем не менее какая-либо обработка данных, извлеченных из БД, предполагает использование программы-клиента, написанного для решения конкретных прикладных задач. Разработка приложений и организация их взаимодействия с базой данных рассматривается в главе 7.

3.4. Итоги главы

Начиная с этой главы для эффективного освоения материала книги необходимо установить СУБД PostgreSQL и демонстрационную базу данных. В начале главы указывается, каким образом получить детальные инструкции по установке. Далее показано, каким образом можно познакомиться с логическими структурами, имеющимися в базе данных, используя программу psql, которая входит в комплект системы PostgreSQL и необходима для выполнения почти всех упражнений в следующих главах этой книги.

3.5. Упражнения

Упражнение 3.1. Установите соединение с демонстрационной базой данных, используя psql.

Упражнение 3.2. С помощью команды \d (и других) получите информацию об объектах демонстрационной базы данных.

Упражнение 3.3. Установите программу pgAdmin (https://www.pgadmin.org). Найдите с ее помощью объекты демонстрационной базы данных.

Глава 4

Введение в SQL

4.1. Назначение языка SQL

Одним из основных требований к системам управления базами данных является наличие высокоуровневых средств выполнения запросов. В системах, реализующих реляционную модель данных, в качестве такого средства используется язык SQL. Фактически этот язык содержит полный набор операций, необходимых для выполнения любых действий с базой данных.

Стандарты SQL предусматривают подразделение средств языка на несколько категорий, отличающихся по их обязательности, для того чтобы реализация удовлетворяла требованиям стандарта:

- средства манипулирования данными (Data Manipulation Language, DML) обеспечивают выполнение поиска, извлечения, добавления, изменения и удаления данных, определенных в описании логической структуры базы данных, но не позволяют изменять эту структуру;
- средства определения данных (Data Definition Language, DDL) обеспечивают создание, модификацию и удаление элементов описания структуры базы данных — как логической, так и структуры хранения;
- расширения, предусмотренные стандартом, но не входящие в основное ядро языка;
- дополнительные средства, обеспечивающие описание особенностей, специфических для конкретной СУБД. Чаще всего это — определения конкретных структур хранений или их параметров.

В системе PostgreSQL имеется довольно много расширений, реализующих дополнительную функциональность системы (не предусмотренную стандартом SQL). Такие расширения, конечно, относятся к логическому уровню.

Далее в этой главе кратко описываются средства языка SQL, минимально необходимые для того, чтобы начать работу с базой данных. Более сложные конструкции, а также их использование обсуждаются в следующих главах. Ни эта глава, ни последующие не заменяют документацию или стандарт. Приводимые здесь сведения неформально описывают назначение отдельных конструкций языка, их взаимосвязи, соотношение с другими моделями баз данных, а также некоторые особенности их использования. Изложение иллюстрируется примерами, однако для всестороннего освоения языка SQL и системы PostgreSQL, конечно, необходимо изучить документацию и выполнить достаточное количество упражнений.

4.2. Быстрый старт

В этом разделе приводятся сведения о различных элементах SQL, необходимые, для того чтобы начать практическую работу с базой данных, в том числе для выполнения упражнений.

4.2.1. Простые типы данных

В теоретической реляционной модели данных значения атрибутов являются элементами доменов. Напомним, что домены могут описывать свойства некоторых конкретных типов значений, например длин, весов, денежных сумм, имен и т. п. Однако в ранних вариантах стандарта SQL не были предусмотрены средства учета специфики доменов и допускалось использование только таких типов, как числа, текстовые строки, моменты времени и некоторые другие, не привязанные к конкретным доменам.

В системе PostgreSQL имеются средства для определения пользовательских типов данных, однако в этой главе будут использоваться только типы, аналогичные типам ранних версий стандарта SQL. Определение доменов обсуждается позже.

В языке SQL предусмотрены следующие числовые типы данных:

Целые числа (int, integer) обычно представляют собой двоичные числа, для которых определены алгебраические операции и операции сравнения.

Вещественные числа (real, double precision) также обычно хранятся во внутреннем представлении, естественном для компьютера, на котором работает СУБД, и могут иметь различную точность. Конечно, для них тоже определены все обычные операции и отношения.

Десятичные числа (decimal, numeric) обрабатываются по правилам десятичной арифметики с фиксированным положением десятичной точки (отделяющей дробную часть от целой). Для этих типов можно указывать максимальное количество цифр и количество цифр после десятичной точки.

Последняя из перечисленных разновидностей числовых типов требует дополнительных пояснений. Результаты обычных арифметических операций в десятичной арифметике могут отличаться от результатов, получаемых при использовании других числовых типов. Это связано с особенностями правил округления. Для десятичных чисел, имеющих дробную часть, результаты зачастую оказываются неожиданными для программистов, привыкших работать только с целыми или вещественными числами (с плавающей точкой). Десятичные числа полезны для хранения денежных величин. Во многих странах, в которых основная денежная единица состоит из 100 более мелких единиц, законодательство требует, чтобы все вычисления выполнялись с сохранением ровно трех знаков после десятичной точки (т. е. вычисления должны выполняться с точностью до 0,1 цента или копейки, если такие правила применяются).

Отметим, однако, что промежуточные результаты в выражениях, содержащих десятичные числа, могут вычисляться с очень большой точностью, поэтому, для того чтобы получить правильное округление, необходимо принимать специальные меры.

Простой пример, иллюстрирующий особенности десятичной арифметики: процент, вычисленный от суммы нескольких значений, может не совпадать с суммой процентов, вычисленных для каждого значения отдельно.

В этом примере, который показывает особенности применения десятичной арифметики, используются операторы, действие которых объясняется ниже. Необходимость в использовании таких операторов связана с тем, что для промежуточных вычислений PostgreSQL автоматически выбирает тип данных, по возможности исключающий ошибки. Округление выполняется только при записи результатов. Поэтому в примере создается и заполняется таблица, затем вычисляются значения для колонки процентов. Далее операторы выборки данных показывают содержимое полученной таблицы и различия при подсчете суммы до и после вычисления процента.

```
demo=# CREATE TABLE fixed_precision (
    amount  numeric(7,2),
    percent numeric(7,2)
);
CREATE TABLE
demo=# INSERT INTO fixed_precision (amount) VALUES (2.13), (3.14);
INSERT 0 2
demo=# UPDATE fixed_precision
SET percent = amount * 0.03;
UPDATE 2
demo=# SELECT *
FROM fixed_precision;
 amount | percent
--------+---------
   2.13 |    0.06
   3.14 |    0.09
(2 rows)
demo=# SELECT sum(amount) * 0.03, sum(percent)
FROM fixed_precision;
 ?column? | sum
----------+------
   0.1581 | 0.15
(1 row)
```

Другие примеры, иллюстрирующие особенности использования числовых типов, содержатся в упражнениях к этой главе.

Для хранения символьных данных можно использовать **строки фиксированной длины** (char) и **строки переменной длины** (varchar, text). Тип varchar требует указания максимальной длины, а text — нет и пригоден для хранения строк большого размера. В последних версиях PostgreSQL рекомендуется использовать text во всех случаях, когда ограничение максимальной длины не требуется.

Для времени предусмотрены типы **дат** (date), **времени суток** (time) и **отметок времени** (timestamp), которые включают и дату, и время. Для типов, содержащих время (т. е. time и timestamp), имеются разновидности, включающие (или не включающие) информацию о часовом поясе. В системе PostgreSQL эти типы определены так, как предусмотрено стандартом SQL, однако, как отмечено в документации, полезность указания часового пояса без одновременного указания даты сомнительна, поэтому обычно часовой пояс указывают только для типа timestamp.

Логический тип (boolean), определенный в системе PostgreSQL, как и следовало ожидать, хранит булевы значения (истина или ложь).

4.2.2. Основные конструкции и синтаксис

Любое обращение клиентской программы к серверу базы данных, реализующему язык SQL, должно быть оформлено как *оператор SQL* (SQL statement). В языке SQL предусмотрено небольшое количество видов операторов, которые будут рассмотрены ниже. Каждый оператор начинается с ключевого слова, определяющего вид этого оператора. За этим ключевым словом обычно следуют *предложения* (clause), многие из которых также начинаются определенным ключевым словом, определяющим назначение этого предложения. Кроме ключевых слов операторы могут содержать константы, имена объектов базы данных и позиции, в которые должны быть подставлены значения переменных, передаваемых программой-клиентом, а также встроенные функции языка.

Например, оператор SELECT, приведенный ниже, возвращает текущую дату, используя функцию current_date:

```
demo=# SELECT current_date;
 current_date
--------------
 2016-03-13
(1 row)
```

В языке SQL прописные и строчные буквы эквивалентны, за исключением символьных констант и идентификаторов, заключенных в двойные кавычки. Во многих учебниках ключевые слова принято записывать прописными буквами, чтобы отличать их от нетерминальных символов в синтаксических формулах. Такой же стиль принят и в документации PostgreSQL.

4.2.3. Описание данных: отношения

Подмножество языка SQL, предназначенное для описания данных (SQL DDL), включает операторы CREATE, ALTER и DROP, а также GRANT и REVOKE, используемые для управления разграничением доступа и рассматриваемые в главе 5.

За ключевым словом, определяющим оператор, следует другое ключевое слово, определяющее тип объекта, к которому применяется оператор, — например CREATE TABLE является оператором создания таблиц. Далее обычно следует имя объекта базы данных и затем предложения, уточняющие, что именно делается при выполнении оператора. В этом разделе рассматривается только работа с таблицами на уровне логической структуры, а управление структурами хранения обсуждается далее в других разделах курса.

В системе PostgreSQL используется термин «отношение», однако он означает не отношения в смысле теоретической реляционной модели, а понятие, включающее таблицы и представления в смысле модели данных SQL, а также некоторые другие виды объектов базы данных.

Простейшая форма оператора CREATE TABLE содержит имя создаваемой таблицы, за которым следует список определений атрибутов (колонок), разделяемых запятыми. Определение каждого атрибута содержит его наименование, тип и может содержать некоторые дополнительные указания, в частности ограничения целостности и значение, присваиваемое этому атрибуту при добавлении строки в таблицу, если оно не указано явно.

Например, таблица дисциплин может быть создана с помощью следующего оператора:

```
demo=# CREATE TABLE courses (
    course_no varchar(30),
    title     text,
    credits   integer
);
CREATE TABLE
```

Последняя строка представляет собой ответ системы, подтверждающий выполнение оператора, и не является частью оператора.

Кроме определений атрибутов, список в команде CREATE TABLE может содержать ограничения целостности. Определение таблицы courses, приведенное выше, содержит не всю необходимую информацию. Более полный вариант оператора создания таблицы содержит определение первичного ключа и требование отсутствия неопределенных значений для атрибута, содержащего название курса:

```
demo=# CREATE TABLE courses (
    course_no varchar(30),
    title     text NOT NULL,
    credits   integer,
    CONSTRAINT course_pkey PRIMARY KEY (course_no)
);
CREATE TABLE
```

Заметим, что имена таблиц должны быть различны, поэтому выполнить оператор создания таблицы можно, только если таблицы с таким именем еще не существует.

Оператор DROP удаляет объект базы данных, заданный параметрами этого оператора. Например, для удаления таблицы courses можно было бы использовать следующий оператор:

```
demo=# DROP TABLE courses;
DROP TABLE
```

Оператор ALTER изменяет структуру или свойства существующего объекта. В частности, оператор ALTER TABLE можно использовать для добавления, переименования или удаления отдельных атрибутов таблицы, а также для добавления или удаления ограничений целостности.

В языке SQL предусмотрены следующие ограничения целостности:

NOT NULL запрещает появление неопределенных значений атрибута.

UNIQUE задает группу атрибутов, которые образуют возможный ключ отношения, т. е. для любых двух строк таблицы значения хотя бы одного из атрибутов, входящих в возможный ключ, должны различаться. Очень часто такое ограничение состоит всего из одного атрибута.

PRIMARY KEY задает атрибут или группу атрибутов, которые используются как первичный ключ отношения. Обычно такое ограничение определяется для атрибутов, составляющих идентификатор сущности, и поэтому обычно эти атрибуты считаются неизменяемыми, однако язык SQL допускает изменение значений атрибутов первичного ключа. Очень часто первичный ключ состоит только из одного атрибута, хотя язык SQL позволяет создавать составные первичные ключи.

Важное отличие уникального ключа от первичного состоит в том, что для атрибутов, входящих в уникальный ключ, могут допускаться неопределенные значения (NULL), а для атрибутов первичного ключа — нет.

FOREIGN KEY задает группу атрибутов, значения которых должны совпадать со значениями атрибутов первичного или уникального ключа другого отношения, указанных в этом ограничении. Таким образом, это ограничение задает бинарные связи между сущностями.

CHECK задает произвольное условие на значения одного или нескольких атрибутов в одной строке таблицы.

Невозможность выполнения операторов DDL вызывает индикацию ошибки: например, попытка создать таблицу с именем, совпадающим с именем существующей таблицы в той же схеме, или попытка удаления несуществующего объекта. В системе PostgreSQL имеются условные формы операторов языка описания данных, проверяющие возможность выполнения.

Например, оператор

```
DROP TABLE IF EXISTS old_table;
```

удалит таблицу old_table, если таблица с таким именем существовала, и ничего не сделает в противоположном случае. Для оператора CREATE условие записывается как IF NOT EXISTS, поскольку выполнение этого оператора возможно, если объект не существует.

В программе psql можно получить список доступных таблиц, используя команду \d или \dt. Команда \d с указанием имени таблицы выводит ее описание:

```
demo=# \d courses
                    Table "education.courses"
  Column   |         Type          | Collation | Nullable | Default
-----------+-----------------------+-----------+----------+---------
 course_no | character varying(30) |           | not null |
 title     | text                  |           | not null |
 credits   | integer               |           |          |
Indexes:
    "courses_pkey" PRIMARY KEY, btree (course_no)
Referenced by:
    TABLE "exams" CONSTRAINT "exams_course_no_fkey"
      FOREIGN KEY (course_no) REFERENCES courses(course_no)
```

(В выводе этой команды через точку перед именем таблицы находится указание схемы. О том, что это такое, говорится в разделе 4.5.)

Отметим, что, хотя по форме операторы определения данных выглядят как выполняемые и изменения схемы можно делать когда угодно, эти операторы следует использовать как статические. Другими словами, все таблицы, которые нужны приложению, должны быть созданы до начала работы приложения, и в дальнейшем изменения схемы производятся относительно редко (например, при изменении версии приложения или появлении новых функций группы приложений, работающих с базой данных).

Это не совсем формальное требование мотивируется тем, что работа системы управления базами данных существенно зависит от статистических характеристик накопленных данных. Слишком частое изменение схемы базы данных

может помешать выбору оптимальных алгоритмов для их обработки и, следовательно, приведет к неэффективной работе СУБД. Кроме этого, операции изменения схемы блокируют доступ к изменяемым объектам на все время выполнения операции, что может помешать нормальной работе других пользователей с этим объектом. Наконец, код приложения может зависеть от схемы базы данных.

Технически, однако, ничто не препятствует динамическому выполнению операторов DDL, а в некоторых приложениях это может быть необходимо. В частности, вре́менные объекты, которые не должны сохраняться после окончания сеанса работы с базой данных, обычно создаются операторами DDL, выполняемыми в рамках того же сеанса.

4.2.4. Заполнение таблиц

Добавление новых данных производится оператором INSERT, содержащим два предложения. Первое из них (INTO) указывает, куда помещаются новые данные, и содержит имя таблицы, за которым может следовать список атрибутов, заключенный в круглые скобки. Второе предложение (VALUES) задает значения, которые добавляются в базу данных. В этом разделе мы используем только одну форму оператора INSERT, которая добавляет в базу данных одну строку. Значения атрибутов перечисляются в круглых скобках вслед за ключевым словом VALUES.

Следующие операторы заносят в базу данных строки созданной ранее таблицы courses:

```
demo=# INSERT INTO courses (course_no, title, credits)
    VALUES ('CS301', 'Базы данных', 5);
INSERT 0 1
demo=# INSERT INTO courses (course_no,  credits, title)
    VALUES ('CS305', 10, 'Анализ данных');
INSERT 0 1
```

Ответ системы (второе число) показывает количество строк в таблице, которые были изменены, в данном случае — добавлены.

Если в предложении INTO указан список атрибутов, то и значения в предложении VALUES должны идти в том же порядке. В этом случае не важно, в каком порядке расположены атрибуты в определении таблицы.

Если в предложении INTO перечислены не все атрибуты, имеющиеся в таблице, то остальные получают значения по умолчанию, указанные в описании таблицы, или неопределенные значения (NULL), если значения в описании таблицы не заданы.

В случае если список атрибутов в предложении INTO не указан, порядок значений в предложении VALUES должен совпадать с порядком в определении таблицы. Использование такой формы оператора INSERT делает код приложения зависимым от схемы базы данных: любое добавление или изменение порядка атрибутов потребует модификации кода приложения.

С другой стороны, если в коде приложений используется формат оператора INSERT, содержащий список атрибутов, то при добавлении новых атрибутов необходимо либо разрешать неопределенные значения, либо определять значения, присваиваемые по умолчанию, если приложение их не задает.

Если при выполнении оператора INSERT нарушаются ограничения целостности, то его выполнение заканчивается с ошибкой. В системе PostgreSQL в предложении ON CONFLICT можно указать действия, которые будут выполняться вместо индикации ошибки при нарушении ограничений уникальности (в том числе уникальности первичного ключа). Например, при попытке повторной вставки записи, в которой значение первичного или уникального ключа совпадает с уже имеющимся в таблице, можно вместо вставки выполнить изменение значений других атрибутов.

Обычно СУБД предоставляют средства для массовой загрузки данных. Это можно делать с помощью другой формы оператора INSERT, позволяющей включить данные, выбираемые из других таблиц (в этой форме вместо предложения VALUES записывается оператор SELECT).

В PostgreSQL предложение VALUES может содержать данные для нескольких строк таблицы. Кроме этого, в системе PostgreSQL имеется оператор COPY (не входящий в стандарт SQL), который может копировать данные из внешних файлов, находящихся в файловой системе того компьютера, на котором выполняется сервер базы данных, или наоборот — из базы данных во внешние файлы. Оператор может работать с несколькими форматами данных, в том числе с форматом CSV (comma-separated values), который часто используется для передачи данных между системами и инструментами анализа данных.

Массовую загрузку можно выполнить и с помощью команды \copy программы psql. Эта команда может записать внешний (по отношению к базе данных, т. е. находящийся в файловой системе компьютера, на котором запущен psql) файл,

содержащий данные в одном из нескольких форматов, или, наоборот, записать содержимое таблицы в файл. Такого рода возможности, однако, не являются частью SQL, и на самом деле в реализации этой команды используются операторы SQL для работы с базой данных, а чтение или запись файлов происходит в программе-клиенте.

4.2.5. Чтение данных

Любая операция чтения из базы данных на языке SQL задается оператором SELECT. Результатом выполнения этого оператора всегда является некоторая таблица, содержимое которой может передаваться в программу-клиент, запросившую выполнение оператора, или использоваться в любой другой конструкции языка SQL, в которой может находиться таблица.

После ключевого слова SELECT в операторе размещается несколько предложений, описывающих, какой именно результат должен быть извлечен из базы данных. Подчеркнем, что оператор описывает именно результат, но не алгоритм его вычисления, даже если запись выглядит как последовательность действий. СУБД, в том числе система PostgreSQL, может изменять порядок выполнения отдельных операций, с тем чтобы выполнить все вычисление наиболее эффективным способом.

В отличие от хранимых таблиц, создаваемых оператором CREATE, не требуется заранее описывать схему таблицы, которая получается в результате выполнения оператора SELECT. Первое предложение в операторе SELECT, следующее за этим ключевым словом, содержит список выражений, вычисление которых задает значения атрибутов результата, возможно, с указанием имен атрибутов результата. Если других предложений в операторе SELECT нет, то эти выражения могут содержать константы и функции с параметрами-константами, а результирующая таблица будет содержать ровно одну строку.

Например, оператор

```
demo=# SELECT current_date AS today, current_time AS right_now;
   today    |     right_now
------------+--------------------
 2016-03-27 | 20:53:48.109582+03
```

вырабатывает таблицу с атрибутами today и right_now, содержащую строку с текущими значениями даты и времени.

Предложение FROM оператора SELECT указывает список источников данных, из которых выбираются данные для результата. Источниками данных могут быть таблицы базы данных, представления или любые конструкции SQL, вырабатывающие таблицы (такие конструкции называются табличными выражениями). Пока мы будем предполагать, что все источники данных являются таблицами базы данных.

Если предложение FROM присутствует, то в выражениях, определяющих вычисляемые значения, можно использовать имена атрибутов таблиц, перечисленных в списке. Количество выводимых строк зависит от способа комбинирования таблиц и от других предложений. В простейшем случае, когда указана только одна таблица, для каждой строки таблицы будет выведена одна строка в таблицу-результат.

Например, оператор

```
demo=# SELECT title AS course_name, credits
FROM courses;
  course_name  | credits
---------------+---------
 Базы данных   |       5
 Анализ данных |      10
(2 rows)
```

выводит таблицу, содержащую для каждой строки таблицы courses наименование курса и количество зачетных единиц. Поскольку имя второго атрибута не указано, используется имя колонки таблицы.

Вместо списка выражений можно указать символ «*». В этом случае выводятся все колонки таблицы, указанной в предложении FROM. Таким способом можно получить копию хранимой таблицы:

```
demo=# SELECT *
FROM courses;
 course_no |     title     | credits
-----------+---------------+---------
 CS301     | Базы данных   |       5
 CS305     | Анализ данных |      10
(2 rows)
```

Отметим, что использование «звездочки» приводит к зависимости приложения от схемы базы данных: приложение должно быть готово принять столько колонок, сколько их имеется в хранимой таблице, независимо от того, нужны значения этих колонок для работы приложения или нет. «Звездочка» может также приводить к снижению эффективности работы сервера базы данных,

поскольку в этом случае просматриваются все атрибуты. По этим причинам не рекомендуется применять такой способ в приложениях, однако это удобно в интерактивных ad-hoc-запросах.

Предложение WHERE задает условия, которым должны удовлетворять строки входных таблиц, для того чтобы эти строки использовались при выполнении оператора. Эти условия могут быть выражены как:

- бинарные отношения, определенные на доменах атрибутов, связывающие значения атрибутов с константами или значения разных атрибутов или выражений (например, для числовых доменов определены бинарные отношения $=, \neq, >, \geqslant, <, \leqslant$);
- логические выражения, получаемые из более простых выражений при помощи операций конъюнкции (AND), дизъюнкции (OR) и отрицания (NOT) и круглых скобок;
- функции, вырабатывающие логические значения.

Например, оператор

```
demo=# SELECT title, credits
FROM courses
WHERE credits > 8;
     title     | credits
---------------+---------
 Анализ данных |      10
(1 row)
```

вырабатывает список курсов, по которым можно получить более 8 зачетных единиц. Отметим, что атрибуты, используемые в условиях, не обязательно должны включаться в число атрибутов результата.

4.2.6. Модификация данных

Оператор UPDATE изменяет значения атрибутов в хранимой таблице. За ключевым словом UPDATE следуют имя таблицы и предложение SET, содержащее список имен изменяемых атрибутов из этой таблицы с указанием выражений, задающих для них новые значения. Далее следует предложение WHERE (такое же, как в операторе SELECT), которое определяет, какие строки будут обрабатываться данным оператором. Если предложение WHERE не указано, изменению подвергаются все строки таблицы.

Оператор UPDATE часто используется для изменения только одной строки таблицы, хотя SQL допускает указание условий, которым удовлетворяет несколько строк. Отметим также, что для изменения значения не требуется его предварительное считывание в приложение оператором SELECT: выражение, вычисляющее новое значение атрибута, может содержать имя этого атрибута, при этом для вычисления нового значения будет использоваться старое. Новое значение может задаваться подзапросом, эта возможность более детально обсуждается ниже.

Например, оператор

```
demo=# UPDATE courses
SET credits = credits + 1
WHERE course_no = 'CS305';
UPDATE 1
```

увеличивает на единицу значение одного атрибута в одной строке.

Оператор DELETE удаляет строки из хранимой таблицы, имя которой указывается в предложении FROM, а строки, подлежащие удалению, задаются предложением WHERE. Как и для операторов SELECT и UPDATE, отсутствие предложения WHERE означает, что должны быть обработаны (т. е. удалены) все строки таблицы. Так же как и оператор UPDATE, чаще всего оператор DELETE используется для удаления только одной строки.

Так, оператор

```
demo=# DELETE FROM courses
WHERE course_no = 'CS305';
DELETE 1
```

удаляет одну строку таблицы (так как условие задает значение первичного ключа).

4.3. Запросы

В этом разделе показывается, каким образом можно выразить на языке SQL основные операции реляционной алгебры и как строить более сложные запросы. Для примеров, кроме уже показанной таблицы курсов, понадобятся также таблицы студентов и экзаменов.

Если используется готовый образ системы с PostgreSQL (как описано в главе 3), то эти таблицы уже есть в базе данных demo; если нет — их можно создать следующими командами:

```
CREATE TABLE students (
    stud_id    integer PRIMARY KEY,
    name       text NOT NULL,
    start_year integer NOT NULL
);
CREATE TABLE exams (
    stud_id    integer     REFERENCES students(stud_id),
    course_no  varchar(30) REFERENCES courses(course_no),
    exam_date  date,
    grade      integer NOT NULL,
    PRIMARY KEY(stud_id,course_no,exam_date)
);
INSERT INTO students
    VALUES (1451, 'Анна', 2014),
           (1432, 'Виктор', 2014),
           (1556, 'Нина', 2015);
INSERT INTO exams
    VALUES (1451, 'CS301', '2016-05-25', 5),
           (1556, 'CS301', '2017-05-23', 5),
           (1451, 'CS305', '2016-05-25', 5),
           (1432, 'CS305', '2016-05-25', 4);
```

4.3.1. Фильтрация и проекция

Операция фильтрации, по-видимому, является наиболее часто используемой, так как именно эта операция позволяет выбрать те данные, которые нужны приложению в данный момент. Фактически запись операции фильтрации на языке SQL уже приведена выше: условие фильтрации задается предложением WHERE, при этом предложение FROM должно содержать ровно одно табличное выражение (т. е. ту таблицу, данные из которой фильтруются), а условие, записанное в предложении WHERE, содержит только константы и атрибуты этой таблицы.

Операция проекции включает в результат только указанные атрибуты исходного отношения. При этом разные строки исходной таблицы могут оказаться совпадающими по значениям оставшихся атрибутов, поэтому в реляционной модели такие строки должны быть представлены одной строкой результата. Поскольку в модели данных SQL дубликаты допускаются, для их устранения после ключевого слова SELECT указывается еще одно ключевое слово DISTINCT.

Например, если таблица студентов содержит следующие данные:

```
demo=# SELECT *
FROM students;
 stud_id |  name  | start_year
---------+--------+------------
    1451 | Анна   |       2014
    1432 | Виктор |       2014
    1556 | Нина   |       2015
(3 rows)
```

то операция проекции, извлекающая все различные годы поступления, может выглядеть так:

```
demo=# SELECT DISTINCT start_year
FROM students;
 start_year
------------
       2015
       2014
(2 rows)
```

Ключевое слово DISTINCT является альтернативой для слова ALL, которое предполагается по умолчанию и поэтому обычно не указывается.

В некоторых случаях указание DISTINCT может существенно увеличивать время выполнения операторов SQL, поэтому не следует его указывать без необходимости. Например, если среди выбираемых атрибутов присутствуют все атрибуты первичного ключа, то строки результата будут различны.

Следует также заметить, что иногда кажущаяся необходимость в использовании ключевого слова DISTINCT может быть следствием ошибки в другой части программы или в данных. Дубликаты могут появляться, если программист забыл указать некоторые из условий фильтрации. Если используются генерируемые (суррогатные) первичные ключи, ошибки в коде приложения могут приводить к повторной вставке строк, описывающих сущности, уже представленные в таблице. Необдуманное использование DISTINCT в подобных случаях существенно затрудняет поиск ошибки.

4.3.2. Произведение и соединение

Операция прямого произведения задается на языке SQL указанием нескольких таблиц в предложении FROM, как показано на рис. 4.3.1.

```
demo=# SELECT *
FROM courses, exams;
 course_no |     title      | credits | stud_id | course_no | exam_date  | grade
-----------+----------------+---------+---------+-----------+------------+-------
 CS301     | Базы данных    |       5 |    1451 | CS301     | 2016-05-25 |     5
 CS305     | Анализ данных  |      10 |    1451 | CS301     | 2016-05-25 |     5
 CS301     | Базы данных    |       5 |    1556 | CS301     | 2017-05-23 |     5
 CS305     | Анализ данных  |      10 |    1556 | CS301     | 2017-05-23 |     5
 CS301     | Базы данных    |       5 |    1451 | CS305     | 2016-05-25 |     5
 CS305     | Анализ данных  |      10 |    1451 | CS305     | 2016-05-25 |     5
 CS301     | Базы данных    |       5 |    1432 | CS305     | 2016-05-25 |     4
 CS305     | Анализ данных  |      10 |    1432 | CS305     | 2016-05-25 |     4
(8 rows)
```

Рис. 4.3.1. Прямое произведение в SQL

Как отмечено в главе 2, операция прямого произведения сама по себе не очень полезна, так как, по существу, ее результат не содержит дополнительной информации. Однако в том случае, если предложение FROM содержит несколько таблиц, в предложении WHERE можно задавать условия, связывающие атрибуты из разных таблиц, и таким образом задавать операцию соединения, как показано на рис. 4.3.2.

Поскольку имена атрибутов должны быть уникальными только в пределах одной таблицы, при совпадении имен атрибутов из разных таблиц необходимо указание, из какой таблицы следует выбирать значение атрибута. В приведенном примере атрибут course_no задан в условии с указанием имен таблиц.

В предложении WHERE можно перемешивать в произвольном порядке условия, задающие операцию фильтрации (которые содержат атрибуты только одной таблицы) и условия, задающие операцию соединения (которые связывают атрибуты разных таблиц).

В языке SQL существует альтернативная форма записи операции соединения с использованием ключевого слова JOIN, показанная на рис. 4.3.3.

Система обрабатывает оба варианта записи операции соединения одинаково. Ни результат, ни алгоритм выполнения не зависят от способа записи, поэтому можно использовать такой способ, который представляется более наглядным. Формально, однако, различие существенно: при использовании ключевого слова JOIN предложение FROM содержит одно табличное выражение, а не две таблицы.

При этом в предложении WHERE можно указать условия фильтрации:

```
demo=# SELECT students.name, exams.grade
FROM students
    JOIN exams ON students.stud_id = exams.stud_id
WHERE course_no = 'CS305';
  name  | grade
--------+-------
 Анна   |     5
 Виктор |     4
(2 rows)
```

Этот пример показывает, что операция соединения не включает в результат данные из строк исходных таблиц, для которых не нашлось пары в другой таблице: условие фильтрации ограничивает только выбор дисциплины, но при этом исключаются и студенты, которые указанную дисциплину не сдавали. Такое поведение вполне соответствует реляционной теории, однако во многих

```
demo=# SELECT *
FROM courses, exams
WHERE courses.course_no = exams.course_no;
 course_no |     title     | credits | stud_id | course_no | exam_date  | grade
-----------+---------------+---------+---------+-----------+------------+-------
 CS301     | Базы данных   |       5 |    1451 | CS301     | 2016-05-25 |     5
 CS301     | Базы данных   |       5 |    1556 | CS301     | 2017-05-23 |     5
 CS305     | Анализ данных |      10 |    1451 | CS305     | 2016-05-25 |     5
 CS305     | Анализ данных |      10 |    1432 | CS305     | 2016-05-25 |     4
(4 rows)
```

Рис. 4.3.2. Условия соединения в предложении WHERE

```
demo=# SELECT *
FROM courses
    JOIN exams ON courses.course_no = exams.course_no;
 course_no |     title     | credits | stud_id | course_no | exam_date  | grade
-----------+---------------+---------+---------+-----------+------------+-------
 CS301     | Базы данных   |       5 |    1451 | CS301     | 2016-05-25 |     5
 CS301     | Базы данных   |       5 |    1556 | CS301     | 2017-05-23 |     5
 CS305     | Анализ данных |      10 |    1451 | CS305     | 2016-05-25 |     5
 CS305     | Анализ данных |      10 |    1432 | CS305     | 2016-05-25 |     4
(4 rows)
```

Рис. 4.3.3. Условия соединения с явным оператором JOIN

случаях бывает необходимо отображать информацию из строк, не попадающих в результат соединения.

В языке SQL предусмотрены варианты операции соединения, которые не являются в строгом смысле реляционными, но решают указанную задачу и поэтому часто используются на практике. Операция левого внешнего соединения (LEFT OUTER JOIN) возвращает, кроме обычного результата соединения, строки из левого операнда, для которых не нашлось парной строки в правом операнде. При этом вместо значений атрибутов правого операнда возвращаются неопределенные значения.

```
demo=# SELECT students.name, exams.grade
FROM students
    LEFT JOIN exams ON students.stud_id = exams.stud_id
                   AND course_no = 'CS305';
  name   | grade
--------+-------
 Анна    |     5
 Виктор |     4
 Нина    |
(3 rows)
```

Аналогично определяется правое (RIGHT) соединение, включающее строки второго операнда, для которых не нашлось пары в первом, и полное (FULL) внешнее соединение, которое включает непарные строки из обоих операндов.

Отметим, что в последнем примере условие фильтрации помещено в предложении ON вместе с условием соединения, а не в предложении WHERE. Это необходимо, потому что условия, записанные в предложении WHERE, вычисляются так, как будто они проверяются после вычисления табличного выражения в предложении FROM. Строки, для которых не нашлось пары (в нашем примере строка «Нина»), содержат неопределенные значения атрибута course_no, сравнение которого с любым значением дает результат NULL, и поэтому такие строки будут исключены из результата. Другими словами, внешнее соединение выполнится, как внутреннее:

```
demo=# SELECT students.name, exams.grade
FROM students
    LEFT JOIN exams ON students.stud_id = exams.stud_id
WHERE course_no = 'CS305';
  name   | grade
--------+-------
 Анна    |     5
 Виктор |     4
(2 rows)
```

Особенности использования условий при наличии неопределенных значений, конечно, никак не связаны ни с операцией внешнего соединения, ни с отношением равенства. Для того чтобы исключить неожиданности при использовании в предикатах фильтрации колонок, допускающих неопределенные значения, следует включать дополнительно проверку значений с помощью встроенных предикатов IS NULL или IS NOT NULL.

4.3.3. Псевдонимы для таблиц

Использование имен таблиц для уточнения принадлежности атрибутов может оказаться невозможным или недостаточным. В предложении FROM могут задаваться не хранимые таблицы, а табличные выражения, которые не имеют своих имен, как таблицы. Кроме этого, одна и та же таблица может использоваться в одном запросе в нескольких разных ролях. Например, в схеме демонстрационной БД авиабилетов, представленной на рис. 2.6.3, для того чтобы вывести наименования пунктов отправления и прибытия, необходимо выполнить соединение с таблицей аэропортов дважды в одном запросе.

В подобных случаях применяется аппарат *псевдонимов* (aliases): в предложении FROM после любого табличного выражения можно указать идентификатор, который будет использоваться в других частях запроса как псевдоним этого табличного выражения.

Использование псевдонимов для указания ролей таблиц показано в следующем примере из демонстрационной базы данных.

```
demo=# SELECT f.flight_no,
    f.departure_airport AS d_airport,
    dep.city AS d_city,
    f.arrival_airport AS a_airport,
    arr.city AS a_city
FROM flights f
    JOIN airports dep ON f.departure_airport = dep.airport_code
    JOIN airports arr ON f.arrival_airport = arr.airport_code
WHERE f.status = 'Departed'
  AND f.scheduled_arrival < bookings.now();
 flight_no | d_airport |   d_city    | a_airport |     a_city
-----------+-----------+-------------+-----------+----------------
 PG0496    | SVO       | Москва      | JOK       | Йошкар-Ола
 PG0574    | OVB       | Новосибирск | HMA       | Ханты-Мансийск
 PG0304    | SGC       | Сургут      | SVX       | Екатеринбург
(3 rows)
```

Зачастую псевдонимы используются для сокращения записи. Например, один из предыдущих запросов, возвращающий оценки, полученные студентами по некоторой дисциплине, можно переписать следующим образом:

```
demo=# SELECT s.name, e.grade
FROM students s
    LEFT JOIN exams e ON s.stud_id = e.stud_id
                      AND e.course_no = 'CS305';
  name  | grade
--------+-------
 Анна   |     5
 Виктор |     4
 Нина   |
(3 rows)
```

Однако использование псевдонимов для указания роли таблицы в запросе, конечно, намного важнее.

4.3.4. Вложенные подзапросы

Любой оператор SQL, вычисляющий какой-либо результат, оформляет этот результат в виде таблицы. Такие вычисленные таблицы можно не только читать в прикладной программе, но и использовать в других запросах. Оператор SQL, вырабатывающий таблицу, после заключения в круглые скобки становится табличным выражением и может использоваться в качестве таблицы в другом операторе. Конечно, такую таблицу нельзя модифицировать: табличные выражения используются только для выборки данных из них.

Табличные выражения, возвращающие одну строку и один атрибут (т. е. содержащие только одно скалярное значение), в языке SQL считаются совпадающими с этими скалярными выражениями. Такие табличные выражения можно использовать в другом запросе в любом месте, в котором требуется скалярное значение.

Подзапросы в списке выбираемых значений

Подзапросы, возвращающие одно скалярное значение, можно использовать в списке выражений после ключевого слова SELECT.

```
demo=# SELECT
    f.flight_no,
    f.departure_airport AS d_airport,
    ( SELECT city
      FROM airports
      WHERE airport_code = f.departure_airport
    ) AS d_city,
    f.arrival_airport AS a_airport,
    ( SELECT city
      FROM airports
      WHERE airport_code = f.arrival_airport
    ) AS a_city
FROM flights f
WHERE f.status = 'Departed'
  AND f.scheduled_arrival < bookings.now();
 flight_no | d_airport |   d_city    | a_airport |     a_city
-----------+-----------+-------------+-----------+----------------
 PG0496    | SVO       | Москва      | JOK       | Йошкар-Ола
 PG0574    | OVB       | Новосибирск | HMA       | Ханты-Мансийск
 PG0304    | SGC       | Сургут      | SVX       | Екатеринбург
(3 rows)
```

Если вложенный подзапрос в списке SELECT возвращает ровно одну строку, то этот оператор эквивалентен оператору, приведенному в предыдущем подразделе, и возвращает точно такой же результат.

Если вложенный запрос для какой-либо строки основного запроса возвращает больше, чем одну строку, выполнение всего оператора прекращается с индикацией ошибки.

Если же вложенный подзапрос не возвращает ни одной строки, в системе PostgreSQL в качестве значения подзапроса используется NULL, однако в других системах это может считаться ошибкой.

Форма оператора с вложенными подзапросами может использоваться, например, для улучшения читаемости кода. Это может быть целесообразно, если из таблицы выбирается только один атрибут.

Отметим, что каждый вложенный подзапрос превращается в операцию соединения. В приведенном выше запросе это не приводит к потере эффективности, потому что, хотя соединения выполняются с одной и той же таблицей, но роли этой таблицы и, главное, предикаты соединения различаются. Однако в следующем примере вложенные подзапросы извлекают значения из одной и той же строки, поэтому использование такой формы записи может привести к потере эффективности по сравнению с использованием явного соединения.

```
demo=# SELECT
    ( SELECT courses.course_no FROM courses
      WHERE courses.course_no  = exams.course_no
    ) AS course_no,
    ( SELECT courses.title FROM courses
      WHERE courses.course_no  = exams.course_no
    ) AS title,
    exams.exam_date,
    exams.stud_id,
    exams.grade
FROM exams;
 course_no |     title     | exam_date  | stud_id | grade
-----------+---------------+------------+---------+-------
 CS301     | Базы данных   | 2016-05-25 |    1451 |     5
 CS301     | Базы данных   | 2017-05-23 |    1556 |     5
 CS305     | Анализ данных | 2016-05-25 |    1451 |     5
 CS305     | Анализ данных | 2016-05-25 |    1432 |     4
(4 rows)
```

В этом примере выполняются два соединения с таблицей courses, хотя для получения результата достаточно выполнить соединение один раз.

Подзапросы в условии фильтрации

Вложенные подзапросы, возвращающие скалярные значения, можно использовать в предложении WHERE, для того чтобы задать условия на атрибуты другой таблицы, например:

```
demo=# SELECT stud_id, grade, course_no
FROM exams
WHERE (SELECT start_year
       FROM students
       WHERE students.stud_id = exams.stud_id) > 2014;
 stud_id | grade | course_no
---------+-------+-----------
    1556 |     5 | CS301
(1 row)
```

Как и для подзапросов в списке выбираемых значений, вложенные подзапросы представляют собой неявные операции соединения.

В системе PostgreSQL можно также использовать условия, которые сравнивают несколько скалярных выражений с подзапросом, возвращающим одну строку, содержащую столько же колонок. В этом случае скалярные выражения должны быть заключены в скобки.

Предикаты, использующие подзапросы

В языке SQL имеются средства, позволяющие формулировать условия на подзапросы, возвращающие произвольное количество строк. Бинарное отношение IN возвращает истинное значение тогда и только тогда, когда скалярный левый операнд этого отношения содержится в таблице, возвращаемой вторым операндом (заключенным в скобки).

Например, следующий запрос вычисляет список студентов, получивших оценки по указанному курсу:

```
demo=# SELECT name, start_year
FROM students
WHERE stud_id IN (
    SELECT stud_id FROM exams WHERE course_no = 'CS305'
);
  name  | start_year
--------+------------
 Виктор |       2014
 Анна   |       2014
(2 rows)
```

Так же как и в случае подзапросов, возвращающих одну строку, в подзапросе можно выбирать несколько колонок и, соответственно, сопоставлять их значения с несколькими скалярными значениями, заключенными в круглые скобки.

Как и в предыдущих примерах, при использовании подзапроса выполняется неявная операция соединения. Отличие, однако, состоит в том, что будет выведена только одна строка результата для каждой строки из первой таблицы, для которой нашлась парная во второй, даже если парных строк несколько. В реляционной алгебре такая операция называется полусоединением.

Бинарное отношение NOT IN работает так же, как и отношение IN, однако возвращает противоположный результат (т. е. ложное значение, если скаляр содержится в таблице).

Следующий запрос возвращает список студентов, получивших только отличные оценки:

```
demo=# SELECT name, start_year
FROM students
WHERE stud_id NOT IN (SELECT stud_id FROM exams WHERE grade < 5);
 name | start_year
------+------------
 Анна |       2014
 Нина |       2015
(2 rows)
```

Неявное соединение в этом случае соответствует реляционной операции антисоединения (выбираются строки, для которых не нашлось пары во второй таблице).

Те же самые условия можно выразить и с помощью предиката EXISTS, который возвращает истинное значение, если подзапрос, заданный его операндом, возвращает непустую таблицу. Этот предикат, как и любой другой, можно использовать вместе с операцией логического отрицания NOT.

Список студентов, имеющих только отличные оценки, можно получить и таким способом:

```
demo=# SELECT name, start_year
FROM students
WHERE NOT EXISTS (
    SELECT stud_id
    FROM exams
    WHERE grade < 5 AND exams.stud_id = students.stud_id
);
 name | start_year
------+------------
 Анна |       2014
 Нина |       2015
(2 rows)
```

Предикат EXISTS можно рассматривать как реализацию квантора существования ($\exists$), а его отрицание можно использовать как квантор всеобщности ($\forall$).

Подзапросы как источники данных

Вложенные подзапросы можно использовать, наряду с другими конструкциями, в предложении FROM — как в качестве отдельных таблиц в списке, так и в качестве аргументов для операций JOIN.

4.3.5. Упорядочивание результата

В реляционной теории результат любого запроса является множеством и поэтому не имеет никакого упорядочения. Однако для практического использования часто бывает важно получать строки результата в определенном порядке. В языке SQL такое упорядочивание результата можно задать с помощью предложения ORDER BY, в котором указывается список значений, по которым требуется выполнить сортировку.

При этом для каждого ключа можно указать направление сортировки: по возрастанию — ASC, по убыванию — DESC. Если задано несколько ключей, то их значения используются лексикографически (значение второго ключа используется, если значения первого совпадают, и т. д.).

В следующем примере результаты экзаменов выводятся по возрастанию оценки, номера студенческого билета и номера курса:

```
demo=# SELECT *
FROM exams
ORDER BY grade, stud_id, course_no;
 stud_id | course_no | exam_date  | grade
---------+-----------+------------+-------
    1432 | CS305     | 2016-05-25 |     4
    1451 | CS301     | 2016-05-25 |     5
    1451 | CS305     | 2016-05-25 |     5
    1556 | CS301     | 2017-05-23 |     5
(4 rows)
```

Обычно упорядочивание делается по атрибутам результата, однако можно использовать и другие выражения.

4.3.6. Агрегирование и группировка

Средства агрегирования позволяют разместить в одной строке результата значения, вычисленные на основе данных из всех строк исходных таблиц или групп строк.

При агрегировании в качестве значений атрибутов можно использовать только выражения, содержащие вызовы агрегирующих функций или выражения, по которым производится группировка строк.

Стандартом SQL предусмотрены следующие агрегирующие функции:

count вычисляет количество значений выражений, указанных в качестве аргумента;

avg вычисляет среднее значение атрибута (или выражения, зависящего от атрибутов исходных отношений);

sum суммирует значения атрибута или выражения;

min находит минимальное значение атрибута или выражения;

max находит максимальное значение атрибута или выражения.

В системе PostgreSQL существуют и другие функции агрегирования, а также можно определить собственные.

Следующий пример показывает, как вычисляются некоторые из агрегирующих функций по всем строкам таблицы:

```
demo=# SELECT count(*), count(DISTINCT stud_id), avg(grade)
FROM exams;
 count | count |        avg
-------+-------+--------------------
     4 |     3 | 4.7500000000000000
(1 row)
```

Функция count допускает в качестве параметра символ «*», при этом функция возвращает количество агрегируемых строк. Вариант со словом DISTINCT подсчитывает количество различных значений указанного выражения, а если вместо выражения указана «звездочка» — количество различных строк.

Для того чтобы вычислить агрегирующие функции по отдельности для каждой группы строк, необходимо задать значения, по которым будет определяться принадлежность строки к определенной группе. Обычно в качестве значений используются атрибуты, однако можно использовать любые выражения, зависящие от атрибутов исходной таблицы или таблиц. Список выражений, по которым производится группировка (ключей группировки), задается в предложении GROUP BY.

Например, следующий запрос выдает агрегированную информацию о курсах:

```
demo=# SELECT courses.title, count(*), avg(exams.grade)
FROM exams
    JOIN courses ON exams.course_no = courses.course_no
GROUP BY courses.title;
     title     | count |        avg
---------------+-------+--------------------
 Базы данных   |     2 | 5.0000000000000000
 Анализ данных |     2 | 4.5000000000000000
(2 rows)
```

Отметим, что совокупность ключей группировки составляет ключ отношения, которое получается в результате выполнения запроса.

Спецификация агрегирования в языке SQL имеет некоторые нелогичности. Так, при вычислении всех функций, кроме count, не учитываются неопределенные значения. В результате деление суммы на количество может давать значения, отличающиеся от значения avg. Кроме этого, агрегирование без указания

GROUP BY всегда возвращает ровно одну строку результата, даже если исходная таблица пуста (или нет строк, удовлетворяющих критериям фильтрации):

```
demo=# SELECT count(*) FROM exams WHERE grade > 200;
 count
-------
     0
(1 row)
```

При добавлении GROUP BY получается пустой результат:

```
demo=# SELECT count(*) FROM exams WHERE grade > 200
GROUP BY course_no;
 count
-------
(0 rows)
```

В запросах, использующих группировку, может возникнуть необходимость в фильтрации на основе значений, полученных в результате агрегирования. Такие условия фильтрации можно задать в предложении HAVING. Его отличие от предложения WHERE состоит в том, что в предложении WHERE можно использовать атрибуты исходных таблиц, а в предложении HAVING — атрибуты результата, т. е. агрегированные значения.

```
demo=# SELECT stud_id
FROM exams
WHERE grade = 5
GROUP BY stud_id
HAVING count(*) > 1;
 stud_id
---------
    1451
(1 row)
```

Некоторые условия фильтрации, например ограничения на значения ключа группировки, можно проверять как в предложении WHERE, так и в HAVING. Результаты будут одинаковыми.

4.3.7. Теоретико-множественные операции

Использование теоретико-множественных операций (объединение, пересечение, разность) проиллюстрируем на примерах выборки из самой маленькой таблицы в демонстрационной базе данных: aircrafts. Во всех примерах будем использовать два подмножества строк из этой таблицы. Конечно, для

теоретико-множественных операций не требуется, чтобы данные выбирались из одной таблицы, необходимо только, чтобы совпадали типы всех выбираемых колонок.

Операция объединения записывается следующим образом:

```
demo=# SELECT * FROM aircrafts WHERE range > 4500
UNION
SELECT * FROM aircrafts WHERE range < 7500;
 aircraft_code |        model         | range
---------------+----------------------+-------
 CR2           | Бомбардье CRJ-200    |  2700
 CN1           | Сессна 208 Караван   |  1200
 733           | Боинг 737-300        |  4200
 319           | Аэробус A319-100     |  6700
 321           | Аэробус A321-200     |  5600
 320           | Аэробус A320-200     |  5700
 773           | Боинг 777-300        | 11100
 763           | Боинг 767-300        |  7900
 SU9           | Сухой Суперджет-100  |  3000
(9 rows)
```

Конечно, условия в подзапросах таковы, что объединение совпадает со всей таблицей aircrafts, и запрос извлекает содержимое этой таблицы далеко не лучшим образом. Заметим, что в этом случае результат, эквивалентный объединению, можно получить, записав условия фильтрации в одном предложении WHERE и соединив их логической операцией OR. Но этот прием невозможно применить, если объединяемые множества выбираются из разных таблиц.

В SQL имеется вариант операции объединения, который не удаляет дублирующие строки из результата:

```
demo=# SELECT * FROM aircrafts WHERE range > 4500
UNION ALL
SELECT * FROM aircrafts WHERE range < 7500;
 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Боинг 777-300        | 11100
 763           | Боинг 767-300        |  7900
 320           | Аэробус A320-200     |  5700
 321           | Аэробус A321-200     |  5600
 319           | Аэробус A319-100     |  6700
 SU9           | Сухой Суперджет-100  |  3000
 320           | Аэробус A320-200     |  5700
 321           | Аэробус A321-200     |  5600
 319           | Аэробус A319-100     |  6700
 733           | Боинг 737-300        |  4200
 CN1           | Сессна 208 Караван   |  1200
 CR2           | Бомбардье CRJ-200    |  2700
(12 rows)
```

Получить такой результат с помощью каких-либо комбинаций логических операций невозможно, потому что проверка условий WHERE никак не может привести к появлению дубликатов.

Следующий запрос вычисляет пересечение тех же множеств. Очевидно, он возвращает те самые строки, которые были дубликатами в предыдущем запросе:

```
demo=# SELECT * FROM aircrafts WHERE range > 4500
INTERSECT
SELECT * FROM aircrafts WHERE range < 7500;
 aircraft_code |      model       | range
---------------+------------------+-------
 321           | Аэробус A321-200 |  5600
 320           | Аэробус A320-200 |  5700
 319           | Аэробус A319-100 |  6700
(3 rows)
```

Как и для объединения, в данном случае такой же результат можно получить, записывая условия фильтрации в одном предложении WHERE, но логическая операция в этом случае будет AND. Как и для объединения, такая замена возможна, только если пересекаемые множества выбираются из одной таблицы (точнее, должны совпадать предложения FROM).

Операция разности вырабатывает строки, которые входят в первое множество, но не входят во второе:

```
demo=# SELECT * FROM aircrafts WHERE range > 4500
EXCEPT
SELECT * FROM aircrafts WHERE range < 7500;
 aircraft_code |     model     | range
---------------+---------------+-------
 773           | Боинг 777-300 | 11100
 763           | Боинг 767-300 |  7900
(2 rows)
```

Эквивалентное логическое выражение должно содержать операции AND NOT.

4.3.8. Вывод результатов после модификации данных

Операторы обновления обычно не вырабатывают никаких результатов, кроме изменений в базе данных и кода, характеризующего успешность выполнения операции (в том числе количество строк, обработанных оператором). Но в некоторых случаях бывает необходима более детальная информация об измененных строках.

Предложение RETURNING задает список тех значений, которые будут выведены после выполнения операции модификации. При этом для каждой строки, обработанной оператором, выводится одна строка в результирующей таблице. Список выражений, специфицирующих атрибуты результата, строится точно так же, как список выражений в предложении SELECT, в том числе можно указывать «звездочку», для того чтобы вывести все атрибуты. Предложение RETURNING можно использовать во всех операторах модификации.

```
demo=# UPDATE courses
SET credits = 12
WHERE course_no = 'CS305'
RETURNING *;
 course_no |     title     | credits
-----------+---------------+---------
 CS305     | Анализ данных |      12
(1 row)
UPDATE 1
```

Для операций INSERT и UPDATE для формирования вывода используются значения, получившиеся после выполнения модификации, а для оператора DELETE — значения, которые были в удаленных из таблицы строках.

4.3.9. Последовательности

Одним из методов идентификации объектов, рассмотренных в главе 2, является применение суррогатных ключей. Значения таких ключей генерируются информационной системой таким образом, что каждое новое значение отличается от всех ранее полученных. Это гарантирует однозначность идентификации.

Генерация уникальных значений реализуется в SQL с помощью механизма *последовательностей* (sequence). В системе PostgreSQL последовательности рассматриваются как особый вид отношений, содержащих одну колонку и одну строку. Как и другие объекты базы данных, последовательности создаются оператором CREATE:

```
demo=# CREATE SEQUENCE my_seq;
CREATE SEQUENCE
```

Для последовательностей можно задавать начальное значение, шаг, конечное значение, а также поведение системы, в случае если достигнуто последнее значение. При исчерпании заданного диапазона может возбуждаться ошибка (при

каждой попытке получения нового значения) или может происходить переход к начальному значению. В последнем случае, конечно, уникальность не обеспечивается. Такие циклические последовательности не годятся для идентификации, но могут быть полезны в других случаях.

Для получения нового уникального значения последовательности вызывается функция nextval:

```
demo=# SELECT nextval('my_seq');
 nextval
---------
       1
(1 row)
demo=# SELECT nextval('my_seq');
 nextval
---------
       2
(1 row)
```

Подчеркнем, что применение функции nextval может гарантировать уникальность, но не гарантирует никаких других свойств получаемых значений. В частности, возможны пропуски некоторых значений. Это может происходить в результате обрыва транзакций (обсуждается более подробно в главе 6) или при одновременной работе нескольких приложений, выбирающих уникальные значения из одной последовательности.

Извлечь последнее сгенерированное в текущем сеансе значение последовательности можно обычным оператором SELECT, используя функцию currval:

```
demo=# SELECT currval('my_seq');
 currval
---------
       2
(1 row)
```

Функциональность последовательностей можно реализовать, обновляя значение обычного атрибута в любой постоянно хранимой таблице, но подобная реализация неизбежно окажется значительно менее эффективной, чем применение последовательности. Это связано с тем, что при работе с последовательностями не нужны и не используются средства изоляции транзакций, которые необходимы при работе с обычными таблицами.

В системе PostgreSQL в качестве типа данных колонки таблицы можно указать serial. В этом случае будет создана последовательность и функция nextval будет задана как значение по умолчанию для этой колонки. Если при вставке строк

значение колонки не указывается, будет выбрано следующее значение из последовательности. Такие колонки можно использовать в качестве первичных ключей, содержащих суррогатные значения.

Начиная с версии 10, в PostgreSQL реализована предусмотренная стандартом SQL конструкция GENERATED AS IDENTITY, обеспечивающая такие же функции, т. е. генерацию уникальных значений на основе последовательности. Использование этой конструкции предпочтительнее, чем serial, поскольку в этом случае последовательность управляется не как самостоятельный объект базы данных, а как неотъемлемая часть определения колонки.

Изменение параметров последовательности выполняется оператором ALTER, а удаление из базы данных — оператором DROP. Последовательности, созданные системой для колонок типа serial, удаляются из базы данных автоматически при удалении таблицы.

4.3.10. Представления

В соответствии с трехуровневой моделью описания данных приложение должно иметь возможность использовать структуры данных, отличающиеся от хранимых в базе данных (внешняя схема может отличаться от концептуальной). В языке SQL такая возможность реализуется с помощью аппарата *представлений* (view). Можно сказать, что представление является виртуальной таблицей: имя представления можно использовать в запросах в тех же местах, где можно использовать имена таблиц, однако содержимое представления специфицируется запросом (оператором SELECT) и обычно не хранится в базе данных.

Представления можно использовать в операторах выборки данных и при вычислении условий в любых операторах, однако в общем случае обновление представлений возможно только для ограниченного класса запросов, определяющих представления. Смысл этих ограничений в том, что по строке представления должна однозначно вычисляться строка хранимой таблицы, к которой нужно применить обновление. Это условие является необходимым, но СУБД может накладывать дополнительные ограничения, облегчающие проверку этого условия.

В системе PostgreSQL существует два механизма, позволяющих существенно расширить класс запросов, которые допускают операции обновления: это механизм правил, позволяющий переопределить способ выполнения любых операций на конкретном объекте, и механизм триггеров INSTEAD OF, которые опи-

сывают действия, выполняемые при попытке обновления тех объектов, на которых такие триггеры определены.

Другое ограничение на запросы, используемые в определениях представлений, состоит в том, что все выбираемые колонки (атрибуты результата) должны быть именованы. Иными словами, если выбираемое значение задается выражением, то такое выражение должно иметь псевдоним в определении запроса.

Кроме изменения структуры и формата данных, представления часто используются, для того чтобы записать сложные или часто встречающиеся подзапросы. Во втором случае представления можно рассматривать как некоторый аналог процедур или функций в императивных языках программирования.

Следующее представление использует таблицы демонстрационной базы данных и содержит список всех запланированных на ближайшую неделю рейсов с указанием даты вылета и кодов аэропортов отправления и прибытия:

```
demo=# CREATE VIEW upcoming_flights AS
    SELECT f.flight_id,
        f.flight_no,
        f.scheduled_departure::date AS d_date,
        dep.airport_code AS d_airport,
        arr.airport_code AS a_airport
    FROM flights f
        JOIN airports dep ON dep.airport_code = f.departure_airport
        JOIN airports arr ON arr.airport_code = f.arrival_airport
    WHERE f.scheduled_departure BETWEEN bookings.now()
      AND bookings.now() + INTERVAL '7 days';
CREATE VIEW
```

В этом представлении формат даты вылета отличается от хранимого в базе данных (в формате timestamp) времени отправления, а колонки с кодами аэропортов извлекаются из другой таблицы, соединение с которой необходимо выполнить дважды, потому что условия соединения различаются.

Когда представления используются в запросах в качестве таблиц, запросы выполняются так, как если бы данные, содержащиеся в представлениях, были бы записаны в хранимых таблицах. Однако не следует думать, что представления полностью вычисляются при выполнении каждого запроса, который их использует (хотя некоторые учебники по базам данных утверждают, что такое вычисление выполняется). Дело в том, что для вычисления результатов запроса могут быть нужны не все строки представления, а только небольшая их часть. Во многих случаях система управления базами данных в состоянии определить, какие именно строки представления необходимы для вычисления результата, и не пытается вычислить остальные.

В следующем запросе выбираются рейсы, запланированные на ближайшую неделю и направляющиеся в один аэропорт. Для вычисления результата нет необходимости рассматривать строки, в которых аэропорт прибытия отличается от указанного:

```
demo=# SELECT * FROM upcoming_flights
WHERE a_airport = 'DYR';
 flight_id | flight_no |   d_date   | d_airport | a_airport
-----------+-----------+------------+-----------+-----------
      1706 | PG0315    | 2017-08-21 | DME       | DYR
      3865 | PG0450    | 2017-08-15 | VKO       | DYR
      6489 | PG0255    | 2017-08-19 | SVO       | DYR
     26493 | PG0088    | 2017-08-19 | KHV       | DYR
(4 rows)
```

В некоторых случаях использование представлений может привести к неэффективным запросам. Например, если из представления выбираются только даты вылета и номера рейсов, то прямое использование хранимой таблицы будет значительно более эффективным, потому что соединение с таблицей аэропортов не потребуется. В системе PostgreSQL при подготовке запросов к выполнению лишние операции соединения могут быть исключены, если данные из этих таблиц не используются для формирования результата запроса. Однако такое преобразование выполняется не всегда, а многие другие системы не делают его никогда.

4.4. Структуры хранения

При создании базы данных для ее хранения должно быть выделено некоторое пространство на устройстве энергонезависимой памяти с произвольным доступом. Это пространство называется *табличным пространством* (tablespace), и обычно такое пространство выделяется в виде файлов операционной системы, однако некоторые СУБД могут работать и непосредственно с устройствами, минуя интерфейс файловой системы.

В PostgreSQL память для табличных пространств не выделяется, а запрашивается отдельно для каждого хранимого объекта. Тем не менее понятие табличного пространства используется и определяет, где именно будут храниться объекты базы данных, в терминах каталогов (директорий, папок) файловой системы.

Управление размещением баз данных является одной из обязанностей администратора данных, которые рассматриваются в главе 20 второй части курса.

Для баз данных небольшого размера обычно используют пространство, которое выделяется системой при отсутствии каких-либо указаний (т. е. используется конфигурация, предусмотренная при установке программных компонент СУБД).

Все объекты базы данных, для которых требуется память, хранятся в табличных пространствах. При создании объекта можно указать, в каком табличном пространстве следует его размещать, но пока мы не будем обсуждать эту возможность.

Подчеркнем, что табличные пространства — структура уровня хранения. Их можно использовать, например, для изменения характеристик производительности (размещать некоторые объекты на устройствах большей производительности, если эти объекты часто используются). В некоторых случаях управление размещением важно для процедур создания резервных копий. Не следует применять табличные пространства для логического структурирования базы данных.

В языке SQL отсутствует четкая граница между описанием логической структуры данных и определением схемы хранения. Во многих операторах, создающих или модифицирующих структуры данных логического уровня, например таблицы, можно указывать параметры, влияющие на ее размещение (в табличном пространстве), и некоторые особенности структуры хранения.

Это относится и к материализованным представлениям, которые, как и обычные представления, специфицируются запросом, но результат выполнения которого записывается на диск как таблица, и поэтому для таких представлений возможно указание таких же параметров хранения, как для таблиц. Кроме этого, для материализованных представлений задаются параметры, определяющие способ обновления его содержимого.

Индексы являются структурой, которая полностью относится к схеме хранения, так как по определению индексы прозрачны для приложения (и, следовательно, не могут быть видны в логической структуре БД). В системе PostgreSQL реализовано большое количество различных типов индексов и предусмотрены разнообразные средства для расширения этого набора. Эти средства детально обсуждаются во второй части курса.

Здесь мы приведем только оператор SQL, создающий наиболее распространенный тип индекса: упорядоченный индекс по значениям одной или нескольких колонок (в последнем случае применяется лексикографическое упорядочение). Такие индексы позволяют существенно ускорить выполнение фильтрации по

точному значению входящих в индекс атрибутов или по диапазону значений и могут быть полезны для некоторых других операций.

Например, индекс, ускоряющий поиск пассажиров по имени, может быть создан командой:

```
demo=# CREATE INDEX tickets_passenger_name_idx
    ON tickets(passenger_name);
CREATE INDEX
```

Некоторые индексы могут создаваться для поддержки ограничений целостности. Для уничтожения как индексов, так и других объектов используется оператор DROP.

Планирование и создание индексов требует учета большого количества разнообразных факторов, в том числе особенностей СУБД, аппаратуры, на которой работает сервер баз данных, и, главное, характера выполняемых приложением запросов. Далеко не во всех случаях индексы приводят к сокращению количества ресурсов, необходимых для выполнения запросов, а в некоторых случаях наличие индексов приводит к заметному снижению производительности системы. Отметим также, что в PostgreSQL создание индексов является транзакционной операцией и может вызвать задержки в нормальной работе системы. Выигрыш от применения индексов достигается в тех случаях, в которых время, затраченное на создание индекса, компенсируется экономией времени за счет ускорения запросов, выполняемых многократно (например, часто выполняемых запросов).

Покажем, каким образом индексы могут влиять на время отклика системы. Для этого понадобятся две ранее не упомянутые возможности.

Команда `\timing on` программы psql включает вывод времени выполнения каждого оператора:

```
demo=# \timing on
Timing is on.
```

Кроме этого, нам понадобится оператор EXPLAIN, который показывает, какие операции и в каком порядке будут использоваться при выполнении запроса. В этой части курса мы не будем подробно обсуждать всю информацию, которую можно получить с помощью EXPLAIN. Нам понадобятся только названия выполняемых операций, для того чтобы понять, чем отличаются разные варианты плана выполнения запроса.

Следующий запрос выводит загруженность самолетов, вылетающих в определенный день из Внуково в Пулково. Поскольку он потребуется несколько раз, создадим для него представление.

```
demo=# CREATE VIEW svo_led_utilization AS
    SELECT f.flight_no,
        f.scheduled_departure,
        count(tf.ticket_no) passengers
    FROM flights f
        JOIN ticket_flights tf ON tf.flight_id = f.flight_id
    WHERE f.departure_airport = 'SVO'
      AND f.arrival_airport = 'LED'
      AND f.scheduled_departure
          BETWEEN bookings.now() - INTERVAL '1 day'
              AND bookings.now()
GROUP BY f.flight_no, f.scheduled_departure;
CREATE VIEW
Time: 8,940 ms
```

Выполним запрос:

```
demo=# SELECT * FROM svo_led_utilization;
 flight_no |  scheduled_departure   | passengers
-----------+------------------------+------------
 PG0469    | 2017-08-15 12:35:00+03 |         40
 PG0470    | 2017-08-15 10:20:00+03 |        169
 PG0472    | 2017-08-14 18:30:00+03 |        121
(3 rows)
Time: 831.227 ms
```

Время выполнения составило около секунды, это очень большое время для демонстрационной базы данных. Причина состоит в том, что потребовалось выполнить полный просмотр (Seq Scan) таблиц, участвующих в запросе. Это можно увидеть, используя команду EXPLAIN (результат выполнения которой немного сокращен):

```
demo=# EXPLAIN (costs off)
SELECT * FROM svo_led_utilization;
                      QUERY PLAN
------------------------------------------------------
 GroupAggregate
   Group Key: f.flight_no, f.scheduled_departure
   -> Sort
        Sort Key: f.flight_no, f.scheduled_departure
        -> Hash Join
             Hash Cond: (tf.flight_id = f.flight_id)
             -> Seq Scan on ticket_flights tf
             -> Hash
                  -> Seq Scan on flights f
                       Filter: ...
```

Попробуем построить индексы, которые могли бы ускорить выполнение этого запроса. Поскольку в запросе есть условие на точное значение аэропорта отправления, построим индекс для этого атрибута.

```
demo=# CREATE INDEX flights_departure_airport_idx
    ON flights(departure_airport);
CREATE INDEX
Time: 247.384 ms
```

Можно убедиться в том, что новый план выполнения того же запроса использует только что созданный индекс, как видно на рис. 4.4.1.

Тем не менее время выполнения запроса почти не изменилось:

```
demo=# SELECT * FROM svo_led_utilization;
 flight_no |  scheduled_departure   | passengers
-----------+-----------------------+------------
 PG0469    | 2017-08-15 12:35:00+03 |         40
 PG0470    | 2017-08-15 10:20:00+03 |        169
 PG0472    | 2017-08-14 18:30:00+03 |        121
(3 rows)
Time: 637.549 ms
```

Это можно объяснить тем, что индекс построен на таблице рейсов, которая по размеру значительно меньше, чем другая таблица, используемая в запросе. Фактически для выполнения запроса нужны только те строки таблицы ticket_flights, которые связаны с рейсами, удовлетворяющими критериям фильтрации. Поэтому следует попробовать создать еще один индекс на атрибут flight_id этой таблицы, который является внешним ключом, связывающим с таблицей flights. Важно, что именно этот атрибут использован в условии операции соединения.

```
demo=# CREATE INDEX ticket_flights_flight_id_idx
    ON ticket_flights(flight_id);
CREATE INDEX
Time: 3718.205 ms
```

Попробуем выполнить тот же самый запрос еще раз:

```
demo=# SELECT * FROM svo_led_utilization;
 flight_no |  scheduled_departure   | passengers
-----------+-----------------------+------------
 PG0469    | 2017-08-15 12:35:00+03 |         40
 PG0470    | 2017-08-15 10:20:00+03 |        169
 PG0472    | 2017-08-14 18:30:00+03 |        121
(3 rows)
Time: 4.268 ms
```

```
demo=# EXPLAIN (costs off)
SELECT *
FROM svo_led_utilization;
                                  QUERY PLAN
------------------------------------------------------------------------------
 GroupAggregate
   Group Key: f.flight_no, f.scheduled_departure
   -> Sort
        Sort Key: f.flight_no, f.scheduled_departure
        -> Hash Join
             Hash Cond: (tf.flight_id = f.flight_id)
             -> Seq Scan on ticket_flights tf
             -> Hash
                  -> Bitmap Heap Scan on flights f
                       Recheck Cond: (departure_airport = 'SVO'::bpchar)
                       Filter: ...
                       -> Bitmap Index Scan on flights_departure_airport_idx
                            Index Cond: (departure_airport = 'SVO'::bpchar)
```

Рис. 4.4.1. План выполнения запроса после создания индекса

Время выполнения сократилось примерно в 200 раз. На базе данных большего размера это соотношение было бы еще более значительным. Причина ускорения состоит в том, что теперь при выполнении запроса используются оба индекса, как видно на рис. 4.4.2.

Ускорение запроса на два порядка выглядит неплохо, но намного важнее то, что при увеличении размера базы данных время выполнения запроса без использования индексов будет расти линейно (пропорционально размеру большей таблицы), а при использовании индексов — пропорционально логарифму размера той же таблицы (но пропорционально количеству строк в результате выполнения запроса).

База данных, на которой выполнялись эти измерения, была размещена на устройстве SSD. Если бы вместо этого использовались вращающиеся диски, то различие во времени выполнения было бы еще более значительным. Заметим, что при выполнении того же запроса на другом компьютере абсолютные значения времени могут получиться совсем другими, однако важно не абсолютное время выполнения, а соотношение времени выполнения разных планов.

Подчеркнем, что задача выбора индексов далеко не так проста, как может показаться на основе этого примера, а выбор индексов — только один из инструментов, которые можно использовать для управления производительностью системы и эффективностью выполнения отдельных запросов.

4.5. Логическая организация данных

Логически взаимосвязанные объекты базы данных группируются в *схему*. Понятие схемы является чисто логическим, оно никак не связано с размещением или структурами, используемыми для хранения данных. Например, схема может содержать объекты, необходимые для работы одного приложения, но любое приложение может работать с объектами, находящимися в разных схемах. Внутри схемы имена объектов должны быть уникальны, полное имя объекта включает имя схемы и идентификатор (имя) объекта, разделенные точкой.

Схемы можно использовать для разграничения доступа, как описано в главе 5. Обычно в базе данных PostgreSQL существует схема public, доступ к которой имеют все пользователи, а доступ к другим схемам зависит от предоставленных привилегий.

```
demo=# EXPLAIN (costs off)
SELECT *
FROM svo_led_utilization;
                                QUERY PLAN
------------------------------------------------------------------------
 GroupAggregate
   Group Key: f.flight_no, f.scheduled_departure
   -> Sort
        Sort Key: f.flight_no, f.scheduled_departure
        -> Nested Loop
             -> Bitmap Heap Scan on flights f
                  Recheck Cond: (departure_airport = 'SVO'::bpchar)
                  Filter: ...
                  -> Bitmap Index Scan on flights_dep_airport_idx
                       Index Cond: (departure_airport = 'SVO'::bpchar)
             -> Bitmap Heap Scan on ticket_flights tf
                  Recheck Cond: (flight_id = f.flight_id)
                  -> Bitmap Index Scan on ticket_flights_flight_id_idx
                       Index Cond: (flight_id = f.flight_id)
```

Рис. 4.4.2. План выполнения запроса с двумя индексами

Для создания схемы используют оператор CREATE SCHEMA. Например, схема для демонстрационного примера создается оператором:

```
CREATE SCHEMA bookings;
```

Если при обращении к объекту схема не указана, то для поиска объекта используется значение переменной search_path. Значение этой переменной можно установить командой SET, а получить текущее значение — командой SHOW.

```
demo=# SHOW search_path;
   search_path
-----------------
 "$user", public
(1 row)
```

Такое определение пути поиска дает возможность не указывать имя схемы для объектов, размещенных в схеме текущего пользователя (если она существует) и в схеме public. При подключении к демобазе значение этой конфигурационной переменной автоматически изменяется так, чтобы не требовалось указывать имя схемы bookings для демонстрационных таблиц. Аналогичное изменение можно выполнить и вручную командой:

```
demo=# SET search_path TO bookings,"$user",public;
SET
```

При такой установке пути поиска следующие два оператора будут эквивалентны, потому что таблица airports находится в схеме bookings:

```
SELECT * FROM bookings.airports;
SELECT * FROM airports;
```

Этот же способ поиска применяется и для представлений, функций, типов и других видов объектов базы данных. Более того, такой же способ применяется и для системных объектов PostgreSQL.

Поскольку имя объекта должно быть уникально только в пределах схемы, возможно создание одинаково названных объектов в разных схемах. Поэтому выполнение запросов, в которых явно не указано имя схемы, будет зависеть от текущего значения пути поиска. Это, с одной стороны, дает возможность использовать идентичные запросы для обработки различных наборов данных за счет размещения их в разных схемах, с другой — создает условия для возникновения ошибок. Поэтому при проектировании базы данных и приложений необходимо установить и соблюдать правила, регламентирующие использование имен схем и размещенных в них объектов.

4.6. Итоги главы

В этой главе показаны основные конструкции языка SQL, способы выражения базовых операций реляционной алгебры в декларативном стиле, а также способы конструирования сложных декларативных запросов. На основе этого материала можно программировать значительную долю запросов, необходимость которых возникает в практических приложениях, использующих СУБД.

Не затрагивались более сложные конструкции SQL, процедурные средства программирования баз данных, средства расширения функциональности СУБД, поддержка расширенной системы типов данных и вопросы проектирования.

Более детальное изложение языка SQL можно найти в учебнике [65]. Описание основных конструкций SQL, ориентированное на систему PostgreSQL, содержится в [67]. Индексные структуры и их влияние на время выполнения запросов подробно обсуждаются в книге [63].

4.7. Упражнения

Предполагается, что все упражнения к этой главе выполняются над демонстрационной базой данных, с которой вы уже познакомились в главе 3.

Основные конструкции и синтаксис

Упражнение 4.1. Запустите клиент psql с демонстрационной базой данных и попробуйте выполнить запрос

```
SELECT 0
```

Что произошло? Какого синтаксического элемента не хватает?

Упражнение 4.2. Выберите все модели самолетов и соответствующие им диапазоны дальности полетов.

Упражнение 4.3. Найдите все самолеты с максимальной дальностью полета:

1) либо больше 10 000 км, либо меньше 4 000 км;
2) больше 6 000 км, а название не заканчивается на «100».

Обратите внимание на порядок следования предложений WHERE и FROM.

Упражнение 4.4. Определите номера и время отправления всех рейсов, прибывших в аэропорт назначения не вовремя.

Упражнение 4.5. Подсчитайте количество отмененных рейсов из аэропорта Пулково (LED), как вылет, так и прибытие которых было назначено на четверг.

Соединения

Упражнение 4.6. Выведите имена пассажиров, купивших билеты эконом-класса за сумму, превышающую 70 000 рублей.

Упражнение 4.7. Напечатанный посадочный талон должен содержать фамилию и имя пассажира, коды аэропортов вылета и прилета, дату и время вылета и прилета по расписанию, номер места в салоне самолета. Напишите запрос, выводящий всю необходимую информацию для полученных посадочных талонов на рейсы, которые еще не вылетели.

Упражнение 4.8. Некоторые пассажиры, вылетающие сегодняшним рейсом («сегодня» определяется функцией bookings.now), еще не прошли регистрацию, т. е. не получили посадочного талона. Выведите имена этих пассажиров и номера рейсов.

Упражнение 4.9. Выведите номера мест, оставшихся свободными в рейсах из Анапы (AAQ) в Шереметьево (SVO), вместе с номером рейса и его датой.

Агрегирование и группировка

Упражнение 4.10. Напишите запрос, возвращающий среднюю стоимость авиабилета из Воронежа (VOZ) в Санкт-Петербург (LED). Поэкспериментируйте с другими агрегирующими функциями (sum, max). Какие еще агрегирующие функции бывают?

Упражнение 4.11. Напишите запрос, возвращающий среднюю стоимость авиабилета в каждом из классов перевозки. Модифицируйте его таким образом, чтобы было видно, какому классу какое значение соответствует.

Упражнение 4.12. Выведите все модели самолетов вместе с общим количеством мест в салоне.

Упражнение 4.13. Напишите запрос, возвращающий список аэропортов, в которых было принято более 500 рейсов.

Модификация данных

Упражнение 4.14. Авиакомпания провела модернизацию салонов всех имеющихся самолетов «Сессна» (код CN1), в результате которой был добавлен седьмой ряд кресел. Измените соответствующую таблицу, чтобы отразить этот факт.

Упражнение 4.15. В результате еще одной модернизации в самолетах «Аэробус A319» (код 319) ряды кресел с шестого по восьмой были переведены в разряд бизнес-класса. Измените таблицу одним запросом и получите измененные данные с помощью предложения RETURNING.

Упражнение 4.16. Создайте новое бронирование текущей датой. В качестве номера бронирования можно взять любую последовательность из шести символов, начинающуюся на символ подчеркивания. Общая сумма должна составлять 30 000 рублей.

Создайте электронный билет, связанный с бронированием, на ваше имя.

Назначьте электронному билету два рейса: один из Москвы (VKO) во Владивосток (VVO) через неделю, другой — обратно через две недели. Оба рейса выполняются эконом-классом, стоимость каждого должна составлять 15 000 рублей.

Описание данных: отношения

Упражнение 4.17. Авиакомпания хочет предоставить пассажирам возможность повышения класса обслуживания уже после покупки билета при регистрации на рейс. За это взимается отдельная плата. Добавьте в демонстрационную базу данных возможность хранения таких операций.

Упражнение 4.18. Авиакомпания начинает выдавать пассажирам карточки постоянных клиентов. Вместо того чтобы каждый раз вводить имя, номер документа и контактную информацию, постоянный клиент может указать номер своей карты, к которой привязана вся необходимая информация. При этом клиенту может предоставляться скидка.

Измените существующую схему данных так, чтобы иметь возможность хранить информацию о постоянных клиентах.

Упражнение 4.19. Постоянные клиенты могут бесплатно провозить с собой животных. Добавьте в ранее созданную таблицу постоянных клиентов информацию о перевозке домашних животных.

Вложенные подзапросы

Упражнение 4.20. Найдите модели самолетов «дальнего следования», максимальная продолжительность рейсов которых составила более 6 часов.

Упражнение 4.21. Подсчитайте количество рейсов, которые хотя бы раз были задержаны более чем на 4 часа.

Упражнение 4.22. Для составления рейтинга аэропортов учитывается суточная пропускная способность, т. е. среднее количество вылетевших из него и прилетевших в него за сутки пассажиров. Выведите 10 аэропортов с наибольшей суточной пропускной способностью, упорядоченных по убыванию данной величины.

Псевдонимы для таблиц

Упражнение 4.23. С целью оценки интенсивности работы обслуживающего персонала аэропорта Шереметьево (SVO) вычислите, сколько раз вылеты следовали друг за другом с перерывом менее пяти минут.

Представления

Упражнение 4.24. Количество рейсов, принятых конкретным аэропортом за каждый день, — довольно востребованный запрос. Напишите представление данного запроса для аэропорта города Барнаул (BAX).

Глава 5

Управление доступом в базах данных

5.1. Модели защиты и разграничения доступа

Обеспечение защиты данных от несанкционированного использования с самого начала рассматривалось как одно из основных требований к системам управления базами данных, однако вопросы защиты и разграничения доступа к функциям системы важны практически в любой информационной системе. Поэтому соответствующие понятия и модели можно рассматривать в более широком контексте.

Построение любой модели защиты начинается с понятия *принципала* (английский термин — principal). Принципал обладает очень большими полномочиями (например, может принимать на работу или увольнять). Для нас, однако, важно только то, что принципал имеет право разрешать доступ к информационной системе другим лицам, которые в результате становятся *пользователями* этой информационной системы.

Только пользователи имеют возможность работы с системой. Для того чтобы разграничить возможности разных пользователей, вводятся понятия *объекта* и *действия*. Действия могут быть как связаны с объектом (например, методы объекта) или с классом объектов, так и не связаны. На этом уровне абстракции связь между действиями и объектами не имеет значения.

Обычно от имени принципала действует администратор, который для доступа к программной системе (например, операционной системе или системе управления базами данных) использует особый идентификатор пользователя, который называется *суперпользователем*. Как правило, суперпользователь создается при установке программной системы. В дальнейшем нам не понадобится различать понятия принципала, администратора, действующего от его имени, и суперпользователя, хотя эти понятия не совпадают. Например, суперпользователь операционной системы не обязательно совпадает с суперпользователем базы данных, хотя управление и СУБД, и операционной системой может быть поручено одному человеку.

Суперпользователь может выполнять любые действия и имеет неограниченный доступ ко всем объектам. Любой другой пользователь может выполнять только те действия и только над теми объектами, на которые ему предоставлено право.

Особое значение имеет действие, в результате которого возникает новый объект. Конечно, суперпользователь имеет право создания любых объектов, в том числе регистрировать новых пользователей. Кроме этого, он может предоставить право создания объектов (возможно, ограниченного класса) другим пользователям. Любой пользователь, создавший объект, становится его *владельцем*. Владелец любого объекта имеет право выполнять любые действия, связанные с этим объектом, и может предоставлять (возможно, ограниченные) права доступа к своим объектам и действиям над ними другим пользователям. Права доступа к объектам и использования действий называются *привилегиями*.

Для того чтобы упростить управление передачей привилегий пользователям, вводится понятие *роли*. Каждой роли передаются привилегии, необходимые для выполнения всех операций, связанных с этой ролью. Например, в системе интернет-магазина могут быть роли посетителя, покупателя, продавца и администратора.

Каждый пользователь получает право (привилегию) выполнять некоторую роль или несколько ролей. Кроме этого, конечно, пользователю могут передаваться и отдельные привилегии, не обязательно предоставленные какой-либо роли. Подобные модели защиты и разграничения доступа называют моделями, основанными на ролях (RBAC, Role Based Access Control). Кроме отдельных привилегий, роли могут получать другие роли в качестве привилегий. Передача роли в качестве привилегии эквивалентна передаче всех привилегий, имеющихся у этой роли.

Кроме RBAC, существуют другие модели, в частности основанные на значениях атрибутов объектов (ABAC, Attribute Based Access Control). Подобные модели необходимы, например, для того, чтобы пассажир мог получить доступ к своим бронированиям (в демонстрационной базе данных), но не к бронированиям других пассажиров. В этой главе мы не будем обсуждать механизмы системы PostgreSQL, необходимые для реализации разграничения доступа такого типа.

В сложных информационных системах применяется многоуровневая схема разграничения и контроля доступа: контроль может выполняться на уровне вычислительной сети, операционной системы, приложения и системы управления базами данных. При этом на разных уровнях обычно используются разные модели и, как правило, различные понятия пользователя.

На любом уровне работа всегда происходит от имени какого-нибудь пользователя, и при установлении соединения система должна его *аутентифицировать,* т. е. проверить, является ли пользователь тем, за кого себя выдает. Наиболее широко известна аутентификация при помощи имени пользователя и пароля, однако существуют и более сложные схемы, обеспечивающие более высокую степень защиты.

Обычно принято считать, что конфигурация средств контроля доступа относится к компетенции администратора баз данных, а не пользователей готовых приложений или разработчиков. Тем не менее как пользователям, так и разработчикам приложений полезно иметь представление о моделях разграничения доступа, которые могут использоваться на уровне базы данных. В этой главе приводятся основные сведения о средствах и моделях разграничения доступа. Дополнительная информация содержится в главе 19 второй части курса.

5.2. Пользователи и роли в СУБД

Посмотрим, каким образом модели разграничения доступа реализуются в системах управления базами данных.

В системе PostgreSQL определены понятия *роли* и *привилегии* Для каждой роли задается набор *атрибутов*, определяющих ее свойства, и предоставляются привилегии для работы с объектами базы данных.

Роли применяются для представления различных функций:

Набор привилегий, которые часто должны предоставляться вместе, например, необходимы для использования какого-либо приложения. Такие роли часто бывают связаны с группами пользователей, которым необходимы одинаковые права доступа к данным.

Пользователь базы данных. Если роль имеет атрибут LOGIN, от ее имени можно создавать сеансы работы с сервером баз данных.

Важная особенность системы PostgreSQL состоит в том, что роли создаются на уровне кластера баз данных, а возможности работы от имени роли (пользователя) с каждой из баз, входящей в кластер, определяются соответствующими привилегиями. Атрибуты роли задают важные свойства роли, в том числе возможность управления объектами, которые определены на уровне кластера: в частности, ролями и базами данных.

В ранних версиях PostgreSQL использовались понятия пользователя и группы, однако, начиная с версии 8.1, осталось только понятие роли, обобщающее эти два понятия.

Помимо аутентификации, СУБД обеспечивает разграничение доступа: каждой роли позволяется работать только с теми объектами данных и выполнять только те действия, на выполнение которых этой роли предоставлены привилегии. Например, объектами являются таблицы и хранимые процедуры. Пример действий — чтение, добавление, модификация или удаление данных. Перед выполнением любого действия СУБД выполняет *авторизацию*, т. е. проверку права роли на выполнение этого действия. Соединение с базой данных также является операцией, требующей авторизации.

Системы управления базами данных (в том числе и PostgreSQL) поддерживают понятие суперпользователя. Суперпользователь может совершать любые действия — ограничения доступа на него не распространяются. Наличие прав суперпользователя определяется атрибутом роли.

Создание, модификация и удаление ролей и пользователей выполняется с помощью операторов SQL. Оператор CREATE ROLE создает, ALTER ROLE изменяет, а оператор DROP ROLE — уничтожает роль. Во многих системах имеются аналогичные операторы для регистрации пользователей: CREATE USER и т. д. В системе PostgreSQL такие операторы тоже можно использовать, однако они, по существу, не отличаются от операторов для ролей.

В качестве действий, выполнение которых регулируется привилегиями, обычно используются операторы SQL. Так, существуют привилегии на создание объектов (CREATE), вставку, изменение и удаление (INSERT, UPDATE, DELETE соответственно). Та роль (пользователь), от имени которой был создан некоторый объект, становится владельцем этого объекта и имеет все права на доступ, модификацию и уничтожение этого объекта. Владелец может передавать права на доступ к своим объектам другим ролям (и пользователям).

Для некоторых типов объектов некоторые из действий не имеют смысла; в таких случаях, естественно, невозможно предоставление соответствующих привилегий (операция DELETE не имеет смысла для последовательностей). Некоторые привилегии не связаны непосредственно с операторами SQL (для хранимых функций существует привилегия EXECUTE на выполнение этих функций).

Например, оператор

```
demo=# CREATE ROLE reader;
CREATE ROLE
```

создает роль reader (не предоставляя этой роли никаких привилегий). Свойства ролей можно получить с помощью команды \du программы psql:

```
demo=# \du reader
            List of roles
 Role name |  Attributes  | Member of
-----------+--------------+-----------
 reader    | Cannot login | {}
```

Для того чтобы роль reader могла использоваться для создания сеансов, можно выполнить команду

```
demo=# ALTER ROLE reader LOGIN;
ALTER ROLE
```

Конечно, атрибут LOGIN можно указывать и при создании роли.

5.3. Объекты и привилегии

Перечень возможных привилегий не определен в стандарте SQL, поэтому в разных СУБД возможные виды привилегий могут различаться. Среди привилегий, которые могут быть предоставлены ролям, есть привилегии на создание объектов базы данных. В системе PostgreSQL имеются привилегии на создание схемы и на создание объектов внутри схемы. В других системах можно обнаружить отдельные привилегии для создания объектов каждого типа (таблиц, представлений и т. д.).

Пользователь, создавший объект, становится его владельцем. Владелец объектов базы данных имеет право выполнять любые операции над этими объектами и может предоставлять некоторые привилегии для работы с ними другим пользователям.

Хотя обычно владельцем объекта становится именно тот пользователь, который его создал, можно заменять владельца уже существующего объекта. Такая возможность важна, для того чтобы создавать объекты, владельцами которых будут пользователи, не имеющие права создавать объекты такого типа.

Привилегии предоставляются оператором GRANT, простейшая форма которого имеет вид:

```
GRANT привилегия ON объект_данных TO пользователь;
```

При этом можно указывать предложение WITH GRANT OPTION, чтобы получатель привилегии мог передавать эту привилегию другим пользователям. Любой пользователь, которому привилегия была предоставлена с правом передачи, может предоставлять эту привилегию так же, как привилегии на объекты, которыми он владеет.

Перечень привилегий, которые могут быть предоставлены для работы с некоторым объектом базы данных, зависит от типа этого объекта и, как правило, соответствует набору операторов, которые можно выполнять над этим объектом. Например привилегии для таблиц включают SELECT, UPDATE, INSERT, DELETE, TRUNCATE, REFERENCES и TRIGGER. Последние две привилегии позволяют создавать ограничения ссылочной целостности (FOREIGN KEY) и триггеры соответственно.

В качестве списка привилегий в операторе GRANT можно также указывать ALL PRIVILEGES: в этом случае передаются все привилегии, определенные для этого типа объекта, которые имеет пользователь, от имени кого выполняется оператор GRANT.

Если в качестве роли в операторе GRANT указано public, то привилегии предоставляются всем пользователям (ролям) — как уже существующим, так и тем, которые, возможно, будут созданы в будущем.

Например, оператор

```
demo=# GRANT SELECT ON TABLE airports TO public;
GRANT
```

предоставляет право выборки данных из таблицы (или представления) airports всем пользователям (ролям) сервера базы данных.

Если при конфигурации базы данных не определено иначе, то все объекты, определенные в схеме public, становятся доступными для роли public.

Если объект базы данных является таблицей или представлением, то можно указать список колонок, на которые распространяются предоставляемые привилегии.

Предоставленные пользователю привилегии могут быть отозваны с помощью оператора

```
REVOKE привилегия ON объект_данных FROM пользователь;
```

Операторы GRANT и REVOKE позволяют предоставлять привилегии сразу на все объекты некоторого типа.

Например, оператор

```
demo=# GRANT SELECT ON ALL TABLES IN SCHEMA bookings TO reader;
GRANT
```

предоставляет привилегию чтения данных из всех таблиц схемы bookings пользователю (роли) reader. Заметим, однако, что доступ не будет предоставлен для таблиц, созданных в этой схеме после выполнения данной команды.

Некоторые привилегии не связаны с конкретным объектом; например, такими привилегиями являются роли. Для них указание объекта в операторах GRANT и REVOKE не требуется. Так, оператор

```
demo=# GRANT reader TO writer;
GRANT
```

предоставит роли writer все привилегии роли reader (как уже имеющиеся, так и те, которые могут быть ей предоставлены в будущем). Заметим, что в этой форме оператора GRANT нельзя использовать public в качестве получателя привилегий, т. е. роли не могут быть сделаны общедоступными.

5.4. Итоги главы

В этой главе рассмотрены основные понятия модели разграничения доступа на основе ролей (Role Based Access Control, RBAC) и показано, каким образом эта модель реализуется в системе привилегий PostgreSQL.

5.5. Упражнения

При выполнении упражнений этой главы особенно важно понимать, от имени какого пользователя СУБД выполняются команды. Текущего пользователя можно узнать командой

```
SELECT session_user;
```

Чтобы ориентироваться было проще, можно изменить приглашение psql таким образом, чтобы оно включало не только имя текущей базы данных, но и имя пользователя. Это выполняется следующими командами (которые могут быть помещены в файл ~/.psqlrc, чтобы их не приходилось вводить каждый раз заново):

```
demo=# \set PROMPT1 %n@%/%R%#
student@demo=# \set PROMPT2 :PROMPT1
```

Напомним, что переключение на другого пользователя в программе psql выполняется командой \c.

Упражнение 5.1. Создайте роль для доступа на чтение к демонстрационной базе данных без права создания сеансов работы с сервером БД.

Упражнение 5.2. Создайте пользователя сервера БД и предоставьте ему привилегию использования роли, созданной в предыдущем упражнении. Проверьте, что этот пользователь может выполнять любые запросы на выборку из таблиц демонстрационной базы данных, но не может их обновлять.

Упражнение 5.3. Заберите у пользователя привилегию, выданную в предыдущем упражнении. Убедитесь, что этот пользователь не сможет выбирать данные из таблиц демобазы.

Упражнение 5.4. Постройте пример, показывающий, что для доступа к таблицам схемы необходимо также предоставить право использования (USAGE) этой схемы.

Глава 6

Транзакции и согласованность базы данных

Средства управления транзакциями в системах управления базами данных выполняют функции, связанные с реализацией согласованности, т. е. обеспечивают корректное состояние данных. Во время нормальной работы за поддержку согласованности отвечает диспетчер транзакций, работающий на основе некоторого протокола. Каждый протокол гарантирует выполнение некоторого критерия корректности. Выбор протокола и, следовательно, поведение диспетчера зависят от требований приложения и от конфигурации системы.

Отметим, что использование диспетчеров и протоколов необходимо для обеспечения корректности (согласованности) только при одновременной работе с базой данных нескольких сеансов. При этом не имеет значения, от имени каких пользователей и какие приложения используют эти сеансы, важно только то, что операции с базой данных выполняются в разных сеансах независимо друг от друга. Управление транзакциями часто связывают с организацией многопользовательского доступа к базе данных, хотя в этом контексте не имеет значения, созданы сеансы от имени одного пользователя или от имени разных.

Другая, не менее важная функция средств управления транзакциями — восстановление корректного состояния после отказов (обеспечение отказоустойчивости): после любых отказов база данных должна быть приведена в корректное состояние. Важно отметить, что для многих классов современных приложений отказоустойчивость баз данных значительно важнее, чем согласованность. Поэтому довольно часто используются протоколы, обеспечивающие соблюдение только ослабленных критериев корректности, но гарантирующие максимально возможное сохранение результатов работы приложений.

В этой главе обсуждаются требования к средствам поддержки согласованности, возможные последствия неконтролируемого выполнения, а также управление транзакциями в приложениях. Теоретические модели, протоколы, алгоритмы, а также особенности реализации в PostgreSQL рассматриваются в главе 13 второй части книги.

6.1. Определение и основные требования к транзакциям

Транзакцией называется конечное множество операций над базой данных, выполняемое приложением, которое переводит базу данных из одного согласованного состояния в другое согласованное состояние, при условии что транзакция выполнена полностью и без помех со стороны других приложений. В этой главе мы будем предполагать, что множество операций, составляющее транзакцию, упорядочено.

Прежде всего необходимо еще раз подчеркнуть различие между понятиями *целостности* (integrity) и *согласованности* (consistency): целостность определяется с помощью ограничений целостности, и СУБД предотвращает любые попытки нарушения этих ограничений. В то же время согласованность может временно нарушаться и восстанавливается только в конце выполнения транзакции.

Неформально требования к транзакционным системам принято описывать в терминах свойств ACID — по первым буквам английских названий: *атомарность* (atomicity), *согласованность* (consistency), *изоляция* (isolation), *долговечность* (durability). Все эти свойства требуют дополнительных пояснений.

Атомарность означает, что любая транзакция должна быть либо выполнена полностью, либо, если по каким-либо причинам завершение транзакции невозможно, ее частичное выполнение не должно оставлять никаких следов ни в базе данных, ни в результатах работы приложений. Это требование накладывает ограничения как на СУБД (которая должна устранять последствия неполного выполнения транзакций), так и на приложения, которые не должны выводить какие-либо результаты, зависящие от не полностью выполненных транзакций.

Требование **согласованности** включено в наше определение транзакций. Еще раз отметим, что согласованность определяется семантикой приложения и не может быть в полной мере проверена СУБД.

Требование **изоляции** означает, что СУБД должна обеспечить выполнение без помех со стороны других транзакций — ограничение, входящее в определение транзакции. Нарушение изоляции транзакций может приводить к появлению некорректных результатов и состояний базы данных. Подобные ситуации называются аномалиями конкурентного выполнения и обсуждаются ниже в этой

главе. Для предотвращения всех возможных аномалий необходима полная изоляция транзакций, однако требование изоляции может вступать в противоречие с требованием высокой пропускной способности, поэтому довольно часто используются ослабленные условия изоляции.

Свойство **долговечности** предъявляет очень сильные и трудно реализуемые требования к СУБД. Оно означает, что никакие изменения, выполненные завершенными транзакциями, не могут быть потеряны, что бы ни происходило с сервером базы данных или вычислительной системой, на которой этот сервер работает. Таким образом, это требование, по существу, определяет отказоустойчивость базы данных. Конечно, это требование вовсе не означает, что данные, записанные транзакцией, не могут быть изменены другими транзакциями. Возможно, следующая транзакция изменит их через миллисекунды, однако она в своей работе будет учитывать результаты работы завершенных транзакций.

Необходимо также отметить, что современные технологии позволяют обеспечить любую степень отказоустойчивости базы данных, однако это может привести к существенному увеличению стоимости системы как на этапе разработки, так и во время эксплуатации. Выбор уровня защищенности от отказов обычно требует компромиссов, учитывающих реальные требования прикладной системы.

Из свойства атомарности следует, что любая транзакция может завершиться одним из двух способов.

- Нормальное завершение транзакции называется *фиксацией* (commit). Операция фиксации выполняется приложением для того, чтобы сообщить СУБД, что все операции транзакции выполнены.
- Невозможность полного выполнения приводит к необходимости *обрыва* транзакции (abort).

Обрыв может выполняться как по инициативе приложения, так и по инициативе СУБД. Способ реализации обрыва зависит от внутренней организации СУБД, однако в большинстве систем операции выполняются в предположении, что транзакция будет зафиксирована, а в случае обрыва внесенные изменения устраняются из базы данных. Операция устранения изменений называется *откатом* (rollback).

6.2. Аномалии конкурентного выполнения

В соответствии с определением транзакций существует просто реализуемый способ обеспечения согласованности: строго последовательное выполнение транзакций. Очевидно, при этом каждая транзакция выполняется без помех со стороны других транзакций. Единственное, что все-таки нужно сделать в этом случае, — это гарантировать откат транзакций, которые не могут завершиться полностью.

Такой метод выполнения транзакций, однако, привел бы к очень большим задержкам при работе системы, поскольку почти полностью исключил бы возможность одновременного выполнения каких-либо операций. Высокая производительность систем управления базами данных достигается, кроме всего прочего, тем, что большое количество транзакций выполняется одновременно.

Заметим, что для одновременного выполнения транзакций вовсе не обязательно, чтобы какие-либо компоненты СУБД или вычислительной системы работали параллельно. Чередование операций разных транзакций возможно даже в очень простой системе, которая выполняет все операции в рамках одного процесса, поэтому управление транзакциями оказывается необходимым. Такая организация работы дает возможность завершать короткие транзакции быстрее, чем при строго последовательном выполнении, поскольку их операции будут выполняться между операциями других, возможно, более длинных транзакций. Мы будем поэтому говорить не о параллельном, а о конкурентном выполнении транзакций, объединяя таким образом чередование операций разных транзакций и при последовательном или псевдопараллельном (многопроцессном) выполнении операций, и при действительно параллельном выполнении на соответствующих конфигурациях оборудования.

Ситуации, при которых конкурентное выполнение корректных транзакций приводит к некорректным результатам, принято называть *аномалиями*. Для иллюстрации аномалий достаточно предполагать, что операции выполняются последовательно (однако, конечно, операции разных транзакций чередуются). Кроме этого, достаточно использовать только простые операции чтения и записи, обрабатывающие один элемент данных каждая. Сейчас для нас не имеет значения, что представляет собой элемент данных: это может быть физический блок на устройстве хранения, содержащий страницу базы данных, строка таблицы, пара ключ — значение или какой-нибудь еще объект в базе данных. Важно только то, что разные элементы данных не имеют общих частей, т. е.

изменение значения одного элемента никак не влияет на значения других элементов. Для того чтобы облегчить восприятие, при чтении этой главы можно считать, что элементы данных — это строки таблиц. Операция чтения элемента x, выполняемая в транзакции i, обозначается $r_i(x)$, операция записи — $w_i(x)$, а сама транзакция обозначается t_i.

Запись последовательности выполнения всех операций нескольких транзакций называется *историей*, а любой начальный отрезок истории называется *расписанием*.

Рассмотрим простую транзакцию, которая читает числовое значение из базы данных, уменьшает его на 100 и записывает обратно. Если такая транзакция выполняется дважды для разных пользователей, но с одним и тем же числовым значением, в результате это значение, очевидно, уменьшится на 200. Корректное (последовательное) выполнение этих двух транзакций может быть представлено следующим расписанием:

$$r_1(x)\ w_1(x)\ r_2(x)\ w_2(x).$$

Однако если операции двух транзакций выполняются в следующем порядке:

$$r_1(x)\ r_2(x)\ w_1(x)\ w_2(x),$$

то в результате будет записано значение, учитывающее только работу последней записавшей транзакции из двух, выполнявших изменение, т. е. значение x будет уменьшено только на 100, потому что вторая транзакция читает x до того, как первая записывает результаты своей работы. Такая аномалия называется **потерянным обновлением**.

Условие согласованности может связывать значения различных элементов данных. Например, предположим, что сумма значений x и y должна оставаться постоянной в любом согласованном состоянии базы данных. Пусть первая транзакция считывает оба элемента данных, затем вычитает ненулевое значение из x и прибавляет его к y, а вторая транзакция читает оба этих объекта. Тогда следующая последовательность выполнения операций:

$$r_1(x)\ r_1(y)\ r_2(x)\ w_1(x)\ w_1(y)\ r_2(y)$$

приводит к некорректному результату: вторая транзакция считывает несогласованные значения объектов x и y. Аномалии такого типа называются аномалиями **несогласованного чтения**.

Еще один вид аномалий связан с операциями завершения транзакций, т. е. фиксации (*c*) и обрыва (*a*). Рассмотрим следующую последовательность:

$$r_1(x)\ w_1(x)\ r_2(x)\ a_1\ c_2.$$

Поскольку первая транзакция обрывается, ее результаты не должны оставлять следов, однако они уже были прочитаны второй транзакцией. Такая ситуация называется аномалией **грязного чтения**.

Заметим, что устранить эту аномалию можно двумя способами: задержать выполнение операции второй транзакции до завершения первой (неважно, фиксации или обрыва) или оборвать вторую транзакцию, когда обрывается первая. Такие обрывы, которые могут быть нужны для сохранения согласованности, называются *каскадными обрывами*.

Известно еще несколько видов аномалий конкурентного выполнения.

Грязная запись — вторая транзакция записывает новое значение до фиксации первой: $w_1(x)\ w_2(x)\ c_1$.

Нечеткое (неповторяющееся) чтение — повторное чтение элемента данных дает другой результат, поскольку значение элемента было изменено другой транзакцией: $r_1(x)\ w_2(x)\ c_2\ r_1(x)$.

Фантомное чтение — повторный поиск данных по предикату возвращает результат, отличающийся от первого, потому что другая транзакция добавила, удалила или изменила записи таким образом, что они стали удовлетворять (или перестали удовлетворять) предикату.

Несогласованная запись — в отличие от последовательного выполнения транзакций новые значения *x* и *y* не учитывают значений, записанных другой транзакцией: $r_1(x)\ r_2(y)\ w_1(y)\ w_2(x)\ c_1\ c_2$.

Аномалия только читающей транзакции — в приведенном ниже расписании t_3 возвращает некорректные значения, т. к. учитывает изменение элемента *y* первой транзакцией, но не учитывает изменения *x* второй транзакцией, хотя t_2 использует значение, записанное до выполнения t_1:

$$r_2(x)\ r_2(y)\ r_1(y)\ w_1(y)\ c_1\ r_3(x)\ r_3(y)\ c_3\ w_2(x)\ c_2.$$

В дальнейшем нам понадобятся различные критерии корректности. Идеальная корректность расписания достигается, если все результаты работы всех операций всех транзакций — как записываемые в базу данных, так и передаваемые

в приложение — совпадают с результатами некоторого последовательного выполнения тех же транзакций. Расписание, выполнение которого эквивалентно последовательному, называется *сериализуемым*. Заметим, что разные последовательные расписания не обязательно эквивалентны между собой, т. е. могут давать разные результаты, но любое из них будет корректным. Более слабые критерии корректности определяются в терминах запрещаемых аномалий: чем больше аномалий предотвращается, тем выше степень корректности.

6.3. Восстановимость

Требование отказоустойчивости накладывает дополнительные ограничения на то, в каком порядке могут выполняться операции разных транзакций.

Отказы могут приводить к необходимости обрыва транзакций. В большинстве СУБД обрывы реализуются с помощью операции *отката* (rollback). Выполнение этой операции реализуется так, как будто для каждой операции записи $w(x)$, которая была выполнена в обрываемой транзакции, выполняется обратная операция $w^{-1}(x)$ в порядке, обратном тому порядку, в котором выполнялись (прямые) операции записи. По определению обратная операция записывает в объект x то значение, которое этот объект имел непосредственно перед выполнением прямой операции записи.

Во многих высокопроизводительных системах, в том числе в PostgreSQL, применяются многоверсионные протоколы управления транзакциями, в которых содержимое данных перед обновлением сохраняется в базе данных. Поскольку предыдущее состояние данных фактически сохраняется в блоках данных, записывать его заново при выполнении операции обратной записи не требуется. Для того чтобы сделать вновь записанные версии недействительными, достаточно пометить транзакцию как оборванную.

Такое восстановление состояния базы данных можно выполнять только в том случае, если данные, измененные обрываемой транзакцией, не были изменены другой транзакцией. Например, в следующей последовательности операций восстановление невозможно:

$$w_1(x)\ w_2(x)\ c_2\ a_1.$$

Дело в том, что при выполнении $w_1^{-1}(x)$ с целью отката первой транзакции было бы восстановлено значение x перед выполнением первой операции записи, и при этом было бы потеряно значение, записанное второй транзакцией.

Но вторая транзакция была зафиксирована, и поэтому ее результаты не должны быть потеряны. Такие расписания называются невосстановимыми, однако необходимо сразу заметить, что механизм управления транзакциями, применяемый в системе PostgreSQL, гарантирует восстановимость.

6.4. Диспетчеры и протоколы

Примеры, приведенные в предыдущих разделах этой главы, показывают, что бесконтрольное выполнение операций, выполняемых для различных транзакций, может приводить к нежелательным результатам. Функции управления транзакциями реализуются в СУБД диспетчером транзакций. Диспетчер должен, с одной стороны, обеспечить выполнение некоторых критериев согласованности, чтобы исключить возможность появления аномалий, а с другой — обеспечить по возможности высокую пропускную способность системы за счет конкурентного выполнения транзакций.

Эти требования являются в некотором смысле противоположными: диспетчер, выполняющий все транзакции последовательно, обеспечивает идеальную корректность при очень низкой пропускной способности, и наоборот, диспетчер, который не накладывает никаких ограничений, дает высокую пропускную способность без каких-либо гарантий корректности.

Каждый диспетчер реализует некоторый протокол управления транзакциями, т. е. набор правил, определяющих условия выполнения операций транзакций.

Наиболее часто используемым протоколом является *протокол двухфазного блокирования* (two-phase locking, 2PL) в различных модификациях. Все протоколы этого семейства используют понятие *блокировки*: перед выполнением любой операции объект данных, который обрабатывает эта операция, должен быть заблокирован; при этом для разных операций используются разные типы блокировок. После выполнения операции (но не обязательно немедленно) блокировка должна быть снята. В простейших протоколах рассматриваются только операции чтения и записи, поэтому имеется всего два типа блокировок.

Блокировки являются несовместимыми, если:

- они связаны с одним объектом данных;
- устанавливаются для разных транзакций;
- по крайней мере, одна из двух операций является операцией записи.

Из этого следует, что блокировки разных транзакций, работающих с одним объектом данных, совместимы, только если обе операции являются операциями чтения.

По правилам работы с блокировками транзакция, которая пытается установить блокировку, несовместимую с уже установленной, переводится в состояние ожидания, до тех пор пока несовместимая блокировка не будет снята.

Важно отметить, что сами по себе блокировки и правила их установки никак не гарантируют корректности конкурентного выполнения транзакций.

Двухфазный протокол блокирования (2PL) состоит в том, что, после того как хотя бы одна блокировка была снята, рассматриваемая транзакция больше не может устанавливать новые блокировки. Другими словами, на первой фазе транзакция может устанавливать блокировки на те объекты данных, которые необходимы для ее выполнения, а на второй — только снимать ранее установленные блокировки.

Отметим, что хотя имеется возможность явного управления блокировками из приложения, обычно все необходимые блокировки устанавливаются и снимаются от имени транзакции автоматически. Более того, обычно приложение не должно явно управлять блокировками, т. к. диспетчеры могут использовать различные протоколы, в том числе протоколы, не основанные на применении блокировок, или использовать их не так, как требуется при соблюдении двухфазного протокола блокирования. При применении ослабленных критериев корректности (ослабленных уровней изоляции) блокировки могут устанавливаться не для всех операций или сниматься раньше, чем это предусмотрено протоколом 2PL.

Современные протоколы управления транзакциями, в том числе протоколы, применяемые в системе PostgreSQL, рассматриваются в главе 13 второй части книги.

6.5. Использование транзакций в приложениях

В соответствии с определением понятие согласованности зависит от логики приложения: система управления базами данных может реализовать свойства транзакций только в том случае, если приложение сообщает о том, какие именно операции составляют одну транзакцию.

Для того чтобы это сделать, используют следующие операторы SQL:

START TRANSACTION указывает системе, что приложение начинает новую транзакцию. Такое же действие выполняет оператор BEGIN. В PostgreSQL этот оператор выдает сообщение об ошибке, если приложение уже выполняет транзакцию. Такое поведение соответствует стандарту SQL, однако некоторые другие СУБД в этом случае начинают вложенную подтранзакцию.

COMMIT сообщает о том, что все операции транзакции завершены и транзакцию необходимо зафиксировать. После фиксации приложение может начать новую транзакцию оператором BEGIN.

ROLLBACK вызывает обрыв текущей транзакции по инициативе приложения. В этом случае СУБД откатывает все изменения, выполненные этой транзакцией, и завершает ее. После этого приложение может начать новую транзакцию.

Во всех трех операторах BEGIN, COMMIT и ROLLBACK можно указывать необязательное ключевое слово TRANSACTION, которое не оказывает никакого влияния на их работу. В следующем примере иллюстрируется использование операторов управления транзакциями:

```
demo=# BEGIN TRANSACTION;
BEGIN

demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  5700
(1 row)

demo=# UPDATE aircrafts
SET range = 6200
WHERE aircraft_code = '320';
UPDATE 1

demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  6200
(1 row)

demo=# ROLLBACK;
ROLLBACK
```

```
demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model        | range
---------------+-------------------+-------
 320           | Аэробус A320-200 |  5700
(1 row)
```

В этом примере выполняется обновление строки внутри транзакции, а затем эта транзакция обрывается. Операторы SELECT нужны, конечно, только для того, чтобы показать, как изменяется состояние базы данных. Заметим, что различие в результатах работы оператора SELECT не является индикацией аномалии нечеткого чтения, т. к. транзакция в данном случае видит изменения, выполненные ей же самой.

Отметим важную особенность транзакционного поведения PostgreSQL: обрыв транзакций не приводит к восстановлению использованных значений последовательностей. В приведенном ниже примере повторное выполнение той же самой транзакции дает другое значение для атрибута flight_id, несмотря на то что предыдущая транзакция была оборвана. Поскольку предполагаемое использование последовательностей — генерация уникальных значений, такое поведение системы корректно. Можно сказать, что обрывы транзакций возвращают базу данных в логически корректное состояние, эквивалентное ее состоянию до выполнения транзакции, но эти состояния необязательно идентичны. Такое же поведение последовательностей реализуется и в других СУБД.

```
demo=# BEGIN;
BEGIN
demo=# INSERT INTO flights(
    flight_no, scheduled_departure, scheduled_arrival,
    departure_airport, arrival_airport, status, aircraft_code)
VALUES (
    'PG9999', now(), now() + interval '2 hours',
    'SVO', 'LED', 'Delayed', '321')
RETURNING flight_id;
 flight_id
-----------
     33122
(1 row)
INSERT 0 1
demo=# ROLLBACK;
ROLLBACK
demo=# BEGIN;
BEGIN
```

```
demo=# INSERT INTO flights(
    flight_no, scheduled_departure, scheduled_arrival,
    departure_airport, arrival_airport, status, aircraft_code)
VALUES (
    'PG9999', now(), now() + interval '2 hours',
    'SVO', 'LED', 'Delayed', '321')
RETURNING flight_id;
 flight_id
-----------
     33123
(1 row)
INSERT 0 1
demo=# ROLLBACK;
ROLLBACK
```

Такое поведение последовательностей позволяет несколько расширить возможности одновременного выполнения транзакций, использующих одну и ту же последовательность.

Если при выполнении оператора, находящегося внутри транзакции, происходит ошибка, то транзакция обрывается, и любая попытка выполнить какой-либо оператор (кроме оператора, завершающего транзакцию) в рамках той же транзакции также приводит к индикации ошибки. При этом, конечно, пустой результат выполнения запроса или пустое множество изменяемых строк не считается ошибкой.

Если оператор SQL выполняется вне транзакции (например, приложение не выполняло оператор BEGIN), то поведение системы зависит от настроек клиента. Если установлен режим автоматической фиксации, каждый оператор превращается в отдельную транзакцию, которая не может быть оборвана по инициативе приложения, поскольку всегда завершается неявным оператором COMMIT (или ROLLBACK, если операция выполнена с ошибкой). Такой режим используется по умолчанию в psql. Если же автоматическая фиксация не установлена, то перед выполнением первого оператора транзакции выполняется неявная команда BEGIN.

6.6. Уровни изоляции

Полное выполнение всех ACID-свойств транзакций обеспечивает корректность совместного использования базы данных в том смысле, что работа каждого из одновременно (конкурентно) выполняемых приложений происходит так, как если бы с СУБД работало только одно приложение. Однако для реализации

этих свойств сервер базы данных должен несколько ограничивать выполнение транзакций, что может приводить к снижению пропускной способности системы и к увеличению времени ожидания ответа для отдельных транзакций.

Это в особенности справедливо для простых протоколов управления транзакциями, основанных на использовании блокировок. Такие протоколы применялись в ранних реляционных СУБД и до сих пор используются в простых системах. Чтобы исключить снижение пропускной способности и сократить время ожидания приложений, возможны два пути:

- использовать другие, более сложные протоколы управления транзакциями, в меньшей степени снижающие производительность (пропускную способность) СУБД;
- полностью или частично отказаться от соблюдения ACID-свойств транзакций, т. е. снизить требования к согласованности данных.

По историческим причинам первые версии стандарта SQL были разработаны значительно раньше, чем появились высокопроизводительные протоколы управления транзакциями. По этим причинам стандарт SQL предусматривает возможности ослабления свойств ACID. В частности, ослабляются требования по изоляции транзакций, поэтому соответствующие режимы выполнения транзакций называются *уровнями изоляции*.

Некоторые из высокопроизводительных протоколов управления транзакциями, реализованных в системе PostgreSQL, обсуждаются во второй части книги. Заметим, что даже при использовании таких протоколов ослабление требований может приводить к некоторому увеличению производительности СУБД.

Уровни изоляции в стандарте SQL определены в терминах аномалий, т. е. для каждого из уровней указывается, какие из аномалий допустимы для этого уровня. Реализация СУБД, однако, может предотвращать появление некоторых аномалий, даже если они допускаются стандартом SQL для некоторого уровня.

Для всех уровней изоляции требуется атомарность, т. е. любая транзакция либо выполняется полностью, либо ее выполнение не оставляет никаких следов в базе данных. Кроме этого, для всех уровней изоляции не допускается аномалия потерянного обновления.

Стандартом предусмотрены следующие уровни изоляции:

Read Uncommitted разрешает доступ к результатам выполнения еще не зафиксированных транзакций и никак не ограничивает выполнения транзакций, тем самым допуская появление любых аномалий;

Read Committed — результаты других транзакций становятся доступными после их фиксации, т. е. запрещается аномалия грязного чтения;

Repeatable Read — повторное выполнение операций поиска и выборки данных дает такие же результаты, как первое, т. е. запрещаются аномалии грязного и нечеткого чтения;

Serializable требует, чтобы выполнение транзакций было эквивалентно некоторому последовательному выполнению.

В реализации PostgreSQL фактически используются только три последних уровня. Указание Read Uncommitted допускается, однако этот уровень работает точно так же, как Read Committed. Заметим, что в PostgreSQL применяются более сложные механизмы управления транзакциями, поэтому никакой потери производительности не происходит.

Уровень изоляции для отдельной транзакции указывается оператором SET TRANSACTION:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Возможно также указание (или изменение) уровня изоляции с помощью параметров сеанса или для всего сервера баз данных.

Если ничего не указывать, то PostgreSQL устанавливает уровень изоляции в соответствии со значением параметра default_transaction_isolation, обычно Read Committed. Это обеспечивает наиболее высокую пропускную способность, однако может приводить к появлению некоторых аномалий при обработке транзакций. В частности, возможно появление фантомов и аномалии нечеткого чтения.

Поскольку операторы поступают от приложений в произвольном порядке, СУБД не всегда может организовать выполнение транзакций, обеспечивая требуемый уровень изоляции. В этом случае транзакция, которую невозможно выполнить, обрывается по инициативе СУБД при попытке выполнения той операции транзакции, которую невозможно включить в общее расписание. В PostgreSQL приложение извещается об этой ситуации с помощью кода ошибки, возвращаемого при попытке выполнения операции. Такая ошибка может появляться, если приложение использует уровни Repeatable Read или Serializable. При ее появлении приложению следует повторить попытку выполнения транзакции заново (так, как будто она еще не выполнялась).

6.7. Точки сохранения

Полезность свойства атомарности состоит в том, что в случае неудачного выполнения некоторой транзакции частичные изменения, выполненные этой транзакцией, удаляются из базы данных и не мешают работе других транзакций. Следовательно, каждое приложение вправе ожидать, что оно начинает работу с корректным (согласованным) состоянием базы данных. Однако ресурсы, затраченные на выполнение оборванной транзакции, оказываются безвозвратно потерянными. Это вполне допустимо для коротких транзакций, на выполнение каждой из которых затрачивается небольшое количество ресурсов, и если аварийные завершения случаются редко. В противоположность этому потеря результатов выполнения работы большого объема нежелательна.

Для транзакций, выполняющих большие объемы работы, может быть целесообразно использовать оператор SAVEPOINT, который запоминает состояние всех данных, измененных транзакцией к моменту его выполнения. В случае если при продолжении выполнения транзакции обнаруживается ошибка, можно восстановить состояние базы данных, выполняя операцию ROLLBACK с указанием имени того состояния, которое было предварительно записано оператором SAVEPOINT.

В следующем простом примере иллюстрируется работа операторов создания точек сохранения. В рамках транзакции выполняется изменение строки и создается точка сохранения; затем эта строка удаляется. После отката к точке сохранения с помощью оператора ROLLBACK вставленная строка снова появляется в базе данных. Наконец, вся транзакция обрывается — база данных восстанавливается в состояние, эквивалентное тому, которое было до начала транзакции.

```
demo=# BEGIN TRANSACTION;
BEGIN
demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  5700
(1 row)
demo=# UPDATE aircrafts
SET range = 6200
WHERE aircraft_code = '320';
UPDATE 1
```

```
demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  6200
(1 row)

demo=# SAVEPOINT svp;
SAVEPOINT

demo=# DELETE FROM aircrafts
WHERE aircraft_code = '320';
DELETE 1

demo=# ROLLBACK TO svp;
ROLLBACK

demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  6200
(1 row)

demo=# ROLLBACK;
ROLLBACK

demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  5700
(1 row)
```

Механизм точек сохранения дает возможность организовать достаточно сложное поведение в рамках одной транзакции, но для других транзакций ее поведение остается атомарным.

6.8. Долговечность

Свойство долговечности требует, в частности, чтобы СУБД гарантировала сохранность изменений, выполненных зафиксированными транзакциями, в случае системного отказа или отказа носителя. После перезапуска сервера база

данных всегда оказывается в согласованном состоянии и содержит все изменения, выполненные зафиксированными транзакциями. Для того чтобы удовлетворить этим требованиям, СУБД выполняет некоторые дополнительные действия во время нормальной работы системы.

В большинстве систем (в том числе в PostgreSQL) для реализации этого свойства используются *журналы СУБД*, в которых последовательно записываются все изменения, выполненные всеми транзакциями, а также записи о фиксации транзакций. Благодаря тому, что запись в журнал ведется последовательно, такой способ обеспечения корректности восстановления оказывается наиболее эффективным. Правила ведения журналов и алгоритмы восстановления обсуждаются в главе 14 второй части книги.

Несмотря на относительную эффективность журналов, расходы на их ведение довольно значительны и в отдельных случаях в некоторых СУБД могут превышать затраты ресурсов на обновление основного хранилища СУБД. Может оказаться, что для части данных такая степень долговечности не нужна. Например, данные о состоянии сеанса связи с пользователем, скорее всего, очень часто обновляются, но становятся ненужными, когда сеанс прекращается. Поэтому можно повысить производительность системы, если исключить действия, обеспечивающие возможность восстановления этих данных в случае отказа системы.

В PostgreSQL есть возможность отказаться от регистрации изменений в журналах БД для отдельных таблиц. Заметим, что для таких таблиц ослабляется только один из аспектов долговечности, а именно возможность восстановления после отказов. Поведение таких таблиц по отношению к фиксации транзакций не отличается от поведения обычных (журналируемых) таблиц.

6.9. Итоги главы

В этой главе приведены основные свойства транзакций и требования, которые предъявляются к их выполнению. Охарактеризованы методы, с помощью которых могут достигаться требуемые свойства, возможные компромиссы и последствия применения ослабленных критериев согласованности. Показаны операторы языка SQL, необходимые для управления транзакциями на уровне приложений.

6.10. Упражнения

Пример транзакции

Упражнение 6.1. Начните транзакцию (командой BEGIN) и создайте новое бронирование в таблице bookings сегодняшней датой. Добавьте два электронных билета в таблицу tickets, связанных с созданным бронированием.

Представьте, что пользователь не подтвердил бронирование и все введенные данные необходимо отменить. Выполните отмену транзакции и проверьте, что никакой добавленной вами информации действительно не осталось.

Упражнение 6.2. Теперь представьте сценарий, в котором нужно отменить не все данные, а только последний из добавленных электронных билетов. Для этого повторите все действия из предыдущего упражнения, но перед добавлением каждого билета создавайте точку сохранения (с одним и тем же именем). После ввода второго билета выполните откат к точке сохранения. Проверьте, что бронирование и первый билет остались.

Упражнение 6.3. В рамках той же транзакции добавьте еще один электронный билет и зафиксируйте транзакцию. Обратите внимание на то, что после этой операции отменить внесенные транзакцией изменения будет уже невозможно.

Уровень изоляции Read Committed

Упражнение 6.4. Перед началом выполнения задания проверьте, что в таблице bookings нет бронирований на сумму total_amount 1 000 рублей.

1. В первом сеансе начните транзакцию (командой BEGIN). Выполните обновление таблицы bookings: увеличьте total_amount в два раза в тех строках, где сумма равна 1 000 рублей.
2. Во втором сеансе (откройте новое окно psql) вставьте в таблицу bookings новое бронирование на 1 000 рублей и зафиксируйте транзакцию.
3. В первом сеансе повторите обновление таблицы bookings и зафиксируйте транзакцию.

Осталась ли сумма добавленного бронирования равной 1 000 рублей? Почему это не так?

Уровень изоляции Repeatable Read

Упражнение 6.5. Повторите предыдущее упражнение, но начните транзакцию в первом сеансе с уровнем изоляции транзакций Repeatable Read. Объясните различие полученных результатов.

Упражнение 6.6. Выполните указанные действия в двух сеансах:

1. В первом сеансе начните новую транзакцию с уровнем изоляции Repeatable Read. Вычислите количество бронирований с суммой 20 000 рублей.
2. Во втором сеансе начните новую транзакцию с уровнем изоляции Repeatable Read. Вычислите количество бронирований с суммой 30 000 рублей.
3. В первом сеансе добавьте новое бронирование на 30 000 рублей и снова вычислите количество бронирований с суммой 20 000 рублей.
4. Во втором сеансе добавьте новое бронирование на 20 000 рублей и снова вычислите количество бронирований с суммой 30 000 рублей.
5. Зафиксируйте транзакции в обоих сеансах.

Соответствует ли результат ожиданиями? Можно ли сериализовать эти транзакции (иными словами, можно ли представить такой порядок последовательного выполнения этих транзакций, при котором результат совпадет с тем, что получился при параллельном выполнении)?

Уровень изоляции Serializable

Упражнение 6.7. Повторите предыдущее упражнение, но транзакции в обоих сеансах начните с уровнем изоляции Serializable.

Если вы правильно ответили на его последний вопрос, вы поймете, почему теперь эти действия приводят к ошибке. Если же результат этого упражнения стал для вас неожиданностью, четко сформулируйте различие уровней Repeatable Read и Serializable.

Вложенные транзакции

Упражнение 6.8. Некоторые СУБД (но не PostgreSQL) позволяют использовать вложенные транзакции. Если начать вторую транзакцию, не завершая уже открытую первую, то вторая транзакция будет считаться *вложенной*: ее результат фиксируется только в том случае, если фиксируется первая транзакция, но при этом ее результаты можно отменить независимо от первой транзакции.

Реализуйте такое поведение для PostgreSQL, т. е. предложите, какие команды следует выполнять для открытия вложенной транзакции, для ее отмены и для фиксации. Подсказка: используйте оператор SAVEPOINT.

Глава 7

Разработка приложений СУБД

Технологии разработки приложений не являются частью технологий баз данных, которые составляют основное содержание этой книги. Однако нельзя обойти вопросы, связанные с разработкой приложений, поскольку на сегодняшний день базы данных практически не используются сами по себе, а являются хранилищем для бизнес-приложений. Обычно пользователь работает с приложением через графический интерфейс, специально разработанный для решения задач предметной области. Интерфейс скрывает детали реализации системы, и пользователю не нужно знать ни код приложения, ни тем более структуру базы данных, и не нужно уметь применять язык запросов.

Методы разработки приложений относятся к группе дисциплин, составляющих программную инженерию, в которой применяются принципы и критерии, существенно отличающиеся от критериев, применяемых при разработке баз данных. В частности, обычно наиболее важными критериями оказываются стоимость и время, затраченные на разработку приложения, а не критерии производительности, рассмотренные в главе 1. Опытные разработчики хорошо знают, что в начале каждого проекта по созданию приложения им необходимо сделать выбор между следующими альтернативами:

- тщательное проектирование структуры базы данных и запросов на основе анализа требований;
- использование программных средств, обеспечивающих быструю разработку приложений.

Хорошо известно, что первый вариант обеспечивает высокое качество и высокую производительность, а второй — быстрое получение результатов и низкую стоимость, достигаемые за счет многочисленных компромиссов, ухудшающих качество и производительность результата. В подавляющем большинстве случаев руководство выбирает второй вариант.

Интересный анализ применения различных подходов к проектированию приложений, работающих с базами данных, содержится в докладе [50].

Наиболее серьезные проблемы, связанные с реализацией приложений, работающих с базами данных, так или иначе оказываются следствиями или проявлениями общей проблемы различия вычислительных моделей и моделей данных, применяемых в доминирующих языках программирования приложений и в системах управления базами данных. Это несоответствие известно под метафорическим названием «потери соответствия» (impedance mismatch). Как СУБД, так и языки программирования могут работать в терминах высокоуровневых абстракций, однако системы этих абстракций различаются, и, что еще важнее, взаимодействие между этими моделями обычно описывается в терминах низкоуровневых интерфейсов.

Задачи предметной области реализуются при помощи программного кода, так называемой бизнес-логики приложения. В программной инженерии плохой практикой считается программирование бизнес-логики на уровне пользовательского интерфейса, поскольку требования к интерфейсу меняются наиболее часто. Это удорожает разработку, осложняя повторное использование кода и применение уже готовых и протестированных сторонних (third-party) компонентов. Чтобы минимизировать влияние изменений требований, применяется архитектура приложения, включающая тонкого клиента, единственной задачей которого становится отображение данных.

Независимо от используемых технологий хорошо спроектированное (с точки зрения, принятой в программной инженерии) приложение имеет слой, абстрагирующий доступ к данным (data access layer). Такой подход позволяет ограничить доступ к данным на уровне программной архитектуры и дает возможность заменить СУБД на какое-нибудь альтернативное решение, не затрагивая бизнес-логику. Возможность работы с разными СУБД может быть полезна для тиражируемых программных продуктов, но совсем не имеет значения при разработке конкретных информационных систем и, более того, может приводить к снижению качества приложения. Заметим также, что этот аргумент игнорирует необходимость миграции данных, которая может оказаться более дорогостоящим проектом, чем разработка приложения. Другая проблема состоит в том, что слой, абстрагирующий доступ к данным, зачастую полностью блокирует доступ приложения к высокоуровневым и высокоэффективным средствам, предоставляемым СУБД.

Другим крайним вариантом является размещение бизнес-логики в хранимых процедурах. По данным, приведенным в [50], такое решение применяется примерно в 11 % приложений баз данных. Это позволяет использовать общий код, выполняемый в базе данных, в нескольких приложениях и создает благоприятные условия для эффективной реализации и настройки этих функций. Одно-

временно появляется возможность сохранить бизнес-логику при переходе на другой язык программирования приложения (хотя возможность смены языка программирования обычно не рассматривается). Такой подход снижает зависимость кода приложения от схемы базы данных, а также повышает производительность, однако приводит к существенному увеличению стоимости разработки.

В этой главе мы коснемся вопросов программирования приложений, взаимодействующих с базами данных. Несмотря на то что подробное освещение этой темы выходит за рамки данного курса, мы познакомим вас с основными технологиями доступа к данным и расскажем о проблемах интеграции баз данных и приложений.

7.1. Проектирование схемы базы данных

Методологии проектирования информационных систем, развитые в течение последних десятилетий XX века, предусматривали построение различных моделей, в том числе моделей данных, на основании анализа предметной области. Полученные модели использовались для создания схемы базы данных.

С развитием технологий многие организации стали ощущать потребность в документировании своих внутренних бизнес-процессов. Бизнес-процесс — это комплекс взаимосвязанных действий, направленных на создание продукта или услуги, представляющих ценность для потребителей. В задачах автоматизации предприятия описания бизнес-процессов используются в качестве требований к будущим информационным системам. Языки описания бизнес-процессов не моделируют данные, а описывают сообщения, которые являются основным способом коммуникации между участниками процесса.

Наряду с моделированием бизнес-процессов развитые методологии проектирования предусматривают создание целого ряда других моделей, в том числе диаграмм потоков данных и моделей данных, а также перекрестную проверку соответствия этих моделей. Однако создание таких моделей хотя и повышает качество проектируемой системы, но существенно увеличивает сроки разработки и ее стоимость. Поэтому дополнительные модели создаются редко, а современные инструменты не поддерживают их совместное применение. Это означает, что вы не можете нарисовать диаграмму бизнес-процессов и связать ее с моделью данных. На практике приходится использовать оба типа моделирования, а связывание производить вручную.

В этом курсе нас интересует построение моделей данных. Процесс моделирования начинается со сбора и анализа требований. На основании требований строится концептуальная модель, описывающая основные сущности информационной системы и взаимосвязи между ними. Далее множество сущностей концептуальной модели уточняется, к ним добавляются атрибуты и ограничения, и в результате создается логическая модель системы.

Логическая модель системы содержит все сведения, необходимые для написания программного кода. При этом существенно изменяется точка зрения на роль СУБД: если традиционно база данных рассматривалась как разделяемый ресурс, используемый несколькими приложениями, то при проектировании на основе логической модели предполагается, что база данных доступна только через монолитное приложение, созданное на основе логической модели. В результате появляется иллюзия того, что многие средства СУБД, такие как независимое описание данных и поддержка конкурентного доступа, становятся ненужными. Отказ от использования функций СУБД вынуждает добавлять компоненты, дублирующие эти функции вне СУБД, что, по-видимому, не удешевляет систему, однако на той фазе жизненного цикла системы, когда необходимость в этих функциях становится очевидной, изменение архитектуры, как правило, оказывается уже невозможным.

На практике данные хранятся в реляционных СУБД, а программы, обрабатывающие эти данные, пишутся на современных объектно-ориентированных языках программирования, не приспособленных для использования высокоуровневых возможностей СУБД. То есть из логической модели нужно получить и схему базы данных, представляющую данные в виде реляционных таблиц, и объектную модель данных.

При проектировании схемы базы данных неизбежны компромиссы, обычно ухудшающие свойства базы данных, но упрощающие согласование с объектной моделью приложения. В частности, схемы низкого качества получаются при использовании инструментов, реализующих объектно-реляционные отображения. Причины, по которым компромиссы необходимы, связаны со следующими различиями в свойствах моделей:

Идентификация. В моделях данных СУБД применяется естественная идентификация, а в объектных моделях приложений — идентификация на основе суррогатов. Как правило, это приводит к включению суррогатных ключей, никак не связанных с предметной областью, в логическую модель данных и в схему базы данных.

Следствием такого решения может оказаться появление скрытых дубликатов вследствие ошибок в коде приложения, а также других дефектов данных, необходимость обработки которых может приводить к существенному усложнению кода приложения.

Представление связей. В моделях СУБД используются ассоциативные связи, основанные на значениях атрибутов и устанавливаемые динамически во время выполнения запросов, а в объектных моделях используются статические связи, представленные суррогатными объектными указателями.

Для того чтобы приблизить структуру базы данных к структуре модели приложения, связи в базе данных (первичные и внешние ключи) определяются в терминах суррогатов, и, таким образом, ответственность за корректность связей также возлагается на приложение.

Другие различия в моделях в меньшей степени влияют на схему БД, однако существенно влияют на вид запросов, обеспечивающих доступ к данным.

Современные методы разработки предполагают итеративный подход. Многие детали будут обнаружены после построения первых версий модели, а возможно, и после создания первых работающих версий системы. Изменения придется вводить в течение всего проекта, и необходимо будет поддерживать все модели в согласованном состоянии.

Для ускорения разработки применяется генерирование программных артефактов. Инструменты проектирования позволяют генерировать из логической модели объекты программного кода, файлы с объектно-реляционными отображениями и схему базы данных. Сгенерированные классы, как правило, не требуют серьезных изменений, но схему базы данных и объектно-реляционные отображения нельзя получить полностью автоматически — требуется ручная настройка.

Возможный вариант компромисса между требованиями методологий и систем разработки приложений (в первую очередь каркасов) и требованиями, обеспечивающими высокое качество проектируемой схемы, получается при использовании следующих правил:

- Необходимо синхронизировать логическую и объектную модели. Логическая модель приложения — это артефакт, который будет использоваться не только программистами, но также аналитиками или тестировщиками, которые незнакомы с тонкостями ООП или реляционной теории. Необходимо, чтобы логическая модель всегда была актуальной.

- Первую версию объектно-реляционных отображений можно получить при помощи генерирования, однако затем необходимо уточнить типы, ограничения, отображения взаимосвязей и наследования.
- Схему базы данных нужно создавать или модифицировать при помощи скриптов SQL, используя типы и особенности, поддерживаемые выбранной СУБД. Генерирование при помощи каркасов, как правило, не бывает однозначным, и, генерируя схему в разных окружениях, можно получать разные результаты (например, порядок колонок в таблице).

Последняя из перечисленных рекомендаций связана с тем, что обычно каркасы не могут модифицировать уже существующую схему базы данных, а генерируют ее заново. При этом, конечно, игнорируются основные принципы СУБД, в частности принцип разделения (независимости) данных и программ, из которого следует, что порядок колонок в таблицах не имеет никакого логического значения.

В современном мире каркасы для разработки приложений появляются намного чаще, чем меняются технологии работы с данными. Разработанная система через некоторое время устареет и может быть переписана с помощью новых языков программирования и библиотек; при этом данные останутся и будут перенесены в новую систему. Развертывание новой версии приложения — это, как правило, быстрая и дешевая операция, которая проводится в реальном времени, а в случае неудачи можно вернуться к предыдущей версии. Миграция данных занимает значительно больше времени, а ошибки в миграции данных приводят к серьезным сбоям и даже остановкам системы. Поэтому в разработке информационных систем вопрос проектирования данных является одним из самых важных.

7.2. Объектно-реляционная потеря соответствия

Одновременное использование объектно-ориентированной и реляционной парадигм вызывает определенные трудности. В процессе проектирования мы получаем логическую модель информационной системы, которая, вообще говоря, не привязана к конкретным технологиям. На следующем этапе нам нужно получить модель, которую будет использовать приложение, и модель, представляющую данные в СУБД. Эти модели имеют серьезные различия, и здесь мы сталкиваемся с проблемой, которая называется *объектно-реляционной потерей соответствия* (object-relational impedance mismatch).

При обсуждении потери соответствия обычно указывают следующие различия:

Идентификация объекта. В реляционной базе данных запись идентифицируется своим первичным ключом, который является частью состояния записи. В объектном мире идентификация объекта не зависит от его состояния: несколько объектов с одинаковым состоянием могут иметь разную идентификацию. При проектировании приложений возникает вопрос, обсуждаемый в главе 2: как однозначно идентифицировать сущность в обеих парадигмах.

Системы типов. Типы данных в языках программирования не соответствуют типам в реляционных базах данных. Например, отличаются представление строк и диапазон числовых значений. Также объектно-ориентированные языки не поддерживают неопределенных значений для примитивных типов.

Этот вид различий моделей оказывается важным, потому что разработчики приложений игнорируют возможность задания ограничений целостности, поскольку такие ограничения трудно формулировать в объектных моделях. Так, ограничения на диапазоны значений зависят от предметной области приложения, а не от возможностей языка программирования, и могут быть легко выражены в схеме базы данных (например, определением доменов в PostgreSQL), но не могут быть выражены в объектной модели приложения.

Навигация между объектами. В отличие от реляционных систем, основанных на ассоциативных связях, объектные модели поддерживают возможность навигации с помощью ссылок. Если приложение выбирает множество записей и отображает их в объекты, нужно ли выбирать также объекты, на которые ссылается первое множество? Что делать, если бизнес-процесс требует только часть состояния объекта?

В действительности эти вопросы не связаны с несоответствием моделей, т. к. они сформулированы полностью в терминах объектов. Проблема здесь в том, что при использовании каркасов запросы на выборку данных генерируются автоматически, а логическая модель не дает достаточной информации о том, какие части или какие другие объекты необходимо выбирать из базы данных.

Навигационный поиск объектов может приводить и к более серьезным осложнениям, связанным с проверкой значений атрибутов.

Инкапсуляция. Состояние объекта изменяется только при вызове его методов. Считается, что вызов метода, выполняющего присваивание значения, безопаснее, чем оператор присваивания, потому что код метода может выполнить дополнительные проверки. Считается, что базы данных не имеют такой защиты и содержание записей может быть изменено сторонними приложениями. В базах данных аналогичную роль выполняют ограничения целостности и триггеры, а возможность доступа из разных приложений необходима для совместного использования данных.

С другой стороны, обязательность вызова методов приводит к тому, что объекты могут обрабатываться только по одному (метод объекта не применим к множеству), а высокая эффективность баз данных достигается, если запросы обрабатывают множества объектов.

Снова проблема не в том, что в объектных моделях есть нечто, невозможное в моделях баз данных, а в том, что эти модели различаются.

Различные библиотеки и каркасы пытаются решить вышеуказанные проблемы и сделать работу с базой данных прозрачной для программистов, разрабатывающих приложения. Однако необходимо заметить, что большая часть перечисленных осложнений возникает именно потому, что авторы каркасов пытаются добиться «прозрачности», игнорируя особенности моделей баз данных. Намного более серьезные осложнения возникают вследствие других различий:

- объектные языки ориентированы на обработку объектов по одному, а реляционная модель данных оперирует множествами;
- ассоциативный поиск и связывание по значениям атрибутов, применяемые в реляционной модели данных, плохо согласуются с навигацией по объектным указателям.

Эти различия не только усложняют разработку кода приложения, но и могут приводить к ухудшению характеристик производительности вследствие низкой эффективности приложения, написанного без учета этих различий.

7.3. Использование каркасов объектно-реляционных отображений

Каркасы *объектно-реляционных отображений* (object-relational mapping, ORM) определяют, каким образом объекты приложения будут сохранены в таблицах

базы данных, и позволяют выбирать записи, используя специальный объектный язык запросов.

Рассмотрим простой случай, когда у класса нет предков и потомков, а его атрибуты имеют примитивные типы. Такой класс отображается в отношение, а атрибуты класса — в атрибуты отношения. При этом имена классов и атрибутов сохраняются или преобразуются согласно правилам именования. Для атрибута отношения каркас отображения выбирает тип, максимально соответствующий типу атрибута класса. Также на уровне отображения устанавливаются ограничения, не поддерживаемые напрямую в объектных языках программирования. Например, допускаются ли неопределенные значения, является ли атрибут уникальным и из каких атрибутов состоит первичный ключ.

Каркасы минимизируют труд программистов по разработке кода приложений, что отвечает основному критерию программной инженерии. Однако получаемый таким способом код обычно оказывается низкоэффективным, и созданные приложения могут работать только с небольшими базами данных и при невысокой интенсивности их использования.

7.3.1. Наследование

Одним из наиболее важных и широко используемых понятий объектно-ориентированного подхода является понятие наследования. Существует большое количество разных моделей для описания и для применения наследования. Неформально наследование используется для указания того факта, что свойства объектов из некоторого множества имеются также и у объектов подмножества этого множества. Это не только позволяет исключить повторное описание данных свойств, но и дает возможность обрабатывать объекты меньшего множества вместе с объектами большего. Например, свойства всех автомобилей применимы и к легковым автомобилям.

Поскольку в императивных объектно-ориентированных языках программирования наследование применяется довольно часто, возникает задача отображения наследования при использовании нескольких различающихся моделей, в частности при организации взаимодействия между базой данных и приложением.

Реляционная теория не поддерживает напрямую понятие наследования, а наследование в модели данных «сущность — связь», также как и наследование, реализованное в современных системах (в том числе в PostgreSQL), существенно

отличаются от принятых в объектных моделях различных языков программирования. В этой главе обсуждаются варианты отображения структур наследования объектно-ориентированных языков в базу данных в предположении, что СУБД не поддерживает понятия наследования. Такие отображения важны, поскольку каркасы не используют средств наследования СУБД, даже если они имеются. Возможности применения наследования в системе PostgreSQL обсуждаются ниже в главе 8.

Для иллюстрации отображения иерархии наследования в табличную форму используем предельно упрощенный пример. Пусть класс person содержит информацию обо всех лицах, связанных с учебным заведением, подкласс professor — о преподавателях, а подкласс student — о студентах. Преподаватели имеют дополнительный атрибут — год присвоения ученой степени, а студенты — номер студенческого билета.

Для отображения наследования объектной модели на упрощенную табличную модель используется одна из следующих стратегий.

Отображение иерархии классов в одну таблицу. Все атрибуты классов в иерархии записываются в одну таблицу. Один специальный атрибут (дискриминатор) указывает, какой именно класс объекта соответствует записи.

Пример такого представления показан на рис. 7.3.1.

persons

class	name	title	degree	stud_id
person	Елена	секретарь		
person	Анастасия	лаборант		
professor	Аристарх	профессор	2001	
professor	Ксенофон	доцент	2012	
student	Анна	студент		1451
student	Виктор	студент		1432
student	Нина	студент		1556

Рис. 7.3.1. Представление иерархии наследования в одной таблице

При этом для выбора объекта не надо использовать операции соединения, что в некоторых случаях является преимуществом.

Недостатком подхода является то, что нельзя задавать ограничения целостности NOT NULL для атрибутов подклассов, поскольку другие подклассы будут иметь неопределенное значение в соответствующих колонках, а также многие колонки будут иметь пустые значения. Возможность записи неопределенных значений можно регулировать с помощью ограничений целостности CHECK, включающих значения атрибута-дискриминатора, но каркасы не генерируют ограничения такого типа.

Кроме этого, извлечение объектов некоторого подкласса может оказаться вычислительно неэффективным. Например, операция полного просмотра списка профессоров (относительно небольшое множество) потребует просмотра всей информации из таблицы persons (которая по размеру значительно больше).

Горизонтальное разделение на таблицы. Атрибуты класса, а также унаследованные атрибуты отображаются в колонки таблицы, при этом для каждого класса используется отдельная таблица.

Пример такого представления показан на рис. 7.3.2.

persons

name	title
Елена	секретарь
Анастасия	лаборант

professors

name	title	degree
Аристарх	профессор	2001
Ксенофон	доцент	2012

students

name	title	stud_id
Анна	студент	1451
Виктор	студент	1432
Нина	студент	1556

Рис. 7.3.2. Горизонтальная фрагментация таблиц при наследовании

В этом случае для загрузки объектов тоже не надо выполнять операцию соединения. Однако метод поиска во всей иерархии классов должен будет просмотреть все таблицы, соответствующие подклассам (т. е. выполнить операцию UNION в терминах SQL). Как и в предыдущем варианте, идентификация объектов должна быть уникальна в иерархии.

Этот метод представления наследования близок к модели наследования, которая используется в системе PostgreSQL, и поддерживается расширенным синтаксисом SQL, однако каркасы для разработки приложений не используют возможности СУБД.

Вертикальное разбиение. Каждый класс отображается в таблицу и хранит только атрибуты своего класса (без унаследованных). Атрибут, являющийся первичным ключом, определяется в корневом классе. Таблицы, соответствующие подклассам, ссылаются при помощи внешнего ключа на таблицы, соответствующие предкам. Для загрузки объекта нужно выполнить операцию соединения.

Пример такого отображения показан на рис. 7.3.3.

persons

name	title
Елена	секретарь
Анастасия	лаборант
Аристарх	профессор
Ксенофон	доцент
Анна	студент
Виктор	студент
Нина	студент

professors

name	degree
Аристарх	2001
Ксенофон	2012

students

name	stud_id
Анна	1451
Виктор	1432
Нина	1556

Рис. 7.3.3. Фрагментация таблиц при наследовании в модели ER

Такой способ наследования применяется в модели данных «сущность — связь», если внешние ключи в таблицах, соответствующих подклассам, являются также первичными ключами в этих таблицах. В некоторых языках программирования, однако, для этого могут создаваться отдельные атрибуты, содержащие суррогатные ключи.

Возможны и другие варианты отображения. Например, в последнем варианте можно избыточно хранить в каждой таблице все атрибуты соответствующего класса, а не только специфические для этого класса. Такая избыточность может ускорить выполнение операций выборки данных, но, конечно, потребует дополнительных затрат при модификации данных.

7.3.2. Запросы

Для чтения объектов из базы данных используется объектный язык запросов, который обычно эквивалентен небольшому подмножеству SQL с несколько отличающимся синтаксисом. Запросы в нем формулируются в терминах объектной, а не реляционной модели и не зависят от схемы базы данных, поскольку отображение строится каркасом автоматически при преобразовании этих запросов в SQL. Запрос может быть представлен в виде строки или иметь объектную структуру. Результатом выполнения такого запроса является (в терминах теории графов) лес, каждое дерево в котором соответствует некоторому объекту верхнего (по отношению к запросу) уровня и некоторым объектам, достижимым из него через объектные ссылки.

Поскольку навигация между объектами осуществляется с помощью ссылок, важно определить, будут ли выбираться также объекты, на которые ссылаются объекты верхнего уровня. Такие объекты можно или загружать всегда, или использовать «ленивую загрузку», т. е. загружать объект, когда происходит доступ по ссылке. В случае ленивой загрузки доступ к каждому свойству навигации приводит к выполнению отдельного запроса к базе данных. Здесь можно столкнуться с проблемой, которая называется *проблемой эн плюс одной выборки*. Если мы выбираем объект, который ссылается на n объектов, то для получения всей структуры нам понадобится выполнить $n+1$ запрос. Однако в случае доступа ко всем сопряженным объектам эффективнее было бы выбрать всю структуру за один запрос. Важно понимать, как будет происходить доступ к ассоциированным объектам, потому что их загрузка серьезно влияет на производительность приложения.

Многие объектные языки запросов не поддерживают пакетного обновления или удаления объектов, что создает определенные неудобства и плохо влияет на производительность. Кроме этого, выделение слоя абстракции данных практически исключает возможность выполнения обновлений без считывания данных в память приложения, что во многих случаях удваивает количество запросов, выполняемых приложением.

7.3.3. Когда применять каркасы?

Каркасы объектно-реляционных отображений сокращают усилия, которые необходимо приложить программисту. Однако они работают в рамках унифицированного и сильно ограниченного подмножества стандарта SQL 1992 г. и

не используют новых возможностей, предусмотренных более поздними версиями стандарта, и тем более расширений, специфических для СУБД. Не используются также многие конструкции SQL. Однако основной причиной низкой эффективности приложений, разработанных с использованием каркасов, оказывается слишком большое количество слишком мелких запросов. Эффект $n + 1$ запроса повторяется на каждом уровне иерархии объектов, и количество запросов, очевидно, растет экспоненциально с ростом глубины вложенности. Фактически при этом высокоуровневые операции базы данных (такие как соединение и агрегирование) выполняются кодом приложения, разумеется, без какой-либо оптимизации. Известны случаи, когда для формирования HTML-страницы приложение выполняло десятки тысяч запросов — в несколько раз больше, чем размер генерируемой страницы в байтах.

Все это приводит к тому, что части приложения, реализация которых с применением каркаса наименее эффективна, приходится переделывать с использованием запросов на SQL, забывая о правилах хорошего тона, рекомендуемых учебниками по программной инженерии. Поскольку обычно проблемы производительности возникают в частях приложения с наиболее сложными функциями, говорить о качестве кода не приходится. Получаемый при этом результат нельзя считать удовлетворительным и с точки зрения администратора базы данных, т. к. настройка запросов, собираемых динамически из констант, рассеянных по коду приложения, оказывается трудно выполнимой.

Можно сказать, что применение каркасов оправдано для быстрой реализации несложных приложений или частей приложений, для которых производительность не критична. Если ожидается, что проектируемая система будет использоваться с большой интенсивностью и, следовательно, создавать большую нагрузку, то целесообразно применить более сложные методы, обеспечивающие высокую эффективность работы системы.

7.4. Кеширование данных

Применение каркасов, как правило, приводит к тому, что приложение выполняет очень много небольших запросов. Это создает повышенную нагрузку на вычислительную сеть, потому что каждый из запросов выполняется синхронно. В результате приложение подавляющую часть времени ожидает результаты выполнения запросов, несмотря на крайне низкую загруженность сервера БД. Другими словами, обе компоненты (приложение и сервер) практически все

время находятся в состоянии ожидания. Единственным радикальным решением проблемы является использование сложных запросов. Обычно это приводит к сокращению времени выполнения приложения на 2–3 порядка, однако требует существенных затрат труда. Поэтому на практике применяются приемы, не решающие основную проблему, но несколько снижающие отрицательные последствия. Зачастую при этом создаются новые проблемы.

Одним из таких приемов является кеширование на уровне приложения.

Современные высокопроизводительные СУБД широко используют самые разные виды кеширования, причем использование этих средств не требует никаких усилий со стороны программистов приложений. Использование кеша БД означает, что при выборке данные будут по возможности браться из буферного кеша, а не считываться с дисков, ранее выполнявшиеся запросы не обязательно будут повторно оптимизироваться, а результаты их выполнения сохраняются для возможного повторного использования. Однако, как правило, сервер баз данных физически расположен на отдельном компьютере, и кеширование на стороне БД не может снизить нагрузку на вычислительную сеть. Поэтому для приложения, выполняющего огромное количество мелких запросов, время ожидания ответа не может существенно сократиться.

Сэкономить это время помогает кеш на уровне приложения. Как правило, программисты не уделяют достаточного внимания вопросам кеширования, считая, что кеширование можно добавить потом, если возникнут проблемы с производительностью. Часто это связано с непониманием принципов работы кеша и приводит к ошибкам в приложении.

Приложение работает с графом объектов, имеющим сложную структуру. Один и тот же объект может быть выбран из базы данных разными способами, и это может привести к появлению дубликатов: нескольких объектов с одинаковыми идентификаторами и разными состояниями. Поэтому для корректной работы приложения необходимо либо использовать сложные запросы, либо следить за идентичностью всех загружаемых объектов на уровне приложения, и часто эта задача делегируется кешу.

Возникает вопрос: что произойдет, если разные пользовательские сеансы будут обращаться к одним и тем же объектам. Задача обеспечения корректности полностью решается средствами управления транзакциями на уровне СУБД, однако при использовании кеширования в приложении возникает необходимость решать эти задачи заново.

Существуют различные стратегии параллельного доступа к кешированным данным.

Транзакционная. Такой кеш связан с транзакцией и гарантирует, что состояние объектов является актуальным и обновления объектов в одной транзакции проводятся атомарно.

Чтение-запись. При изменении объекты блокируются в кеше, а при фиксации транзакции в БД блокировки снимаются. Если другая транзакция пытается получить заблокированный объект, то считается, что объекта нет в кеше, и запрос перенаправляется к БД, которая отвечает в зависимости от установленного уровня изоляции транзакции.

Нестрогое чтение-запись. Согласованность между кешем и БД не поддерживается. Данные могут быть обновлены несколькими транзакциями без гарантии результата. После фиксации транзакции данные снова считываются из базы данных.

Только чтение. Предполагается, что данные не изменяются, и поэтому в кеше они не обновляются. Такую стратегию можно использовать для справочных данных.

Каркасы объектно-реляционных отображений обычно поддерживают два типа кеширования, дублируя функции СУБД:

Сеансовый кеш, или кеш первого уровня. Такой кеш является транзакционным. Выбранные во время сеанса объекты помещаются в кеш и при повторной выборке возвращаются из кеша без обращения к БД. Гарантируется, что если во время одного сеанса объект будет запрошен два раза, то вернется один и тот же объект приложения.

Разделяемый кеш, или кеш второго уровня. Этот кеш доступен всем клиентам приложения. При выборке объекта каркас проверяет, не находится ли этот объект в кеше первого уровня, а затем, если объект не найден, запрашивает кеш второго уровня. Этот кеш не обязательно является транзакционным и может содержать устаревшие данные. Для такого кеша нужно явно указать стратегию параллельного доступа и стратегию, по которой данные устаревают и удаляются из кеша.

Каркасы объектно-реляционных отображений поддерживают сеансовый кеш по умолчанию для всех объектов, а для кеша второго уровня нужно явно указать классы, которые будут кешироваться.

Кеш приложения может работать не только на уровне объектов, но и на уровне результатов запросов. В случае кеширования запросов ключом является запрос со всеми параметрами. Как правило, в кеше хранятся не сами выбранные объекты, а их идентификаторы, поэтому кеш запросов используется вместе с кешем второго уровня. Если в кеше второго уровня не будет нужных объектов, то они будут выбираться из БД по одному, что плохо влияет на производительность.

При использовании кеширования могут возникнуть следующие проблемы:

- если базу данных обновляют другие приложения, то данные в кеше становятся некорректными, поскольку кеш не знает об этих изменениях;
- при загрузке большого количества объектов может не хватить оперативной памяти;
- если ограничения доступа к данным реализуются на уровне БД, то те же самые ограничения должны быть реализованы на уровне кеша (т. е. функции СУБД должны быть продублированы).

Правильно настроенный механизм кеширования эффективно работает для одного приложения и в пределах одного сервера приложений. При использовании нескольких серверов приложений задача синхронизации кешированных данных становится весьма сложной.

7.5. Взаимодействие с базой данных

Любой каркас генерирует запросы на языке SQL и передает их для выполнения на сервер СУБД через предназначенный для этого интерфейс. Любое приложение — как построенное на основе каркаса, так и без него — может формировать запросы SQL и передавать их для выполнения через такой же интерфейс.

Рассмотрим способы взаимодействия приложения с базой данных без использования каркасов объектно-реляционных отображений.

7.5.1. Параметры запросов

Способ оформления и передачи оператора SQL из приложения на сервер БД зависит от языка программирования и от типа интерфейса, но в любом случае

в конечном итоге запрос передается на сервер базы данных в виде текстовой строки, возможно, с параметрами.

Текстовое представление оператора, переданное на сервер, подвергается предварительной обработке, зачастую довольно сложной. *Этап подготовки* включает синтаксический анализ и проверку на соответствие схеме базы данных, оптимизацию и генерацию кода. После подготовки следует *этап интерпретации (выполнения)*, на котором полученный код исполняется с использованием параметров оператора и полученные результаты оформляются для передачи в клиентскую программу.

Обычно параметры могут задавать значения для операций поиска в базе данных. В этом случае оператор будет находить разные объекты в базе данных в зависимости от значений переменных программы, переданных в качестве параметров оператора.

Этапы подготовки и интерпретации оператора можно выполнять отдельно друг от друга. Это может быть полезно, если один и тот же оператор предполагается использовать с несколькими разными наборами значений параметров. Альтернативный метод состоит в совмещении подготовки и выполнения в одном обращении к СУБД.

Выбор между непосредственным выполнением операторов и предварительной подготовкой операторов зависит от многих факторов. Использование подготовленных операторов при их многократном выполнении позволяет исключить затраты на повторную подготовку оператора. С другой стороны, при непосредственном выполнении оптимизатору известны значения всех параметров оператора, что потенциально дает возможность генерировать разные планы. Это может быть важно при неравномерном распределении значений атрибутов, на которые заданы условия фильтрации. Заметим, что далеко не каждый оптимизатор эту возможность использует.

С другой стороны, при непосредственном выполнении запросы, отличающиеся только значениями констант, оказываются разными, что препятствует их кешированию (на сервере базы данных). Для того чтобы повысить эффективность кеширования запросов, некоторые СУБД могут заменять константы, заданные в запросе, на параметры. В этом случае новый запрос, отличающийся только значениями констант, будет найден среди ранее выполненных, т. е. фактически непосредственное выполнение не будет отличаться от выполнения подготовленного запроса.

В некоторых системах, в том числе в PostgreSQL, в кеш обычно заносится не готовый для выполнения план, а только результат синтаксического разбора. В этом случае оптимизатор может генерировать различающиеся планы в зависимости от фактических значений параметров подготовленного запроса, поскольку во время оптимизации значения параметров уже известны.

В отличие от запросов, полученных конкатенацией строк, содержащих части запроса и значения параметров, применение параметризованных (подготовленных) запросов исключает возможность их изменения при передаче значений параметров. Эта особенность важна для предотвращения атак типа внедрения SQL-кода.

После выполнения оператора необходимо получить результаты в переменные программы-клиента. Многие операторы (например, SELECT и любые операторы обновления, содержащие предложение RETURNING) возвращают множество объектов, удовлетворяющих критериям поиска. Для того чтобы получить результаты полностью, необходимо выполнить цикл, в теле которого обрабатывается очередная строка результата.

7.5.2. Унифицированные средства взаимодействия

Обычно для доступа приложения к базе данных применяются унифицированные драйверы (библиотеки), которые устанавливают соединение с сервером БД, посылают SQL-запросы в текстовом формате и возвращают приложению выбранные данные. Унифицированные драйверы имеют интерфейс, который не зависит от конкретной СУБД и реализует ограниченное подмножество SQL. Некоторые драйверы допускают также использование любых возможностей SQL, реализованных в СУБД. Способность работы с разными СУБД может быть полезна для «коробочных» продуктов, но не для приложений, разрабатываемых для конкретных информационных систем.

Использование запросов в виде текста может вызывать проблемы, если игнорируется принцип независимости данных и программ. В случае изменения схемы БД приложение об этом не узнает и будет посылать запросы, написанные для старой версии схемы. Если СУБД не поймет запрос, сервер вернет ошибку, а драйвер выбросит исключение. Драйверы поддерживают и вызов хранимых процедур, имена которых также задаются в текстовом виде.

Драйвер возвращает данные, полученные при выполнении запроса, в виде коллекции, состоящей из примитивных типов. Это ограничение вызвано тем,

что широко распространенные стандарты на интерфейс драйверов (например, ODBC и JDBC) основаны на устаревшем стандарте SQL 92. Преобразование полученного набора данных в соответствующие объекты должно проводиться в коде приложения. Если СУБД (например, PostgreSQL) допускает хранение и возврат сложных типов, таких как массивы, драйверы могут возвращать и такие типы данных, но это обеспечивается не каждым драйвером.

Существуют каркасы, которые вместо объектно-реляционных отображений устанавливают соответствие между объектами приложения и SQL-запросами. В этом случае SQL-запросы пишутся в коде приложения, а каркас преобразует полученные данные в объекты.

Драйверы баз данных имеет смысл использовать, когда приложение строится над уже имеющейся базой данных или если производительности SQL, который генерирует каркас объектно-реляционных отображений, недостаточно.

7.5.3. Интерфейс PostgreSQL для приложений

Возможности СУБД PostgreSQL, предоставляемые программам-клиентам, отражены в клиент-серверном протоколе. Наиболее полно эти возможности реализованы библиотекой libpq, которая может непосредственно использоваться в программах, написанных на языке C, а также является основой для интерфейсов многих других языков программирования, в том числе C++, Python, Perl, хотя имеются и альтернативные реализации этого протокола.

Чтобы начать работать с сервером, клиент должен установить соединение, указав необходимые параметры (в частности, имя пользователя и название базы данных) и пройдя аутентификацию.

Простой способ выполнения оператора состоит в передаче серверу текстовой строки с SQL-запросом без параметров. В ответ сервер возвращает клиенту результат выполнения полностью, сколько строк он бы ни содержал.

Расширенный протокол позволяет разбить выполнение команды на несколько этапов: подготовка (возможно, параметризованного) запроса, привязка фактических значений параметров, выполнение. Для операторов, возвращающих данные, можно получить информацию об именах и типах результирующих колонок результата. При выполнении оператора клиент может получать результат построчно (такая возможность обычно представляется в языках программирования понятием *курсора*).

Часто клиентские библиотеки неявно используют расширенный протокол, поскольку процедурная обработка множества объектов, включенных в результат, требует цикла по этим объектам.

Обычно обмен сообщениями между клиентом и сервером происходит синхронно: клиент посылает следующую команду, после того как обработает результаты предыдущей. Однако протокол PostgreSQL поддерживает и асинхронную отправку команд серверу, что позволяет клиенту обрабатывать результаты одной команды, в то время как сервер выполняет другой запрос. Кроме libpq, такая возможность поддерживается, например, драйвером для Erlang.

7.6. Некоторые общие задачи

В этом разделе мы рассмотрим некоторые типичные задачи, которые приходится решать почти в каждом проекте, независимо от его предметной области.

7.6.1. Ограничение доступа к данным

Требования к информационной системе должны содержать правила доступа к данным. Эти правила определяют для каждого конкретного пользователя, какие действия ему разрешено совершать. В настоящий момент наиболее распространенной является ролевая модель ограничения доступа, в которой роли назначаются пользователям, а разрешения — ролям. При этом у пользователя может быть много ролей, а у роли много разрешений.

Что понимать под разрешением, зависит от конкретной задачи. Большинство задач требует детального контроля доступа к отдельным объектам. Современные базы данных, в том числе и PostgreSQL, предоставляют возможности определить правила доступа на уровне строк.

Для проектирования модели доступа к данным нужно решить, где будет контролироваться доступ: на уровне СУБД или на уровне приложения. В случае если контроль доступа осуществляется на уровне СУБД, можно использовать встроенную поддержку безопасности. Однако такой подход приводит к некоторым осложнениям при проектировании приложения.

- Пользователи приложения должны быть также пользователями базы данных. Это может оказаться неприемлемым в системах, в которых предварительная регистрация не требуется (например, в интернет-магазинах),

или в тех случаях, когда количество пользователей слишком велико. Задача регистрации нового пользователя приложения включает в этом случае задачу создания пользователя СУБД, т. е. задачу администратора базы данных, а не приложения, а добавление строки в таблицу пользователей не рассматривается как задача администрирования.

- Исключается использование одного сеанса для обслуживания нескольких пользователей. В 90-е гг. прошлого века было обнаружено, что подключение к СУБД является весьма дорогостоящей операцией. Поэтому на серверах приложений стали реализовывать *пул соединений*, повторно использующий заранее открытые соединения для обслуживания разных пользователей приложения.

  В результате усовершенствования СУБД и изменения мощностей компьютеров относительная стоимость создания нового соединения значительно снизилась, но пулы соединений по-прежнему применяются.

- Ограничения доступа задаются на уровне таблиц, строк и операций БД. Как правило, требования к модели доступа определяются в терминах предметной области, которая плохо отображается на базу данных. Например, право *заказать билет* нужно будет отобразить в операции вставки над соответствующими представлениями или таблицами.

- Как правило, ограничить доступ к базе данных недостаточно. Пользовательский интерфейс для пользователей с разными правами тоже выглядит по-разному.

Более распространенным является программирование ограничений доступа на уровне приложений. В этом случае все соединения к серверу могут происходить в контексте одного пользователя.

Такой подход также имеет некоторые недостатки:

- невозможно проводить аудит на уровне СУБД, т. к. неизвестно, какой пользователь приложения совершил операцию над данными;

- объектно-ориентированные языки не имеют встроенной поддержки ограничения доступа к объектам — придется использовать какую-либо библиотеку или реализовывать эту поддержку в коде приложения.

Возможны комбинированные решения, например роль приложения может соответствовать пользователю базы данных.

Какой бы подход ни был выбран, стоит пользоваться *принципом наименьших привилегий*: все пользователи должны иметь минимальный уровень доступа, необходимый для выполнения их задач.

7.6.2. Поддержка многоязычности

Информационные системы часто должны поддерживать несколько языков не только на уровне пользовательского интерфейса, но и на уровне данных. Это означает, что для некоторых атрибутов надо хранить несколько значений, по одному для каждого из поддерживаемых языков. В примере со студентами и дисциплинами мы можем хранить на нескольких языках названия курсов.

Существуют различные варианты реализации многоязычности на уровне базы данных.

Отдельный атрибут для каждого языка

В этом варианте для атрибута будет создано столько колонок, сколько мы поддерживаем языков. Этот подход показан на рис. 7.6.1.

courses

course_no	title_ru	title_en	credits
CS301	Базы данных	Databases	5
CS305	Анализ данных	Data analysis	10

Рис. 7.6.1. Атрибут для каждого языка

Недостатком такого варианта является необходимость модификации схемы базы данных и приложения при добавлении нового языка, а преимуществом — возможность непосредственного извлечения значения атрибута без обращения к каким-либо справочным таблицам.

Таблица с переводами для каждой сущности

Для каждой сущности, поддерживающей многоязычность, создается таблица с переводами, которая ссылается на главную таблицу. В примере, показанном на рис. 7.6.2, атрибут course_no является внешним ключом.

courses

course_no	credits
CS301	5
CS305	10

courses_translations

course_no	title	language
CS301	Базы данных	ru
CS301	Databases	en
CS305	Анализ данных	ru
CS305	Data Analysis	en

Рис. 7.6.2. Таблицы с переводами

В этом случае не требуется создавать новых структур данных при добавлении языков, а для получения данных достаточно одной операции соединения. Такой подход довольно сложен при использовании каркасов объектно-реляционных отображений, поскольку напрямую ими не поддерживается.

Общая таблица для хранения всех переводов

В этом варианте мы создаем одну общую таблицу для хранения переводов любых других таблиц. Поэтому в примере на рис. 7.6.3 таблица переводов названа просто translations, а связь осуществляется с помощью суррогатных идентификаторов.

courses

course_no	title_trans_id	credits
CS301	1	5
CS305	2	10

translations

trans_id	translation	language
1	Базы данных	ru
1	Databases	en
2	Анализ данных	ru
2	Data analysis	en

Рис. 7.6.3. Общая таблица переводов

Это наиболее гибкий подход. При его использовании также не нужно менять схему при добавлении новых языков, нет проблем с разреженными таблицами, и получается более правильное отображение в объекты.

Использование слабоструктурированных данных

Многие СУБД имеют специальные типы для поддержки слабоструктурированных данных. Так, PostgreSQL поддерживает форматы JSON и XML. Возможности этих средств в системе PostgreSQL рассматриваются далее в главе 8, а здесь мы ограничимся примером, показывающим, как можно применить такие средства для поддержки нескольких языков. Этот метод использован в демонстрационной базе данных PostgreSQL.

Информация об аэропортах хранится в таблице airports_data:

```
demo=# \d airports_data
                Table "bookings.airports_data"
    Column    |     Type     | Collation | Nullable | Default
--------------+--------------+-----------+----------+---------
 airport_code | character(3) |           | not null |
 airport_name | jsonb        |           | not null |
 city         | jsonb        |           | not null |
 coordinates  | point        |           | not null |
 timezone     | text         |           | not null |
Indexes:
    "airports_data_pkey" PRIMARY KEY, btree (airport_code)
...
```

Обратим внимание на то, что столбец airport_name имеет тип jsonb. Значения этого столбца содержат названия аэропорта на нескольких языках:

```
demo=# SELECT airport_name
FROM airports_data
LIMIT 10;
                          airport_name
-----------------------------------------------------------------
 {"en": "Yakutsk Airport", "ru": "Якутск"}
 {"en": "Mirny Airport", "ru": "Мирный"}
 {"en": "Khabarovsk-Novy Airport", "ru": "Хабаровск-Новый"}
 {"en": "Yelizovo Airport", "ru": "Елизово"}
 {"en": "Yuzhno-Sakhalinsk Airport", "ru": "Хомутово"}
 {"en": "Vladivostok International Airport", "ru": "Владивосток"}
 {"en": "Pulkovo Airport", "ru": "Пулково"}
 {"en": "Khrabrovo Airport", "ru": "Храброво"}
 {"en": "Kemerovo Airport", "ru": "Кемерово"}
 {"en": "Chelyabinsk Balandino Airport", "ru": "Челябинск"}
(10 rows)
```

Приложения, работающие с этой базой данных, должны использовать представление, которое выбирает название аэропорта на языке, установленном по умолчанию:

```
demo=# \d+ airports
...
View definition:
 SELECT ml.airport_code,
    ml.airport_name ->> lang() AS airport_name,
    ml.city ->> lang() AS city,
    ml.coordinates,
    ml.timezone
   FROM airports_data ml;
```

Функция lang вырабатывает код языка, установленный в конфигурационном параметре bookings.lang (устанавливается по умолчанию в значение ru, но может быть изменен). При выборке данных из этого представления получаем:

```
demo=# SELECT airport_name
FROM airports
LIMIT 10;
  airport_name
-----------------
 Якутск
 Мирный
 Хабаровск-Новый
 Елизово
 Хомутово
 Владивосток
 Пулково
 Храброво
 Кемерово
 Челябинск
(10 rows)
```

7.7. Настройка

Настройкой называется комплекс мер, направленных на приведение прикладной системы в соответствие с требованиями по производительности (в первую очередь по времени отклика и по пропускной способности), не изменяющий основных функций системы. Некоторые из таких мер часто называют «оптимизациями», однако в контексте курса по системам управления базами данных оптимизацией называется автоматический выбор плана выполнения запроса, который осуществляется оптимизатором и обсуждается более детально в главе 12 второй части курса.

В идеальном мире настройка рассматривается как важная составная часть всех этапов жизненного цикла прикладной системы, начиная с определения требований к системе, однако в реальности о настройке вспоминают только тогда, когда показатели производительности уже работающей системы оказываются неудовлетворительными.

Меры, которые можно считать элементами настройки прикладной системы, конечно, различны на разных фазах ее жизненного цикла. Очевидно, что точное определение требований по производительности создает условия для последующей настройки на всех этапах: выбор архитектуры и методологии разработки существенно влияет на все характеристики системы, определение как логической структуры данных, так и структуры хранения влияет на эффективность манипулирования данными, и т. д.

Меры, направленные на повышение производительности, могут применяться на различных уровнях.

Изменение (расширение) конфигурации оборудования обычно рассматривается как простой метод наращивания производительности, однако далеко не всегда масштабируемость позволяет решить проблемы производительности. При этом, как правило, удается повысить пропускную способность, но очень редко — улучшить время отклика. Неудачно написанный программный код может компенсировать любое увеличение мощности оборудования.

Выбор параметров сервера включает конфигурацию как операционной системы, так и сервера базы данных и является одной из основных задач администратора баз данных. Параметры конфигурации позволяют изменять размеры областей оперативной памяти, используемых сервером базы данных, количество процессов, выполняющих запросы пользователей, стратегии копирования и восстановления в случае отказов и т. п.

Управление схемой базы данных включает прежде всего управление хранением данных: размещение табличных пространств на устройствах; размещение таблиц, индексов и других объектов в табличных пространствах; выбор параметров, управляющих размещением данных в таблицах, и т. д. Сюда же включаются выбор материализованных представлений и другие модификации логической схемы базы данных.

Улучшение запросов. Как правило, оптимизатор базы данных вполне успешно выполняет свои функции и создает планы выполнения запросов, близкие

к оптимальным. Однако в некоторых случаях запрос может содержать избыточные операции, или по каким-либо другим причинам план, построенный оптимизатором, оказывается неудовлетворительным. В подобных ситуациях требуется ручная настройка запроса, обычно состоящая в переписывании его в другую эквивалентную форму, и зачастую могут потребоваться изменения в структуре хранения данных (например, создание дополнительных индексов или материализованных представлений).

Реструктуризация кода приложения обычно рассматривается как дорогостоящее и потенциально опасное действие, однако многие виды неэффективности приложения вызываются именно дефектами кода приложения и не могут быть полностью компенсированы на других уровнях. В частности, одной из наиболее частых причин плохого времени отклика является программный код, выполняющий слишком много слишком мелких запросов.

Наиболее результативной настройка будет только в случае ее проведения на всех этапах разработки и на всех уровнях от оборудования до программного кода приложения, с учетом как особенностей применяемых систем, так и функций прикладной системы и требований к ней.

Подробно вопросы настройки прикладных систем обсуждаются в главе 20 второй части книги.

7.8. Проектирование декларативных запросов

Операторы SQL, автоматически генерируемые средствами объектно-реляционных отображений, зачастую проигрывают по качеству запросам, спроектированным вручную. Поэтому при реализации прикладных систем, в которых необходима массовая обработка большого количества данных с высокими требованиями к производительности, целесообразно отделение функций манипулирования данными от остальных функций приложения. При этом доступ средств объектно-реляционных отображений к данным осуществляется через интерфейс виртуальных отношений, которые могут быть как хранимыми таблицами, так и реализованы другими способами, в том числе параметризованными запросами, представлениями и процедурами базы данных.

Реализованная в PostgreSQL полноценная поддержка отношений в качестве возвращаемых функциями значений является мощным инструментом. Преимуществами пользовательских функций, возвращающих отношения, по сравнению с представлениями, являются:

- возможность передачи параметров, задающих условия на формирование результата;
- возможность формирования динамического кода SQL в зависимости от переданных параметров.

Необходимо подчеркнуть, что разработка функций базы данных требует относительно высокой квалификации, в том числе необходимо глубокое понимание того, каким образом оптимизируются и выполняются запросы в СУБД. Вся существенная работа с данными должна описываться на декларативном языке (т. е. на SQL), хотя возможности языков, на которых реализуются функции базы данных, могут провоцировать интенсивное использование имеющихся в них императивных средств.

Проектирование и разработка декларативных запросов существенно отличаются от разработки программного кода на императивном языке. Декларативное программирование предполагает, как и следует из названия, задание условий и требований, которым должен удовлетворять результат вычислений, но не конкретный алгоритм вычисления.

Однако этого недостаточно. Чтобы возможности СУБД использовались наилучшим образом, необходимо учитывать особенности реляционной модели данных: ориентацию на ассоциативный доступ и массовую обработку. Неформально можно сказать, что для получения высококачественного декларативного описания необходимо «думать в терминах множеств». Это выражение вынесено в заголовок книги [15], в которой детально обсуждаются методы конструирования запросов.

Хорошими руководствами по проектированию запросов и применению SQL являются также книги [27] и [14] (русский перевод [70]).

7.9. Итоги главы

В этой главе кратко характеризуются различные подходы, методологии и инструменты разработки приложений, а также различные варианты определения отображений между объектными моделями приложений и объектно-реляционными моделями баз данных. Кратко представлена методика разработки сложных запросов и сформулирована задача настройки приложений баз данных.

7.10. Упражнения

В демонстрационной базе поддержка нескольких языков реализована с использованием средств JSON. В следующих упражнениях требуется реализовать такую поддержку с помощью других вариантов, рассмотренных в этой главе.

Упражнение 7.1. Создайте новую схему демонстрационной базы таким образом, чтобы названия населенных пунктов хранились в соответствии с выбранным вами способом организации многоязычности. Данные должны быть представлены как минимум на русском и английском языках. Предусмотрите возможность расширения списка используемых языков.

Упражнение 7.2. Мигрируйте все данные из старой схемы демонстрационной базы в новую из предыдущего упражнения. Добавьте названия населенных пунктов на каких-либо других языках.

Упражнение 7.3. Напишите запрос, для каждого города показывающий количество пассажиров, прилетающих в него из Москвы в какой-нибудь определенный день. Запрос должен выдавать названия населенных пунктов на английском языке.

Упражнение 7.4. Реализуйте функцию покупки билета. С точки зрения пассажира покупка состоит из двух этапов. На первом этапе система выдает список возможных перелетов для указанных параметров: пункты отправления и назначения, количество мест, дата вылета. На втором — выбранный пассажиром вариант оформляется в виде бронирования, и пассажиру возвращается номер бронирования.

Попробуйте различные способы: с помощью запросов SQL, с применением хранимых функций, на основе использования каркаса.

Глава 8

Расширения реляционной модели

В этой главе обсуждаются расширения традиционных технологий применения систем, основанных на модели данных SQL, вплоть до попыток полного отказа от функциональности языка запросов. Зачастую при обсуждении подобных расширений принято отождествлять реляционную модель данных с комплексом технологий, основанных на подмножестве SQL, однако мы будем различать эти понятия.

8.1. Ограниченность реализаций SQL

Напомним, что ранние реализации реляционной модели данных и вместе с ними ранние версии языка SQL предусматривали только простые скалярные типы данных в качестве значений атрибутов: числа, символьные строки, даты и время и т. п. Как только началось относительно широкое применение подобных систем (во второй половине 80-х гг.), стало ясно, что модели и структуры данных, предоставляемые подобными системами, слишком ограничительны для целого ряда классов приложений.

В частности, для реализации систем автоматизированного проектирования (САПР, CAD) необходимо организовать хранение нескольких версий проекта. За время существования проекта может быть всего несколько десятков версий, однако каждая из них представляет собой сложный объект, объем которого может достигать значительных размеров. Для поддержки таких систем было введено понятие *вложенных отношений* или *отношений не в первой нормальной форме* (nested relations, non-first normal form, NFNF, NF^2). В таких отношениях значения атрибутов могут быть таблицами или отношениями (возможно, тоже содержащими вложенные таблицы).

Вложенные отношения были достаточно детально изучены теоретиками, однако до массового применения этой модели дело не дошло. Заметим, что структура данных вложенного отношения может быть эквивалентно отображена на

обычные (плоские) отношения, поэтому речь идет скорее о логической структуре, приближенной к структуре хранения данных, чем о действительно новой модели данных. Напомним, что теоретическая реляционная модель не занимается вопросами организации хранения данных.

Несколько более успешной оказалась идея объектно-ориентированных СУБД, появившихся в период бурного развития объектно-ориентированных языков программирования и методологий объектно-ориентированного программирования.

Модель данных объектно-ориентированных СУБД строится как совокупность классов или типов, описывающих как состояние (атрибуты), так и поведение (методы) объектов. Для описания взаимосвязей между объектами или их множествами используются явные или неявные указатели или ссылки, что, по существу, означает преимущественное применение навигации для поиска объектов в базе данных.

Ожидалось, что объектно-ориентированные СУБД вытеснят все остальные модели баз данных, однако фактически объектно-ориентированные системы смогли занять весьма скромную долю рынка СУБД и применялись для очень ограниченного круга прикладных областей, в которых нужны базы данных относительно небольшого объема, но требующие значительных вычислительных мощностей для обработки. При таких условиях размещение кода методов в базе данных оказывается вполне оправданным, т. к. это сокращает до минимума накладные расходы на передачу данных между местом хранения и местом, где фактически выполняются вычисления.

Применение объектно-ориентированных СУБД в других случаях оказалось неоправданным. В качестве причин, которые привели к такому результату, можно обозначить следующие:

- Навигационный поиск весьма эффективен для доступа к отдельным объектам, но оказывается неэффективным при массовой обработке большого количества данных.
- Для объектно-ориентированных моделей не удается построить полноценный высокоуровневый декларативный язык запросов.
- Модель данных оказывается привязанной к одному языку программирования (на котором записаны методы), что затрудняет использование базы данных другими приложениями. По существу, приходится отказаться от принципа независимости данных и программ.

Намного более успешным оказался альтернативный подход, получивший название объектно-реляционных СУБД. В настоящее время все высокопроизводительные системы, в том числе PostgreSQL, фактически относятся к этому классу систем. Центральным понятием в таких системах остается понятие таблицы, однако в качестве типов атрибутов могут использоваться объектные типы данных, как встроенные в систему, так и определенные в конкретной схеме базы данных.

В рамках такой системы можно использовать и сочетать самые различные способы организации данных и применять как высокоуровневый язык запросов, так и методы для обработки отдельных объектов данных.

Например, если схема базы данных содержит таблицу с единственным атрибутом и единственной строкой, но тип этого атрибута является сложным объектным типом, по существу, такая база данных становится объектно-ориентированной, хотя некоторые возможности применения высокоуровневых запросов при этом сохраняются.

Таблица, в которой типы некоторых атрибутов представляют собой коллекции, может использоваться для хранения отношений не в первой нормальной форме (NFNF), упомянутых выше.

Язык запросов объектно-реляционных систем пополняется средствами для доступа к составляющим сложных объектов, которые могут быть значениями атрибутов отношений.

Несмотря на использование близкой терминологии, объектные средства в системах управления базами данных зачастую существенно отличаются от аналогичных средств в языках программирования.

Заметим, что, хотя объектно-реляционные СУБД, безусловно, являются расширениями стандарта SQL, получившего очень широкое распространение, однако никак не рассогласуются с теоретической реляционной моделью данных, в которой в качестве значений атрибутов могут использоваться значения из произвольных доменов, которые реализуются объектными типами данных.

Напомним, что в теоретической реляционной модели домены могут определяться абстрактным типом данных, единственным требованием к которому является наличие предиката равенства значений из этого домена. Остальные операции, функции и предикаты на этом домене могут отражать особенности предметной области или применения. Так, можно различать домены весов, длин, денежных сумм, геометрических объектов определенного вида (точек, прямых, прямоугольников) и т. д.

Для того чтобы рассматривать сложные (нескалярные) значения атрибутов в теоретической реляционной модели, нужно ввести функции, преобразующие значение атрибута в структурный тип или коллекцию. Этот прием используется в объектно-реляционных СУБД для реализации, например, таких типов, как XML и JSON, рассматриваемых далее в данной главе.

В следующих разделах этой главы показано, каким образом объектные средства представлены в PostgreSQL. Некоторые из этих средств рассматриваются как отдельные расширения, однако концептуально это объектные типы с богатой семантикой и специфическими операциями.

Объектные возможности PostgreSQL в основном сводятся к определению новых структур данных (но не поведения, которое должно описываться функциями, определяемыми отдельно от описаний структур данных).

8.2. Реализация объектных расширений в PostgreSQL

Система PostgreSQL не предоставляет объектные расширения в виде какой-либо полной объектной модели данных. Вместо этого имеются возможности для создания высокоуровневых объектных средств на основе имеющихся конструкций. В этом разделе рассматриваются главным образом возможности определения сложных структур данных и типы данных, необходимые для представления объектов и связей между ними. Поведение объектов может описываться с помощью функций, которые могут быть написаны на языке SQL или на любом из поддерживаемых в PostgreSQL императивных языков программирования.

Можно сказать, что наиболее важной характеристикой PostgreSQL является расширяемость, т. е. возможность добавления новых типов данных, функций, операторов и даже индексных структур без изменения ядра системы PostgreSQL.

8.2.1. Наследование

Применение наследования в объектно-ориентированных языках программирования уже обсуждалось в главе 7. Здесь мы кратко охарактеризуем возможности представления наследования на уровне базы данных в системе PostgreSQL.

В PostgreSQL наследование можно использовать для определения таблиц. Это отличает PostgreSQL от стандарта SQL, в котором, начиная с издания 1999 г., предусмотрено наследование для типов. Если при определении таблицы указано, что она наследует из другой (родительской) таблицы, то в состав атрибутов определяемой таблицы включаются все атрибуты родительской таблицы. При этом в запросах можно указывать, требуется ли выбрать строки только из указанной таблицы или из указанной и всех ее дочерних таблиц. При вставке новых строк (INSERT) операция выполняется на указанной таблице независимо от того, участвует эта таблица в наследовании или нет.

Ограничения ссылочной целостности (PRIMARY KEY, FOREIGN KEY) и ограничения на уникальность значений (UNIQUE) применяются только к каждой из таблиц по отдельности. Из этого следует, что в иерархии наследования могут появляться, например, дублирующиеся значения, даже если заданы ограничения UNIQUE на каждой из таблиц иерархии. Эти особенности заметно снижают полезность наследования; документация рекомендует использовать его с осторожностью.

Можно заметить, что наследование таблиц в PostgreSQL предполагает горизонтальную фрагментацию, т. е. каждая таблица содержит лишь строки, не принадлежащие наследующим из нее таблицам, и содержит все атрибуты, логически входящие в эту таблицу. В противоположность этому теоретическая модель «сущность — связь» неявно предполагает вертикальную фрагментацию.

8.2.2. Определение типов данных

В системе PostgreSQL можно определять несколько разновидностей пользовательских типов данных.

Составной тип данных (запись, record) представляет собой структуру, состоящую из нескольких атрибутов (подобно определению строки таблицы). На самом деле каждая таблица определяет и составной тип, атрибуты которого соответствуют описанию этой таблицы.

Тип диапазона (range) задает интервал значений некоторого другого типа, для которого должно быть определено отношение полного упорядочивания. Встроенными диапазонными типами являются диапазоны для различных числовых типов (int4range, int8range, numrange) и интервалов времени (tsrange, tstzrange) и дат (daterange).

Перечисляемый тип (enum) соответствует перечислительным типам, которые могут быть определены во многих языках программирования. Такие типы могут принимать фиксированное в определении типа количество различных именованных значений.

Новые базовые типы могут определяться с помощью указания как внутреннего представления (через другие типы данных), так и функций, реализующих операции над этим типом и обеспечивающих возможность его эффективного использования (например, указания для оптимизатора запросов). Определение базовых типов требует некоторых знаний о внутреннем устройстве СУБД и представляет собой достаточно трудоемкий процесс, поэтому вряд ли целесообразно создавать базовые типы при разработке отдельных приложений.

8.2.3. Домены

Понятие *домена* в SQL отличается от понятия домена теоретической реляционной модели данных: это не абстрактный скалярный тип данных, а ранее определенный тип данных с дополнительными ограничениями на его значения.

8.2.4. Коллекции

Для представления вложенных коллекций в PostgreSQL можно использовать *массивы* (array), во многом похожие на массивы в языках программирования. Массивы состоят из элементов одного типа, которые идентифицируются целочисленными индексами, задающими их положение. В PostgreSQL массивы могут быть и многомерными.

Кроме обычных операций вырезки (выделения подмассива меньшего размера или отдельных элементов), определены операции, позволяющие склеивать несколько массивов в один (операция конкатенации и несколько аналогичных по назначению функций). Операции конкатенации особенно полезны для одномерных массивов, т. к. с их помощью легко реализовать списки.

Значения массивов можно записывать как константы, добавлять или изменять элементы массивов, а также формировать массивы из значений, извлекаемых из таблиц:

```
demo=# SELECT array_agg(airport_code)
FROM (SELECT * FROM airports LIMIT 5) air;
       array_agg
----------------------
 {AAQ,ABA,AER,ARH,ASF}
(1 row)
```

В этом запросе функция array_agg собирает в массив значения, находящиеся в разных строках таблицы, при этом количество элементов массива будет равно числу строк в отношении, полученном в результате выполнения подзапроса. Подзапрос в предложении FROM нужен только для того, чтобы размер результата был обозримым, никакого содержательного смысла ограничение числа строк не имеет. Аргументом функции может быть значение не только скалярного типа; так, можно построить массив из записей или многомерный массив.

Обратное преобразование массива во множество (таблицу) можно выполнить с помощью функции unnest:

```
demo=# SELECT * FROM unnest(ARRAY['AAQ','ABA','AER','ARH','ASF']);
 unnest
--------
 AAQ
 ABA
 AER
 ARH
 ASF
(5 rows)
```

Здесь в качестве источника данных использован массив, сконструированный из констант, но, конечно, допустимо и значение некоторой колонки таблицы.

Необходимо отметить, что массивы не могут рассматриваться как полноценная замена таблиц, потому что операции выборки данных из таблиц и некоторые операции реляционной алгебры реализуются более эффективно. Некоторые свойства данных, например ограничения целостности, нельзя определить для элементов массивов. Тем не менее массивы в PostgreSQL обеспечивают много полезных возможностей, в том числе реализацию вложенных отношений, возможности специализированных СУБД для обработки массивов (научных данных).

8.2.5. Указатели

Одна из важных особенностей объектных моделей — навигационный доступ к данным, который обычно поддерживается с помощью объектных указателей.

Как правило, предполагается, что значения указателей не могут изменяться и для них применяются целочисленные суррогаты, хотя, в принципе, в качестве указателей можно использовать любые типы данных. Связь между строками таблиц по первичному и внешнему ключам можно считать частным случаем применения объектных указателей. В других случаях значения объектных указателей могут быть собраны в массив.

Можно сказать, что объектный указатель — это не особый тип данных, а особый способ использования значений, хотя в объектных языках программирования указатели выделяются как отдельный тип.

Среди встроенных типов данных в PostgreSQL имеется несколько типов, которые могут использоваться как объектные указатели на различных уровнях. Один из этих типов — oid (object identifier) — используется в этом качестве самой СУБД, однако из-за его малого размера (4 байта) в прикладных системах рекомендуется применять другие типы, например длинные целые числа.

8.3. Функции

Наиболее мощные объектные возможности PostgreSQL реализуются при помощи аппарата функций, хранимых и выполняемых на сервере базы данных. Такие функции могут быть написаны на различных языках программирования, в том числе на SQL, однако наиболее развитые возможности манипулирования данными, хранимыми в БД, предоставляются в языке PL/pgSQL.

В системе PostgreSQL функции могут принимать аргументы и вырабатывать результат любых типов, как встроенных, так и пользовательских, включая скалярные типы, записи, массивы и множества (таблицы).

Как и любой мощный инструмент, процедурные языки могут быть опасными. Непродуманное использование процедурных языков может привести к серьезной потере вычислительной эффективности. Это связано с тем, что оптимизатор запросов, как правило, не имеет информации о стоимости выполнения функции и поэтому не может правильно оценить затраты ресурсов, необходимые для ее выполнения. Другая возможная причина снижения эффективности при необдуманном применении функций связана с тем, что запросы, содержащиеся в теле функции, оптимизируются и выполняются отдельно. Это может привести к эффекту «слишком много слишком мелких запросов».

Если функция написана на языке SQL, обработчик запросов PostgreSQL может в некоторых случаях подставлять тело такой функции в запрос, содержащий ее вызов, и тогда потеря эффективности не происходит.

Применение языка PL/pgSQL и других процедурных языков, в том числе расширение функциональности СУБД посредством создания новых базовых типов, более детально обсуждается в главах 16 и 17 второй части этого курса.

8.4. Слабоструктурированные данные: JSON

Ограниченность систем типов данных, реализованных в ранних СУБД, использующих SQL, привела к тому, что развитые на их основе методологии проектирования и технологии применения СУБД оказались слишком ограничительными для многих классов приложений. Зачастую эту ограниченность связывают с самой реляционной моделью данных, поэтому средства для преодоления этих ограничений называют расширениями реляционной модели, хотя фактически они только снимают ограничения технологий, построенных на ее основе.

Одним из таких ограничений является раздельное хранение схемы базы данных и самих данных. Такое разделение оказывается неудобным для приложений, использующих внешние источники данных (например, получаемые из информационных ресурсов, доступных в интернете). В подобных случаях схема может быть опубликована не полностью, и поэтому фактическая структура данных может отличаться от описанной в известной части схемы. Важной особенностью таких данных является также присутствие неструктурированных данных (например, текстов на естественном языке, изображений и т. п.). Подобные данные принято называть *слабоструктурированными* (semi-structured). Часто по-русски используется также термин *полуструктурированные*.

Для передачи таких данных по вычислительным сетям используются форматы XML и JSON, в которых имена атрибутов указываются вместе со значениями этих атрибутов. Это означает, что схема таких данных является динамической. Отметим, что при использовании таких ничем не ограниченных схем разработка приложения существенно усложняется, т. к. функции по управлению схемой, обычно выполняемые в СУБД, перекладываются на приложение. Кроме этого, приложение должно быть готово к обработке различных исключительных ситуаций, которые не могли бы возникать при наличии ограничений, обрабатываемых СУБД. Наиболее очевидными исключительными ситуациями

могут быть отсутствие элементов данных (атрибутов), необходимых для работы приложения, дублирование атрибутов, которые приложение считает уникальными, различие в именовании атрибутов и т. п.

Тем не менее обработка данных в таких форматах необходима, и для того чтобы ее обеспечить, в составе PostgreSQL имеется тип xml для хранения данных в формате XML и два типа (json и jsonb) для хранения в формате JSON, отличающихся способом внутреннего представления в базе данных.

Как и все остальные типы, эти типы можно задавать для отдельных атрибутов таблиц. Если в таблице имеется всего один атрибут, например типа json, и всего одна строка, то, по существу, такая таблица хранит один документ и работа с этим документом происходит так, как в базах данных, ориентированных исключительно на хранение документов; при этом возможности SQL почти не используются. Если же таблица содержит и другие атрибуты (возможно, некоторые из них также слабоструктурированного типа) или несколько строк, то появляется возможность сочетать преимущества различающихся моделей данных при работе с одной базой данных.

В определении слабоструктурированных типов данных предусмотрены возможности конструирования значений этих типов из других, структурированных атрибутов объектов, хранящихся в базе данных, а также преобразования слабоструктурированных значений в структурированные с выделением значений отдельных атрибутов.

Следующий оператор вырабатывает результат, содержащий одну колонку типа json, при этом в каждой строке результата находятся данные только из одной строки исходной таблицы airports:

```
demo=# SELECT json_build_object(
    'code', airport_code,
    'name', airport_name)
FROM airports
LIMIT 8;
               json_build_object
-----------------------------------------------
 {"code" : "YKS", "name" : "Якутск"}
 {"code" : "MJZ", "name" : "Мирный"}
 {"code" : "KHV", "name" : "Хабаровск-Новый"}
 {"code" : "PKC", "name" : "Елизово"}
 {"code" : "UUS", "name" : "Хомутово"}
 {"code" : "VVO", "name" : "Владивосток"}
 {"code" : "LED", "name" : "Пулково"}
 {"code" : "KGD", "name" : "Храброво"}
(8 rows)
```

Вариант этого запроса собирает те же данные в один JSON-документ, представляющий собой массив (в смысле JSON) документов, полученных из отдельных строк таблицы с помощью функции агрегирования json_agg:

```
demo=# SELECT json_agg(
  json_build_object('code', airport_code, 'name', airport_name)
)
FROM (
  SELECT * FROM airports LIMIT 8
) air;
                                 json_agg
-----------------------------------------------------------------------
 [{"code" : "YKS", "name" : "Якутск"}, {"code" : "MJZ", "name" :
 "Мирный"}, {"code" : "KHV", "name" : "Хабаровск-Новый"}, {"code" :
 "PKC", "name" : "Елизово"}, {"code" : "UUS", "name" : "Хомутово"},
 {"code" : "VVO", "name" : "Владивосток"}, {"code" : "LED", "name" :
 "Пулково"}, {"code" : "KGD", "name" : "Храброво"}]
(1 row)
```

В следующем примере объект JSON преобразуется в таблицу пар ключ — значение:

```
demo=# SELECT *
FROM json_each('{"code" : "KVK", "name" : "Апатиты-Кировск"}');
 key  |       value
------+-------------------
 code | "KVK"
 name | "Апатиты-Кировск"
(2 rows)
```

Еще один пример показывает, каким образом объект JSON можно преобразовать в структуру, соответствующую строке таблицы:

```
demo=# SELECT *
FROM json_populate_record(
    NULL::airports,
    '{"airport_code" : "KVK",
      "name" : "Апатиты-Кировск",
      "city" : "Кировск"}'
);
 airport_code | airport_name |  city   | coordinates | timezone
--------------+--------------+---------+-------------+----------
 KVK          |              | Кировск |             |
(1 row)
```

При этом преобразовании:

- значения атрибутов, ключи которых совпадают с именами колонок таблицы, включаются в результат (airport_code и city);

- атрибуты, для ключей которых не нашлось колонки, выбрасываются (name);
- колонки, для которых не нашлось атрибутов объекта JSON, заполняются неопределенными значениями (airport_name, coordinates, timezone).

Если документ в формате JSON представляет собой массив объектов, можно преобразовать этот документ в таблицу, содержащую каждый элемент в отдельной строке. В следующем примере массив JSON записан в явном виде, но он может передаваться из приложений или храниться в качестве значений в столбцах таблиц типа json.

```
demo=# SELECT *
FROM json_populate_recordset(
  NULL::airports,
  '[{"airport_code" : "YKS", "airport_name" : "Якутск"},
    {"airport_code" : "MJZ", "airport_name" : "Мирный"},
    {"airport_code" : "KHV", "airport_name" : "Хабаровск-Новый"},
    {"airport_code" : "PKC", "airport_name" : "Елизово"},
    {"airport_code" : "UUS", "airport_name" : "Хомутово"},
    {"airport_code" : "VVO", "airport_name" : "Владивосток"},
    {"airport_code" : "LED", "airport_name" : "Пулково"},
    {"airport_code" : "KGD", "airport_name" : "Храброво"}]'::json
);
 airport_code |  airport_name   | city | coordinates | timezone
--------------+-----------------+------+-------------+----------
 YKS          | Якутск          |      |             |
 MJZ          | Мирный          |      |             |
 KHV          | Хабаровск-Новый |      |             |
 PKC          | Елизово         |      |             |
 UUS          | Хомутово        |      |             |
 VVO          | Владивосток     |      |             |
 LED          | Пулково         |      |             |
 KGD          | Храброво        |      |             |
(8 rows)
```

Структуру получаемого множества можно также задавать в самом запросе, а не ссылаться на заранее созданный тип записи:

```
demo=# SELECT *
FROM json_to_recordset(
  '[{"airport_code" : "YKS", "airport_name" : "Якутск"},
    {"airport_code" : "MJZ", "airport_name" : "Мирный"},
    {"airport_code" : "KHV", "airport_name" : "Хабаровск-Новый"},
    {"airport_code" : "PKC", "airport_name" : "Елизово"},
    {"airport_code" : "UUS", "airport_name" : "Хомутово"},
    {"airport_code" : "VVO", "airport_name" : "Владивосток"},
    {"airport_code" : "LED", "airport_name" : "Пулково"},
    {"airport_code" : "KGD", "airport_name" : "Храброво"}]'::json
) AS (airport_code text, airport_name text);
```

```
 airport_code |  airport_name
--------------+-----------------
 YKS          | Якутск
 MJZ          | Мирный
 KHV          | Хабаровск-Новый
 PKC          | Елизово
 UUS          | Хомутово
 VVO          | Владивосток
 LED          | Пулково
 KGD          | Храброво
(8 rows)
```

8.5. Слабоструктурированные данные: XML

Данные преобразуются из таблиц в формат XML с помощью функций, определенных стандартами XML и реализованными в PostgreSQL, а также рядом дополнительных функций, упрощающих такое преобразование.

Наиболее важными из стандартных функций для генерации XML являются:

- xmlelement — строит один элемент XML с заданным именем, значением и, возможно, атрибутами;
- xmlforest — строит список из нескольких элементов XML;
- xmlagg — агрегатная функция, объединяющая в один документ элементы, полученные из нескольких строк таблицы.

Например, рассмотрим следующий запрос:

```
demo=# SELECT f.flight_no,
  dep.airport_code dep,
  arr.airport_code arr
FROM ticket_flights tf
  JOIN flights f ON tf.flight_id = f.flight_id
  JOIN airports dep ON f.departure_airport = dep.airport_code
  JOIN airports arr ON f.arrival_airport = arr.airport_code
WHERE tf.ticket_no = '0005432369015'
ORDER BY f.scheduled_departure;
 flight_no | dep | arr
-----------+-----+-----
 PG0233    | VKO | BZK
 PG0649    | BZK | EGO
 PG0481    | EGO | AAQ
 PG0480    | AAQ | EGO
 PG0650    | EGO | BZK
 PG0237    | BZK | VKO
(6 rows)
```

Запрос возвращает все перелеты для одного из билетов в демонстрационной базе данных. Та же информация в формате XML может быть получена как таблица, содержащая единственную колонку типа xml:

```
demo=# SELECT xmlelement(
    name flight,
    xmlforest(
      dep.airport_code AS dep,
      arr.airport_code AS arr
    )
  )
FROM ticket_flights tf
  JOIN flights f ON tf.flight_id = f.flight_id
  JOIN airports dep ON f.departure_airport = dep.airport_code
  JOIN airports arr ON f.arrival_airport = arr.airport_code
WHERE tf.ticket_no = '0005432369015'
ORDER BY f.scheduled_departure;
                  xmlelement
-----------------------------------------------
 <flight><dep>VKO</dep><arr>BZK</arr></flight>
 <flight><dep>BZK</dep><arr>EGO</arr></flight>
 <flight><dep>EGO</dep><arr>AAQ</arr></flight>
 <flight><dep>AAQ</dep><arr>EGO</arr></flight>
 <flight><dep>EGO</dep><arr>BZK</arr></flight>
 <flight><dep>BZK</dep><arr>VKO</arr></flight>
(6 rows)
```

Для того чтобы вывести ту же информацию в виде одного документа, применим функцию агрегирования:

```
demo=# SELECT xmlagg(xmlelement(
    name flight,
    xmlforest(
      dep.airport_code AS dep,
      arr.airport_code AS arr
    )
  ) ORDER BY f.scheduled_departure)
FROM ticket_flights tf
  JOIN flights f ON tf.flight_id = f.flight_id
  JOIN airports dep ON f.departure_airport = dep.airport_code
  JOIN airports arr ON f.arrival_airport = arr.airport_code
WHERE tf.ticket_no = '0005432369015';
                                 xmlagg
-----------------------------------------------------------------------
 <flight><dep>VKO</dep><arr>BZK</arr></flight><flight><dep>BZK</dep>
 <arr>EGO</arr></flight><flight><dep>EGO</dep><arr>AAQ</arr></flight
 ><flight><dep>AAQ</dep><arr>EGO</arr></flight><flight><dep>EGO</dep
 ><arr>BZK</arr></flight><flight><dep>BZK</dep><arr>VKO</arr></fligh
 t>
(1 row)
```

Предложение ORDER BY в этом случае должно быть размещено в качестве параметра агрегирующей функции.

В системе PostgreSQL имеются также высокоуровневые функции, преобразующие в формат XML содержимое таблицы, результат выполнения запроса, содержимое курсора или схемы. В этом случае имена элементов выбираются на основе соответствующих имен в реляционной схеме, а преобразование типов выполняется в соответствии с правилами для неявных преобразований. Например, функцию query_to_xml можно использовать для преобразования результата выполнения запроса в XML:

```
demo=# SELECT query_to_xml(
  'SELECT aircraft_code, range FROM aircrafts WHERE range < 3000',
  true, false, ' '
);
                          query_to_xml
---------------------------------------------------------------
 <table xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns=" ">                                                   +
                                                              +
 <row>                                                        +
   <aircraft_code>CN1</aircraft_code>                         +
   <range>1200</range>                                        +
 </row>                                                       +
                                                              +
 <row>                                                        +
   <aircraft_code>CR2</aircraft_code>                         +
   <range>2700</range>                                        +
 </row>                                                       +
                                                              +
 </table>                                                     +

(1 row)
```

Преобразование XML в табличный формат лучше всего выполнять с помощью функции xmltable, предусмотренной в стандарте SQL/XML и включенной в основной комплект PostgreSQL, начиная с версии 10. Функция xmltable описывает, как извлекать данные в форме таблицы из документа в формате XML:

```
xmltable( путь-к-таблице
  PASSING документ-источник
  COLUMNS (список-столбцов)
)
```

где

- *путь-к-таблице* — XPath-запрос, задающий путь к данным таблицы в исходном документе;

- *документ-источник* — выражение SQL, содержащее исходный документ;
- *список-столбцов* — описание форматов и методов вычисления значений в результирующей таблице.

Описание каждой колонки включает ее имя, тип SQL (в который следует преобразовать данные) и выражение XPath, указывающее путь от корня таблицы (заданного первым параметром функции xmltable) к значению (элементу исходного документа) и преобразующееся в значение описываемой колонки.

Функция xmltable вырабатывает значение типа отношение (таблица), которое можно использовать в обычных операторах SQL.

В следующем примере функция xmltable использована для формирования таблицы из XML-документа, который записан как константа в том же запросе. В реальных применениях, конечно, вместо констант используются значения переменных. Например, это может быть значение из столбца таблицы или входное сообщение, полученное из другой системы.

```
demo=# SELECT *
FROM xmltable( '//flights/flight'
  PASSING (
    SELECT xmlconcat(
      '<ticket>
       <number>0005432369015</number>
       <flights>
       <flight><from>VKO</from><to>BZK</to></flight>
       <flight><from>BZK</from><to>EGO</to></flight>
       <flight><from>EGO</from><to>AAQ</to></flight>
       <flight><from>AAQ</from><to>EGO</to></flight>
       <flight><from>EGO</from><to>BZK</to></flight>
       <flight><from>BZK</from><to>VKO</to></flight>
       </flights>
       </ticket>'
    )
  )
  COLUMNS
    departs_from text PATH 'from',
    arrives_to   text PATH 'to'
);
 departs_from | arrives_to
--------------+------------
 VKO          | BZK
 BZK          | EGO
 EGO          | AAQ
 AAQ          | EGO
 EGO          | BZK
 BZK          | VKO
(6 rows)
```

Существуют также функции более низкого уровня, извлекающие значения отдельных элементов из XML-документов на основе выражений XPath или XQuery.

8.6. Активные базы данных

Обычно системы управления базами данных выполняют действия только по запросам приложения. Рассмотрим СУБД, которые выполняют не только действия, явно указанные приложением, но также реагируют на события, возникающие в самой БД. Такие системы называются *активными базами данных*. Например, счетчик комментариев на сайте может быть рассчитываемой величиной, а может храниться в отдельной таблице и обновляться при добавлении нового комментария. В этом случае логика обновления счетчика реализована в БД как функция, которая вызывается при вставке новой записи в таблицу комментариев.

Формально поведение таких систем определяется в терминах предписаний, содержащих три компонента:

1) событие;
2) дополнительные условия;
3) описания действий, которые должны выполняться при указанном событии, если условия оказались истинными.

В научной литературе такие предписания называются *ECA-правилами* (event-condition-action). Мы используем термин *предписания*, для того чтобы избежать путаницы с *правилами*, которые применяются в PostgreSQL для совсем других целей.

На практике, в том числе в PostgreSQL, активность базы данных реализуется с помощью аппарата *триггеров*. Триггером называют функцию, обычно написанную на процедурном языке, которая вызывается системой автоматически при срабатывании связанных с ней предписаний. В спецификации триггера могут быть определены дополнительные условия.

В системе PostgreSQL различают триггеры, срабатывающие при модификации данных (INSERT, UPDATE, DELETE и TRUNCATE), и триггеры событий, срабатывающие при выполнении операторов языка описания данных, к числу которых относятся ALTER, CREATE, DROP, GRANT, REVOKE.

Процедурный код, реализованный в триггере, может существенно дополнить или изменить семантику стандартных операторов SQL. Например, триггеры можно использовать для проверки условий целостности, которые невозможно описать стандартными средствами языка SQL, или для регистрации изменений, выполняемых приложением, в другой таблице.

Важно подчеркнуть, что действия, предусмотренные в триггере, будут выполняться всегда, когда возникает специфицированная ситуация, и выполняются в рамках той же транзакции. Приложение не имеет никакой возможности обойти или отменить действие триггера. Это, с одной стороны, делает триггеры особенно полезными, например для регистрации действий пользователей, с другой — делает механизм триггеров потенциально опасным, поскольку ошибки в коде триггеров могут привести к существенному разрушению функциональности СУБД.

Действия, выполняемые триггером, в системе PostgreSQL задаются функцией, которая должна быть определена в базе данных до создания триггера. Обычно функция триггера не имеет явно описанных параметров, потому что информация о контексте вызова функции может быть получена другим способом, и возвращает значение типа trigger. Такие функции могут быть написаны на любом процедурном языке программирования, доступном для использования в PostgreSQL, при этом способ доступа к контексту, в котором возбужден триггер, зависит от языка программирования. В функциях, написанных на языке PL/pgSQL, для этого доступны предопределенные переменные.

Рассмотрим, как в PostgreSQL определяются триггеры, возбуждаемые при модификации данных. Для определения триггера модификации используется оператор CREATE TRIGGER, в котором указывается следующая информация:

- Уровень триггера.

  Триггеры могут быть определены на уровне операторов SQL (FOR EACH STATEMENT) или на уровне строк (FOR EACH ROW). На уровне строк триггер вызывается для каждой строки таблицы, которая обновляется оператором SQL. На уровне оператора триггер выполняется один раз при исполнении возбуждающего оператора.

- Операторы, выполнение которых возбуждает триггер (INSERT, UPDATE, DELETE, TRUNCATE).

- Объект базы данных, при модификации которого запускается триггер (таблица или представление).

- Относительное время выполнения триггера (BEFORE, AFTER, INSTEAD OF).

  Триггеры BEFORE срабатывают непосредственно до, а триггеры AFTER — сразу после возбуждающего оператора. Триггеры INSTEAD OF используются только для представлений и позволяют определить, какие действия должны выполняться вместо операций модификации данных. Таким образом, после определения триггеров представления можно сделать неотличимыми (по функциям) от хранимых отношений не только для оператора выборки данных SELECT, но и для операторов обновления данных INSERT, UPDATE, DELETE.

- Дополнительные условия, ограничивающие запуск триггера (WHEN).

  Эти условия можно рассматривать как реализацию условий ECA-предписаний.

- Функция триггера, выполняющая необходимые действия.

- Возможно, дополнительные параметры функции триггера.

Заметим, что одна и та же функция триггера может использоваться для определения разных триггеров.

В PL/pgSQL для триггеров уровня строк определены переменные OLD и NEW, содержащие соответственно старое и новое значения строки. При этом для оператора INSERT не существует старого значения, а для DELETE — нового.

Триггеры BEFORE могут изменять значения атрибутов в переменной NEW. Для того чтобы выполнение оператора, возбудившего триггер, было нормально продолжено, функция триггера должна вернуть непустое (определенное) значение. В триггерах, определенных для операторов INSERT и UPDATE, это значение будет использоваться в качестве нового значения обновляемого кортежа, поэтому функция должна вернуть исходное или измененное значение переменной NEW.

Если функция триггера возвращает неопределенное значение NULL, то для триггеров уровня строк прекращается обработка соответствующей строки, а для триггеров уровня оператора прекращается выполнение всего оператора. Однако откат транзакции ни в том, ни в другом случае не производится.

Приведенная далее функция триггера выводит значения переменных, определяющих контекст вызова триггера, и не выполняет никаких других действий. Мы используем эту функцию для иллюстрации возможных определений триггеров.

```
demo=# CREATE OR REPLACE FUNCTION show_trigger_parameters()
RETURNS trigger
AS $$
BEGIN
  RAISE NOTICE '%: % %.% % %',
    TG_NAME, TG_OP, TG_TABLE_SCHEMA, TG_TABLE_NAME, TG_WHEN, TG_LEVEL;
  IF TG_OP = 'DELETE' THEN
    RETURN OLD;
  ELSE
    RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
```

Воспользуемся определенной функцией для создания нескольких триггеров:

```
demo=# CREATE TRIGGER row_before
  BEFORE INSERT OR DELETE OR UPDATE
  ON flights
  FOR EACH ROW
  EXECUTE PROCEDURE show_trigger_parameters();
CREATE TRIGGER
demo=# CREATE TRIGGER row_after
  AFTER INSERT OR DELETE OR UPDATE
  ON flights
  FOR EACH ROW
  EXECUTE PROCEDURE show_trigger_parameters();
CREATE TRIGGER
demo=# CREATE TRIGGER stmt_before
  BEFORE INSERT OR DELETE OR UPDATE
  ON flights
  FOR EACH STATEMENT
  EXECUTE PROCEDURE show_trigger_parameters();
CREATE TRIGGER
demo=# CREATE TRIGGER stmt_after
  AFTER INSERT OR DELETE OR UPDATE
  ON flights
  FOR EACH STATEMENT
  EXECUTE PROCEDURE show_trigger_parameters();
CREATE TRIGGER
```

При выполнении операции обновления будут возбуждены все описанные выше триггеры:

```
demo=# BEGIN TRANSACTION;
BEGIN
```

```
demo=# UPDATE flights
  SET status = 'Cancelled'
  WHERE flight_id = 12345;
NOTICE:  stmt_before: UPDATE bookings.flights BEFORE STATEMENT
NOTICE:  row_before: UPDATE bookings.flights BEFORE ROW
NOTICE:  row_after: UPDATE bookings.flights AFTER ROW
NOTICE:  stmt_after: UPDATE bookings.flights AFTER STATEMENT
UPDATE 1

demo=# ROLLBACK;
ROLLBACK
```

Откат транзакции необходим, конечно, только потому, что мы не намерены оставлять подобные изменения в демонстрационной базе данных.

Для спецификации триггеров событий используется команда CREATE EVENT TRIGGER. Написание функций для этих триггеров и определение самих триггеров в целом аналогичны определениям функций и триггеров модификации данных.

Внутри функции триггера можно выполнять любые операторы SQL, допустимые в функциях. Это может привести к каскадному запуску другого или того же самого триггера, в том числе может вызвать бесконечную рекурсию, ответственность за предотвращение которой возложена на программиста. Заметим, что рекурсия может возникнуть вследствие определения триггера для других таблиц. Например, триггер на таблицу T_1, регистрирующий изменения в таблице T_2, может стать некорректным после того, как на таблице T_2 определяется триггер, модифицирующий T_1, т. к. обновление любой из этих таблиц приведет к запуску триггера, модифицирующего другую таблицу, т. е. к бесконечной рекурсии. Ошибки, связанные с применением триггеров, трудно обнаруживать, потому что триггеры вызываются неявно.

Триггеры являются удобным методом для решения некоторых задач, например для журналирования изменений в таблицах. Однако перенос логики на уровень триггеров базы данных несет в себе существенные риски. Хорошее обсуждение различных видов применений триггеров можно найти в [16].

Существует альтернативный метод описания способа модификации запросов с помощью *правил* (rules). Механизм правил позволяет переопределять не только операторы обновления, но и операторы выборки данных из таблиц и представлений. Мы не будем обсуждать этот метод.

8.7. Итоги главы

В этой главе мы обсудили различные расширения языка SQL, реализованные в системе PostgreSQL. Значительная часть этих расширений может быть охарактеризована как объектные средства. В частности, это относится к возможностям определения новых базовых типов данных, наследованию, использованию объектных указателей и определению пользовательских функций. Все это дает основания, для того чтобы называть PostgreSQL объектно-реляционной системой управления базами данных.

Особое место среди расширений занимают типы xml, json и jsonb, предназначенные для хранения и манипулирования данными в слабоструктурированных форматах. Эти типы не только иллюстрируют возможность создания новых типов данных, но и особенно важны в связи с их широким применением, например при передаче данных по информационным сетям.

Наконец, обработка событий в базе данных реализуется как в PostgreSQL, так и в других СУБД с помощью механизма триггеров.

8.8. Упражнения

Упражнение 8.1. Напишите запрос, выдающий список самолетов из демонстрационной базы в формате JSON.

Упражнение 8.2. Напишите запрос, выдающий список рейсов из демонстрационной базы в формате XML.

Упражнение 8.3. Напишите запрос, выдающий заданное бронирование в формате JSON, включая все входящие в него билеты и перелеты для каждого из билетов.

Упражнение 8.4. Решите задачу, обратную предыдущей: получив бронирование в формате JSON, вставьте в таблицы демонстрационной базы данных соответствующие строки.

Упражнение 8.5. Выполните два предыдущих упражнения, используя формат XML вместо JSON.

Упражнение 8.6. Создайте триггер, реализующий правило целостности в демонстрационной базе: рейсы могут совершать только те типы самолетов,

максимальная дальность полета которых превышает расстояние между аэропортами. Для расчета расстояния воспользуйтесь расширением earthdistance.

Упражнение 8.7. Создайте в базе данных триггер, который не позволит выполнять операторы CREATE в ночное время.

Упражнение 8.8. Создайте в демонстрационной базе вспомогательную таблицу и триггеры для аудита изменений рейсов. Изменения можно записывать в таблицу с тем же набором полей, а можно — в один JSON-столбец (что позволит избежать проблем при изменении структуры таблицы).

Упражнение 8.9. Создайте в демонстрационной базе событийный триггер, автоматически создающий для новых таблиц обычные триггеры для аудита изменений в этих таблицах.

Глава 9

Разновидности СУБД

9.1. Классы приложений БД

Как уже отмечено в главе 1, одной из основных предпосылок для выделения систем управления базами данных как отдельного класса программных систем стало появление устройств хранения данных с произвольным доступом относительно большой емкости. Именно возможности быстрого доступа к любым участкам пространства, в котором могут храниться данные, открыло возможности для создания приложений, работающих в режиме оперативного доступа и характеризующихся малым временем отклика.

В последующие десятилетия класс задач и приложений, для реализации которых использовались СУБД, постепенно расширялся. В настоящее время принято выделять два больших класса задач, отличающихся по характеру использования СУБД:

OLTP (Online Transaction Processing) — системы оперативной обработки (коротких) запросов;

OLAP (Online Analytical Processing) — системы оперативной аналитической обработки (больших объемов) данных.

Эти два больших класса не исчерпывают всех известных классов приложений СУБД, и границы этих классов нельзя считать четкими.

Системы оперативной обработки стали исторически первыми применениями баз данных, что в значительной мере определило требования к СУБД, упомянутые в главе 1. Подчеркнем, что в этом контексте термин *транзакция* обозначает совокупность действий, выполняемых приложением при однократном запуске, т. е. выполнение некоторой функции приложения, а не транзакцию в базе данных. В обоих случаях термин заимствован из прикладной области (банковские транзакции), однако в реальных системах обработка банковской транзакции обычно включает несколько транзакций в смысле баз данных. Даже такая

простая операция, как получение наличных в банкомате, состоит из нескольких транзакций СУБД, выполняемых в разных базах данных (банка — владельца банкомата, банка держателя карты и авторизационного центра).

Классические приложения класса OLTP характеризуются относительно большим потоком очень коротких транзакций, каждая из которых обрабатывает (считывает и зачастую изменяет) небольшое количество очень коротких записей (единицы или десятки записей, содержащих десятки или единицы сотен байтов каждая). Все транзакции в потоке независимы и выполняются от имени разных владельцев прав доступа. Все это определяет важность требований согласованности, целостности, быстрого поиска и обновления, отказоустойчивости, разграничения доступа и др.

К этому классу примыкают современные интернет-приложения, в которых в качестве транзакций выступают HTTP-запросы. Для таких приложений характерны несколько большие объемы обрабатываемых данных и несколько меньшая доля обновлений. Для небольшого числа лидеров отрасли наиболее важным требованием является масштабируемость, трудно (и чрезмерно дорого) достижимая при использовании СУБД общего назначения, что ведет к разработке специализированных систем и реализации упрощенных моделей управления данными. Однако для компаний малого и среднего размеров применение технологий традиционного типа, в частности на основе PostgreSQL, оказывается более эффективным.

Системы оперативной аналитической обработки предназначены для формирования обобщенных отчетов, получаемых обработкой всех или значительной доли записей, хранящихся в базе данных. Такие системы характеризуются полным отсутствием обновлений, что делает многие требования к СУБД, например согласованность и целостность, менее актуальными. Для того чтобы удовлетворить требования по времени отклика, обычно необходимо создавать вторичные копии данных, организация хранения которых существенно отличается от хранения первичных источников данных, и материализовывать промежуточные результаты, необходимые для быстрого получения окончательных отчетов.

Особый класс составляют системы совместного редактирования и разработки, в том числе системы автоматизированного проектирования (САПР). Подобные системы характеризуются относительно малым количеством объектов очень большого размера и специфическими требованиями, отличающимися от требований к СУБД общего назначения. Так, в системах этого класса обычные

требования к транзакциям оказываются слишком ограничительными: изменения, вносимые в процессе редактирования, могут занимать большое время, поэтому изоляция и атомарность оказываются нежелательными, хотя согласованность и долговечность остаются важными.

В таких системах хорошо проявили себя объектно-ориентированные СУБД, однако зачастую в них (вместо СУБД) используются специализированные надстройки над файловыми системами. В недавнем прошлом базы данных для хранения и обработки документов в слабоструктурированных форматах позиционировались как отдельный класс систем, однако в настоящее время их функциональность интегрирована в СУБД общего назначения.

Особые классы приложений связаны с обработкой очень длинных последовательностей (например, временны́х рядов или геномов), а также с хранением и обработкой больших графов.

9.2. Структуры хранения

Многообразие классов приложений, в которых используются СУБД, приводит к необходимости реализовывать и поддерживать большое разнообразие структур хранения.

Так, с точки зрения структур хранения для задач OLTP характерны объекты с небольшим числом атрибутов, при этом почти все атрибуты используются почти в каждом запросе, обрабатывающем такой объект. Поэтому системы, ориентированные на этот класс задач, размещают атрибуты каждого объекта данных по возможности в смежных участках памяти, а взаимосвязанные объекты, которые часто обрабатываются вместе, также могут группироваться. В терминах реализаций реляционной модели данных такое хранение принято называть «хранением по строкам». Такой способ организации хранения рассматривается как основной (если не единственный) в большинстве СУБД общего назначения, в том числе в PostgreSQL.

С другой стороны, для задач OLAP характерны записи, содержащие большое количество атрибутов, однако в каждом запросе используется только их небольшое подмножество. Это обстоятельство делает альтернативный способ хранения «по колонкам» более эффективным для таких задач в силу целого ряда факторов: более быстрое последовательное сканирование, возможность

сжатия данных, отсутствие обновлений и др. В то же время колоночная структура хранения существенно усложняет такие операции, как выборка всех атрибутов одного объекта, что делает ее менее эффективной для задач OLTP.

Исследования относительной эффективности строкового и колоночного представлений ведутся с середины 70-х гг., и к настоящему времени сформировалось понимание того, в каких случаях тот или другой способ оказывается более эффективным.

Точно так же относительная полезность и эффективность индексных структур существенно зависят от класса приложения. Например, индексы, которые хорошо работают для OLTP, оказываются малополезными для OLAP. Большое разнообразие индексных структур, реализованных в системе PostgreSQL, и в особенности возможности добавления новых типов индексов создают огромный потенциал для разработки высокоэффективных систем на основе этой СУБД. Эти возможности рассматриваются во второй части курса.

9.3. Архитектуры связи с приложениями

Одним из основных требований к системам управления базами данных является возможность совместного использования данных различными приложениями (или разными экземплярами одного приложения).

Поскольку основную нагрузку в ранних системах создавали приложения OLTP, для которых важно малое время отклика, необходимо было обеспечить совместное одновременное использование данных, т. е. запросы к БД должны были выполняться параллельно. По этим причинам практически все ранние СУБД использовали архитектуру, в которой приложение и программные компоненты системы выполняются в разных процессах. При этом процесс приложения направляет запросы и получает ответы на них, взаимодействуя с процессом, в котором выполняется СУБД.

Несколько позже для обозначения таких взаимодействий стали применяться термины *клиент* и *сервер*. Эти термины используются в разных смыслах. В этой книге мы обозначаем ими роли при взаимодействии программных или аппаратных компонент, при этом одна и та же компонента может иметь разные роли в разных взаимодействиях. В контексте баз данных клиент-серверная архитектура взаимодействия позволяет обеспечить доступ к базе данных нескольких сотен и даже тысяч сеансов одновременно.

Появление персональных компьютеров привело к относительно широкому распространению однопользовательских систем управления базами данных, которые, как правило, использовали табличное представление логической структуры данных, но включали упрощенный язык манипулирования данными без вычислительно сложных операций (таких, как операции соединения). Недостаточная мощность вычислительных систем, на которых работали такие СУБД, заставила отказаться от ряда требований, в частности от совместного использования данных, транзакций и т. п. Это привело к отказу от клиент-серверной архитектуры для этого класса СУБД. Совместное использование данных в таких системах реализуется размещением файлов базы данных на файл-сервере. Конечно, подобная архитектура не может обеспечить одновременную работу с общей базой данных большого количества пользователей (сопоставимого с возможностями клиент-серверных систем).

В настоящее время однопользовательские базы данных широко применяются в качестве встроенных систем в мобильных устройствах. В этом случае необходимость совместного использования данных на уровне базы данных не возникает, а потребление вычислительных ресурсов ограничивается емкостью аккумуляторной батареи.

Создание соединений с сервером базы данных и ведение активных сеансов требуют некоторых ресурсов на сервере БД. Поэтому системы управления базами данных, использующие архитектуру клиент — сервер, оказывались недостаточно масштабируемыми по числу активных соединений для некоторых классов OLTP-приложений, характеризующихся очень большим количеством пользователей.

Решением этой проблемы стала *многослойная* (multi-tier) архитектура, в которой часть приложения выполняется на системе клиента, а другая часть — на сервере. Ранние реализации таких серверов назывались мониторами транзакций, а позже стали называться *серверами приложений* (application servers). В такой архитектуре приложение по-прежнему является клиентом сервера базы данных и одновременно — сервером, обслуживающим следующий уровень клиентов (например, выполняя HTTP-запросы).

Масштабируемость по количеству клиентов достигается за счет использования ограниченного пула сеансов для работы сервера приложений с базой данных. Поскольку сервер приложений не хранит никаких общих данных, появляется возможность горизонтального масштабирования, т. е. установки дополнительных серверов приложений, но возникает необходимость в переносе некоторых

функций (например, разграничения доступа) на уровень сервера приложений или в само приложение.

Увеличение пропускной способности прикладных систем, достигаемое за счет использования серверов приложений, может приводить к некоторым негативным последствиям, компенсация которых требует дополнительных усилий при разработке приложений. Так, многократное использование соединений (сеансов работы) с базой данных может приводить к накоплению временных объектов на сервере базы данных, которые обычно удаляются при завершении сеанса. Кеширование данных на сервере приложений, как обсуждалось в главе 7, может потребовать дополнительных усилий для согласования обновлений, выполняемых через разные серверы приложений в одной базе данных.

9.4. Оборудование

9.4.1. Носители данных

На протяжении многих десятилетий единственным видом устройств, пригодным для хранения больших баз данных, оставались вращающиеся магнитные диски. В качестве основных свойств этих устройств, наиболее важных для выбора физической организации хранения данных и алгоритмов их обработки, можно назвать:

- время доступа к данным мало зависит от расположения этих данных на носителе;
- время доступа на несколько порядков превышает время доступа к данным в оперативной памяти;
- скорость передачи данных между устройством и оперативной памятью относительно высока;
- последовательное чтение или запись соседних участков носителя может выполняться на 1–2 порядка быстрее, чем произвольный доступ.

Из этих свойств, в частности, следует, что время считывания и записи относительно больших блоков (страниц) данных почти не отличается от времени обработки коротких записей, поэтому практически все структуры хранения баз данных размещают данные в блоках, которые считываются в оперативную память для обработки с последующей записью измененного состояния на диск.

Другими словами, фактически обработка происходит не на самом носителе, а в кеше (буфере), находящемся в оперативной памяти.

Разница во времени доступа к дискам и к оперативной памяти определяет метрики, необходимые для оценки, например индексных структур и моделей стоимости для основных операций базы данных, используемых оптимизатором. В случае если операция использует основное хранилище данных (на диске), доминирующим слагаемым в оценке ее стоимости является количество обращений (или время доступа) к дисковой памяти.

Появление твердотельных накопителей (SSD) достаточно большой емкости привело к необходимости пересмотра моделей стоимости и некоторых алгоритмов, поскольку характеристики SSD существенно отличаются от характеристик вращающихся дисков. Однако пока это не привело к радикальному изменению подходов к организации хранения данных в СУБД.

Быстрый рост размеров и снижение стоимости оперативной памяти вызвал рост интереса к системам баз данных, хранящим все данные в оперативной памяти. Основной проблемой таких систем является энергозависимость такой памяти, что требует специальных мер для предотвращения потери данных при отключении электропитания.

Можно выделить следующие основные направления и подходы, связанные с базами данных в оперативной памяти:

- Традиционные дисковые базы данных с очень большим размером оперативной памяти, выделенной для кеширования. Увеличение производительности в этом случае достигается за счет того, что вся обработка выполняется в оперативной памяти. Очевидным недостатком этого подхода является ограничение на размер базы данных, а узким местом часто оказывается запись обновленных данных на диск.
- Системы, не обеспечивающие сохранности данных при отключении электропитания. Такие системы зачастую используются для кеширования и ставятся перед традиционными дисковыми системами. Отличие от предыдущего класса систем состоит в том, что копирование данных в память происходит на уровне логической схемы, а не на уровне структур хранения.
- Распределенные системы баз данных в оперативной памяти, в которых каждый элемент данных хранится в нескольких копиях на разных узлах системы. Сохранность данных здесь основана на предположении о том, что одновременное отключение питания на всех узлах, хранящих копии

данных, маловероятно. Производительность подобных систем ограничивается пропускной способностью сети, связывающей узлы распределенной системы, и необходимостью синхронизации разных копий каждого элемента данных.

- Системы, использующие данные почти исключительно для чтения (например, базы данных для аналитической обработки).

Размещение данных в оперативной памяти открывает широкие возможности для применения альтернативных структур хранения, например для индексов.

Появление в последние годы энергонезависимых запоминающих устройств, обеспечивающих время доступа, сопоставимое со временем доступа к оперативной памяти (например, памяти с изменением фазового состояния, phase-change memory), открывает очень широкие возможности для систем управления базами данных, но потребует радикального пересмотра архитектуры средств хранения в составе СУБД.

9.4.2. Вычислительные ресурсы

Увеличение количества одновременно работающих устройств является концептуально наиболее простым способом увеличения мощности системы. В контексте высокопроизводительных СУБД параллельные системы существуют на протяжении десятилетий. В последние годы в связи со снижением темпов роста производительности отдельных устройств интерес к параллельной обработке значительно вырос.

Важно отметить, что увеличение вычислительной мощности (например, количества процессоров) не приводит автоматически к соответствующему увеличению производительности СУБД и программных комплексов, работающих на их основе. Для того чтобы потенциал параллельного оборудования реализовался, необходимо, чтобы структуры данных и алгоритмы могли эффективно использовать возможности такого оборудования.

Основными характеристиками качества параллельных систем являются:

- ускорение;
- масштабируемость по пропускной способности;
- масштабируемость по времени отклика.

В зависимости от конфигурации оборудования различаются следующие классы параллельных систем управления базами данных:

SM (Shared Memory) — многопроцессорные системы с общей оперативной памятью (и дисками), в которых не требуется пересылка данных для выполнения операций на разных процессорах;

SD (Shared Disks) — системы, в которых каждый из процессоров имеет свою оперативную память, недоступную другим процессорам, но все дисковые устройства доступны с любого процессора;

SN (Shared Nothing) — системы без разделения каких-либо устройств, взаимодействие между частями параллельной системы всегда использует вычислительную сеть.

Многие высокопроизводительные СУБД автоматически распознают многопроцессорные конфигурации и могут их использовать без какой-либо дополнительной ручной настройки. Однако возможности масштабирования многопроцессорных систем (SM), очевидно, ограничены.

Особо следует отметить многоядерные процессоры. Хотя отдельные ядра могут работать как независимые процессоры, оказывается, что такая работа может быть малоэффективна в контексте баз данных. Дело в том, что ядра используют общий кеш, и обычные алгоритмы параллельного выполнения основных операций баз данных эффективно работают только при малых объемах данных, не переполняющих этот кеш, т. е. для нагрузок типа OLTP, но не OLAP.

Системы класса SN допускают практически неограниченное масштабирование оборудования, однако пропускная способность сети может быть узким местом, в особенности при использовании сложных запросов. Фактически достигаемые значения характеристик производительности таких систем могут существенно зависеть от размещения данных и от смеси запросов, выполняемых сервером баз данных.

Отметим, что системы класса SN по аппаратной конфигурации мало отличаются от распределенных систем и иногда называются распределенными. Мы тем не менее будем различать эти термины. Параллельные системы баз данных создаются с целью повышения производительности систем, в то время как целью создания распределенных систем является в первую очередь повышение доступности и отказоустойчивости.

9.5. Хранилища данных

Реляционные СУБД по-прежнему являются доминирующими, однако нельзя игнорировать популярность альтернативных архитектур систем хранения данных, объединяемых направлением баз данных NoSQL. Сторонники таких систем обещают более высокую производительность и масштабируемость, по сравнению с реляционными СУБД, и пытаются сократить потери, возникающие из-за несоответствия между моделями и представлением данных в приложении и в хранилище.

По мере того как растет количество пользователей интернета, растет и объем производимых данных. Например, в 2016 г. пользователи Twitter каждую секунду генерировали 6 000 записей, пользователи YouTube каждую минуту добавляли 300 часов видео, а пользователи Instagram каждый день публиковали 80 млн фотографий. С другой стороны, обработка этих данных довольно проста и во всяком случае не требует выполнения каких-либо сложных запросов.

Для обработки такого объема данных используют горизонтальное масштабирование. Это означает, что нагрузка распределяется между большим количеством серверов, объединенных в кластер, и увеличение производительности достигается путем добавления новых машин. Конфигурация коммерческих баз данных для работы на кластерах требует настройки и дополнительных программных компонент, поэтому в NoSQL-сообществе считается, что реляционные системы плохо приспособлены к горизонтальному масштабированию. Действительно, вместо конфигурирования СУБД в системах NoSQL достаточно подготовить конфигурационные файлы, обычно в формате XML.

Поскольку для таких приложений, как правило, используются достаточно простые структуры данных и почти не используются сложные запросы, мощные возможности традиционных СУБД оказываются незадействованными. В частности, традиционные СУБД не могут показать высокую производительность при отсутствии запросов на массовую обработку.

Эти факторы привели к появлению NoSQL-систем, которые изначально проектировались распределенными, причем распределенность достигается «из коробки» и не требует сложной настройки. Сложность, конечно, не исчезает, однако настройка выполняется на другом уровне.

Экстремизм первых пропагандистов движения NoSQL, полностью отрицавших полезность высокоуровневых языков запросов, сменился более осторожными формулировками. Термин NoSQL в последние годы означает not only SQL, т. е.

«не только SQL». Возможно, это связано с тем, что многие NoSQL-системы обзавелись реализациями ограниченных подмножеств SQL (которые, однако, не могут соревноваться с реляционными системами по производительности).

Разумеется, системы обработки данных, не использующие SQL или использующие не только SQL, существовали и раньше. Однако именно термин NoSQL стал популярным после того, как его начали использовать в качестве метки (хештега) в различных информационных сообществах для обозначения нереляционных систем, созданных в XXI столетии.

Некоторые авторы, однако, относят к этому классу также и интенсивно разрабатывавшиеся в начале 1990-х гг. объектно-ориентированные системы баз данных.

Согласно утверждениям сторонников этого движения, NoSQL-системы обладают некоторыми общими характеристиками.

Не используют схему данных. Отказ от использования схемы данных означает, по существу, отказ от независимости данных и приложений и приводит к фактической невозможности совместного использования базы данных различными приложениями. Этот отказ становится возможным благодаря использованию средств синхронизации данных (обмена сообщениями) на уровне приложений.

Слабо поддерживают транзакционную семантику. В отсутствие совместного использования данных задачи поддержки согласованности решаются на уровне приложения, а не СУБД.

Приспособлены для использования на кластерах. Предельная простота применяемых структур хранения и отсутствие явных взаимосвязей между элементами данных, известными на уровне системы (а не приложения), практически устраняет необходимость координации при обработке данных, размещенных на различных серверах, входящих в вычислительный кластер.

Пропоненты NoSQL-систем в качестве отличительного свойства указывают на то, что такие системы являются системами с открытым кодом. В действительности это свойство не может считаться отличительным, потому что многие СУБД с открытым кодом (в частности, PostgreSQL) никак не могут быть отнесены к NoSQL, и не все системы NoSQL являются системами с открытым кодом. Открытость кода нельзя считать техническим свойством, однако оно оказывается существенным при выборе архитектуры, в особенности если систему

предполагается устанавливать на большое количество компьютеров для реализации горизонтальной масштабируемости.

Разные классы приложений предъявляют разные требования к целостности, согласованности и непротиворечивости данных. Очевидно, что у финансовых приложений такие требования значительно выше, чем, например, у социальных сетей. Базы данных класса NoSQL потенциально позволяют получить высокую производительность за счет ослабления ограничений целостности, более слабой поддержки транзакционной семантики и отказа от использования высокоуровневых языков запросов.

NoSQL-системы можно разделить на два типа: агрегатно-ориентированные базы данных и базы данных на основе графов.

9.5.1. Агрегатно-ориентированные базы данных

В NoSQL-системах выделяют три типа агрегатно-ориентированных баз данных.

Системы ключ — значение. Такие системы хранят пары ключ — значение. При этом значение может быть любым: например, строкой, документом или картинкой. Хранилище сохраняет значение в двоичном виде, ничего не знает о его структуре и не контролирует его тип. Чтобы получить значение, нужно обязательно знать ключ; выборки значения по каким-либо другим параметрам не поддерживаются. Соответственно, такие системы используются, когда нужен доступ по (единственному) ключу, а другие виды поиска и взаимосвязи между объектами не требуются.

Самые известные представители этого класса — Redis и Memcached.

Документоориентированные системы. Эти системы хранят произвольные структуры данных, как правило, в формате JSON. В отличие от систем ключ — значение документоориентированные системы поддерживают запросы, позволяющие находить документы по значениям определенных полей. Несмотря на то что документ может иметь произвольную структуру, запрос, выбирающий документы с определенным значением поля, подразумевает наличие этого поля, т. е. схему данных.

По своим функциям эти системы занимают нишу, в которой ранее находились системы, ориентированные на хранение и обработку данных в формате XML (базы данных Native XML).

Примерами таких систем являются MongoDB и CouchDB.

Хранилища семейств колонок. Такие системы являются расширением систем ключ — значение. Данные в них хранятся в виде разреженной матрицы, строки и столбцы которой используются как ключи. Такие системы принято выделять в отдельный тип баз данных NoSQL, но мы описываем их в разделе 9.1, поскольку именно этот способ организации данных используется для OLAP-систем (хотя структуры хранения могут существенно отличаться).

Примеры хранилищ семейств колонок — HBase и Cassandra.

В трех перечисленных типах NoSQL-систем сохраняемый объект может иметь сложную структуру, однако система всегда сохраняет это значение целиком, т. е. работает с ним, как с агрегатом. Агрегаты не зависят друг от друга, поэтому их несложно распределить по узлам кластера, увеличивая масштабируемость системы. ACID-транзакции в таких системах могут поддерживаться только на уровне агрегатов.

9.5.2. Базы данных на основе графов

Данные в этих системах представляют в виде графов, ребро показывает наличие взаимосвязи между объектами. Хранятся ребра, поэтому дополнительных вычислений для обхода графа не требуется. Такие базы данных используются в рекомендательных системах, например для предложений новых продуктов на основании предыдущих покупок пользователя или в социальных сетях, когда нужно получать друзей друзей. Для выборки подграфов используется специальный язык запросов. В отличие от агрегатно-ориентированных баз данных базы на основе графов поддерживают ACID-транзакции.

Примером такой системы является Neo4j.

9.6. Выбор СУБД для построения информационных систем

Выбор СУБД является важным решением, и при его принятии нужно учитывать функциональные и нефункциональные требования конкретного проекта. Ошибка в выборе СУБД стоит очень дорого, поскольку перевод проекта на новую СУБД неизбежно повлечет переработку программного кода и миграцию

данных. При этом нужно учитывать не только требования для ближайшей запланированной версии, но и вероятность последующих изменений функциональности системы, а также предположительный рост нагрузки. В случае создания программного продукта, который будет дорабатываться у заказчика, важна легкость внесения изменений для специфических требований проекта.

Для небольших систем обычно достаточно теоретической проверки, насколько выбранная СУБД соответствует требованиям. В более сложных случаях может быть оправдано создание испытательного проекта (proof of concept), который протестирует несколько кандидатов и поможет выбрать наиболее подходящего. Однако кандидаты для такого проекта также должны выбираться на основании некоторых критериев. Мы выделяем следующие группы требований:

Функциональные требования. Все современные реляционные СУБД поддерживают базовый набор операторов SQL для определения схемы, выборки и манипулирования данными. Следует обратить внимание на требования, для реализации которых могут потребоваться нестандартные типы данных (XML, JSON, геолокационные данные, типы для работы с временны́ми интервалами). Для многих бизнес-приложений необходимы полнотекстовый поиск и многоязыковая поддержка. Принципиальным является вопрос, какая часть логики будет реализована на сервере баз данных, а какая — в приложении. Чем больше логики планируется разместить на уровне СУБД, тем более важной становится поддержка хранимых процедур и триггеров.

Нефункциональные требования. Сюда относятся требования к производительности системы, масштабируемости, безопасности, созданию резервных копий и процессу восстановления после сбоев. Данные требования тесно связаны с бюджетом проекта, поскольку часто стоимость лицензий зависит от необходимого аппаратного обеспечения. Обычно это прямая зависимость от количества ядер процессоров.

Возможности для сопровождения системы. Сопровождение включает в себя инсталляцию и настройку системы, обновление программных компонент, обеспечение бесперебойного функционирования. Необходимо обратить внимание на инструменты для администрирования СУБД и оценить, насколько трудоемкими являются рутинные задачи сопровождения.

В процессе разработки или эксплуатации систем могут быть найдены ошибки в программном коде самого сервера баз данных, для устранения

которых нужно будет обратиться к разработчикам СУБД. В случае использования коммерческой СУБД обычно заключается договор поддержки на определенный срок, в который включено исправление ошибок.

Если выбрана система с открытым программным кодом, важна распространенность СУБД и полнота ее документации. Также следует обратить внимание на активность сообщества ее пользователей. Применение систем с открытым кодом дает возможность выбора между внешней поддержкой на основе договора и поддержкой собственными силами. При этом, как правило, второй вариант оказывается более дорогостоящим.

Бюджет проекта. Важной особенностью значительной части коммерческих СУБД является высокая стоимость лицензий на их использование. Эта особенность оказывается особенно ощутимой для высокопроизводительных (например, параллельных) и высоконадежных конфигураций. При этом сопровождение систем, построенных на основе таких СУБД и в таких конфигурациях, также оказывается технически сложным и дорогостоящим. И конечно, значительная часть бюджета проекта закладывается на заработную плату сотрудников, занимающихся разработкой и сопровождением. Эта сумма также различается в зависимости от выбранной СУБД. Обязательно нужно учитывать, насколько сложно найти сотрудников, обладающих необходимой квалификацией.

Финансовый вопрос часто оказывается решающим. В связи с этим в последние годы (и, возможно, пару десятилетий) растет популярность альтернативных моделей создания и эксплуатации систем, предполагающих интенсивное использование данных. В рамках этой тенденции можно выделить следующие популярные направления:

Отказ от использования функциональности СУБД. По существу, это означает перенос функциональности СУБД на уровень приложения, на которое возлагается ответственность за выполнение сложных операций обработки данных, поддержка согласованности и целостности, а также обмен данными с другими приложениями (поскольку совместное использование данных становится затруднительным).

Использование СУБД с ограниченными характеристиками или NoSQL-систем. Это хороший вариант для случая, когда выбранная СУБД полностью удовлетворяет всем требованиям проекта. Иначе необходимые характеристики достигаются за счет применения избыточных аппаратных ресурсов и, как и в предыдущем случае, ограничения функциональности системы.

Использование реляционных СУБД с открытым кодом. На сегодняшний день системы управления базами данных с открытым кодом являются достаточно привлекательными. Благодаря качественному коду и хорошей документации эти СУБД начинают обращать на себя внимание как малых предприятий, так и больших корпораций.

В этом контексте особую роль играет СУБД PostgreSQL, которая, оставаясь системой с открытым кодом, по своим эксплуатационным характеристикам приближается к высокопроизводительным коммерческим системам.

9.7. Итоги главы и первой части

Системы управления базами данных появились как отдельный класс программного обеспечения в 60-е гг. прошлого века. На протяжении всего периода существования систем этого класса происходит постоянное развитие их возможностей, связанное с непрерывным расширением области применения. Ранние СУБД применялись в небольшом количестве предметных областей, характеризующихся хорошо проработанной формализацией (финансы, телекоммуникации), а в настоящее время базы данных находятся в ядре любой информационной системы.

Развитие практически используемых промышленных систем сопровождалось развитием теории, в первую очередь теоретической реляционной модели данных, создавшей основания для применений высокоуровневых языков запросов и средств концептуального моделирования структур данных.

В современных системах управления базами данных реализуются высокоэффективные алгоритмы хранения, индексирования, оптимизации и выполнения запросов, протоколы, обеспечивающие корректное совместное использование разделяемых (общих) данных между несколькими приложениями и пользователями, а также средства надежного хранения и восстановления при отказах системы, которые могут обеспечить сколь угодно высокую степень защищенности данных от самых разнообразных отказов и разрушений.

Развитие больших систем управления базами данных, предоставляющих мощные возможности для создания разнообразных информационных систем, сопровождается появлением специализированных систем, предназначенных для решения вновь возникающих задач обработки данных. В последние годы подобные системы объединяются общим термином NoSQL. Ранее подобную

роль выполняли системы для хранения данных в формате XML, а в конце прошлого века — объектно-ориентированные СУБД.

Методы и технологии, отработанные в реализациях специализированных систем, зачастую включаются в состав больших универсальных систем управления базами данных. Так, все современные универсальные СУБД реализуют объектные расширения базовой реляционной модели данных, средства хранения и манипулирования XML и JSON. Современные СУБД могут работать в неоднородных распределенных системах, обрабатывая не только данные, которые хранят сами, но и данные из других источников.

Система PostgreSQL дает широкие возможности для выбора конфигурации и обеспечивает эффективную работу в очень широком диапазоне требований по объемам хранимых и обрабатываемых данных, типам и структурам данных, составу оборудования и контексту взаимодействия с другими системами.

Как известно, геометрические теоремы опровергались бы, если бы они задевали интересы людей. Поскольку системы управления базами данных интенсивно используются на практике, работы в этой области задевают интересы людей, поэтому в технической, популярной и даже в научной литературе можно найти огромное количество неточных или неверных утверждений, непроверенных или неверных фактов. Любые аргументы, связанные со сравнительными характеристиками систем, необходимо проверять по другим источникам или экспериментально.

9.8. Упражнения

Упражнение 9.1. Приведите основные отличительные черты OLTP и OLAP.

Упражнение 9.2. По каким критериям принято оценивать качество параллельных систем?

Упражнение 9.3. Какими общими характеристиками обладают системы, относящиеся к классу NoSQL?

Упражнение 9.4. Какие факторы определяют выбор СУБД для построения информационных систем?

Часть II

От практики к мастерству

Глава 10

Архитектура СУБД

Во второй части курса описываются алгоритмы, реализующие основные функции СУБД, применение средств расширения функциональных возможностей СУБД PostgreSQL, средства программирования логики приложений для исполнения на сервере баз данных, а также дополнительные возможности языка SQL, необходимые для создания сложных конфигураций серверов и для настройки производительности. Эти средства можно условно разделить на следующие группы:

Инструменты разработчика базы данных. К этой категории относятся дополнительные возможности языка запросов SQL (не рассмотренные в первой части курса), процедурные языки, используемые для программирования объектов базы данных (функций, триггеров и т. п.), а также средства настройки и приемы, обеспечивающие эффективное использование возможностей СУБД, в том числе выходящих за границы стандарта SQL.

Внутренние структуры и алгоритмы, применяемые в СУБД и, в частности, в системе PostgreSQL. Знание внутренних структур и алгоритмов необходимо для эффективного использования возможностей СУБД, а также для понимания того, в каких случаях и каким образом целесообразно применять средства расширения, предоставляемые СУБД PostgreSQL.

Архитектура любой многоцелевой системы управления базами данных, в том числе PostgreSQL, в значительной мере определяется традиционными требованиями к СУБД, рассмотренными в главе 1.

Такие системы, как правило, работают в режиме клиент — сервер, при этом СУБД выступает в роли сервера и содержит средства для организации сетевого взаимодействия с прикладной программой, выступающей в роли клиента.

Поскольку основной задачей любой СУБД является организация хранения данных, ее основу составляют структуры данных, методы доступа и поиска.

Реализация высокоуровневых языков запросов возлагается на модули, ответственные за компиляцию запросов, их оптимизацию и выполнение.

Поддержка согласованности базы данных и надежности реализуется тесно взаимосвязанными подсистемами управления транзакциями и средствами восстановления после отказов.

Важное свойство архитектуры современных СУБД, и в особенности PostgreSQL, состоит в их расширяемости. Наряду с расширениями языка SQL, уже включенными в состав системы, имеются развитые возможности для реализации средств, необходимых для новых классов прикладных систем и новых областей применения баз данных.

Наконец, средства создания параллельных и распределенных баз данных позволяют получать системы с очень высокими значениями характеристик производительности и надежности. Механизмы взаимодействия с внешними хранилищами данных обеспечивают прозрачность конфигурации распределенной (и, возможно, неоднородной) системы баз данных для приложения, т. е. возможность выполнения в ней распределенных запросов.

10.1. Интерфейс приложений

Основные понятия, необходимые для описания взаимодействия приложений с сервером базы данных, а также программные интерфейсы, применяемые в приложениях для доступа к базе данных, описаны в главе 7.

Напомним, что любое взаимодействие приложения с сервером базы данных начинается по инициативе приложения, которое создает сеанс работы с базой данных (устанавливает соединение). Как правило, для передачи информации между приложением (клиентом) и сервером баз данных используются средства сетевого протокола TCP/IP, даже если клиент обращается к локальной базе данных. Некоторые системы, в том числе PostgreSQL, предоставляют более эффективные возможности взаимодействия сервера с клиентом, работающим на той же вычислительной системе.

Важно отметить, что все взаимодействия клиента с сервером баз данных регламентируются протоколами с состояниями. Информация о состоянии сеанса хранится как на сервере баз данных, так и в процессе, в котором выполняется приложение. Каждый выполняемый оператор SQL регистрируется на сервере, и, если необходимо, на сервере хранится информация о результатах его выполнения. Хранение состояний на сервере баз данных приводит к явным (определяемым параметрами сервера) или неявным ограничениям на количество

одновременных сеансов и операторов, поскольку ресурсы оперативной памяти на сервере ограничены.

Для некоторых классов применений такие ограничения считаются недопустимыми, например для систем, обрабатывающих сообщения очень большого количества пользователей на основе протокола HTTP (без состояния). Для того чтобы исключить относительно дорогостоящие операции создания сеанса, серверы приложений, реализуемые как промежуточный слой таких систем, используют так называемый *пул сеансов* ограниченного размера.

Способ создания сеанса зависит от языка программирования, на котором написано приложение, и для любого языка может быть предусмотрено несколько вариантов. В системе PostgreSQL имеется несколько различных интерфейсов для работы с базой данных, при этом в каждом интерфейсе имеются различные функции создания сеанса, различающиеся способом задания параметров, а также методом синхронизации с сервером баз данных.

В системе PostgreSQL один сервер может обслуживать сеансы работы с несколькими базами, образующими кластер баз данных. При этом пользователь (роль с возможностью создания сеанса) определен на уровне всего кластера, а не отдельной базы данных. Для успешного создания сеанса, таким образом, необходимо, чтобы по указанному сетевому адресу был запущен сервер, слушающий указанный порт, на этом сервере должна существовать указанная база данных и указанный пользователь должен быть определен и должен иметь права доступа к этой базе данных.

Для некоторых языков программирования средства работы с базами данных PostgreSQL реализованы как отдельные пакеты, не входящие в состав PostgreSQL (например, драйверы JDBC). Поскольку такие драйверы не входят в состав системы PostgreSQL и зачастую рассматриваются как универсальные, предоставляющие унифицированный доступ к любым хранилищам данных, они могут явно или неявно существенно ограничивать возможности СУБД.

10.2. Обеспечение согласованности и отказоустойчивости

Независимо от того, какой тип клиентского интерфейса используется приложением, работа с базой данных в рамках одного сеанса представляет собой последовательность операторов SQL, передаваемых клиентом на сервер. Любой

запрос регистрируется менеджером транзакций, который связывает его с некоторой транзакцией, в рамках которой он будет выполняться. Если приложение не создало транзакцию само, то диспетчер автоматически создает неявную транзакцию. Поэтому любое действие приложения всегда выполняется в рамках некоторой транзакции, явной или неявной. Свойства транзакций и особенности их использования в приложениях рассмотрены в главе 6.

После регистрации операции диспетчером транзакций проверяется возможность ее выполнения, устанавливаются необходимые блокировки и т. п. Возможность выполнения и действия диспетчера транзакций зависят от выбранного уровня изоляции и, конечно, от наличия других активных транзакций. Теория управления транзакциями, модели согласованности и работа диспетчеров транзакций более детально обсуждается в главе 13.

Для того чтобы обеспечить сохранность данных при различных отказах, средства восстановления регистрируют все выполняемые операции обновления в журнале. Правила ведения журнала, алгоритмы восстановления и другие механизмы, необходимые для обеспечения сохранности данных, обсуждаются в главе 14.

10.3. Выполнение запросов

Поступивший от клиента оператор SQL прежде всего подвергается синтаксическому анализу и проверке на корректность. Результатом синтаксического разбора является внутреннее представление запроса, переданного клиентом, в виде дерева. Это дерево можно рассматривать как некоторое алгебраическое выражение, составленное из операций реляционной алгебры и дополнительных операций, реализующих конструкции SQL, выходящие за рамки реляционной алгебры. Результат синтаксического разбора запроса часто называют *логическим планом выполнения запроса*.

Набор операций включает все операции алгебры SQL (фильтрацию, проекцию, соединение, теоретико-множественные операции объединения, пересечения и разности, операции группировки и сортировки), а также операции, реализующие операторы языка описания данных (создание, модификация и удаление различных объектов базы данных, таких как таблицы, схемы, индексы, представления, функции).

Важная особенность компилятора запросов (в отличие от компиляторов языков программирования) состоит в том, что при построении логического плана

запроса используется информация об объектах базы данных, обрабатываемых в данном запросе: таблицах, представлениях и т. п. В системе PostgreSQL такая информация хранится в системном каталоге.

Построенный при компиляции логический план подвергается трансформациям на основе правил. Эти преобразования упрощают структуру плана, тем самым упрощая последующую оптимизацию, которая в PostgreSQL выполняется планировщиком.

На основе логического плана, полученного после этих преобразований, строится *физический план выполнения запроса*. Он также представляет собой дерево, вершины которого задают алгоритмы обработки данных, реализующие логические операции. Но структура дерева физического плана отличается от структуры логического плана, поскольку реализация логической операции может состоять из нескольких физических операций.

Структура физического плана выполнения запроса определяет, как результаты выполнения одних операций используются в качестве входных аргументов других. В листьях этого дерева обычно находятся операции извлечения данных из хранимых объектов, а результаты каждой операции передаются родительской операции в качестве аргументов. Результат, выработанный операцией, которая находится в корне дерева, становится результатом всего запроса и возвращается клиенту.

Для некоторых операторов SQL дерево может вырождаться в единственную корневую вершину. Например, это чаще всего бывает с операторами описания данных.

Для многих запросов существует несколько эквивалентных вариантов логического плана, а для многих логических операций — несколько вариантов их представления физическими операциями. Выполнение различных (физических) планов дает эквивалентные результаты, но может требовать различного количества вычислений. Количество вычислительных ресурсов, необходимых для выполнения разных эквивалентных планов, может отличаться на несколько порядков, поэтому задача выбора плана для выполнения очень важна для обеспечения высокой производительности СУБД.

Фактически планировщик решает две связанные задачи: для каждой операции выбирает алгоритм ее выполнения (если для операции имеется несколько алгоритмов, реализованных в системе) и перестраивает дерево таким образом, чтобы ресурсы, затрачиваемые на выполнение запроса, были в каком-то смысле минимальными.

Количество вычислительных ресурсов, необходимых для выполнения операций, зависит от статистических свойств данных, поступающих в качестве аргументов этих операций. Чтобы оптимизатор мог оценить затраты на выполнение операций, обычно система управления базами данных собирает статистическую информацию, характеризующую хранимые данные (в первую очередь информацию о размерах таблиц и количестве строк). На основе этой статистики делаются оценки затрат и размеров результата операций. Такие оценки не могут быть точными, поэтому результат работы оптимизатора не всегда оказывается действительно оптимальным. Точнее, оптимальное по оценкам решение может оказаться не оптимальным при фактическом выполнении. Иногда говорят, что цель оптимизации запросов — не получение оптимального решения, а исключение очень плохих решений; при таком подходе все планы с относительно хорошей оценкой считаются приемлемыми.

Задача оптимизации запросов относится к классу задач дискретной оптимизации, и, как большинство других задач этого класса, оказывается вычислительно сложной. Исчерпывающий поиск оптимального решения возможен только для планов небольшого размера; для сложных запросов применяются другие алгоритмы. Методы оптимизации, применяемые в настоящее время в системе PostgreSQL, обсуждаются в главе 12.

Построенное оптимизатором дерево интерпретируется исполнителем запросов. Алгоритмы, применяемые для выполнения отдельных алгебраических операций, рассматриваются в главе 11.

10.4. Организация хранения данных

Модули, отвечающие за хранение данных, реализуют отображение логических структур базы данных на файловую систему. В некоторых СУБД имеется возможность использовать дисковые устройства непосредственно, минуя файловую систему, но в большинстве из них файловая система используется. Это верно и для PostgreSQL.

Соответствие файлов и объектов базы данных может быть различным в разных СУБД. В некоторых системах отображение устанавливается на уровне табличных пространств: для каждого табличного пространства выделяется один или несколько файлов операционной системы, а остальные объекты размещаются в этих файлах. В других системах, в том числе в PostgreSQL, каждая таблица,

индекс и т. п. отображается на один или несколько файлов операционной системы.

Можно считать, что отображение объектов базы данных на файловую систему включает несколько уровней.

- Файлы, используемые для хранения данных, состоят из страниц (блоков) фиксированной длины. В системе PostgreSQL эта длина одинакова для всех файлов баз данных кластера и почти никогда не отличается от 8 Кб, поскольку может быть изменена только при компиляции сервера.
- Для объектов переменной длины определен общий формат хранения, не зависящий от типа или назначения этого объекта, и определен способ размещения таких объектов переменной длины на странице (или на нескольких страницах).
- Определены способы хранения составных объектов (например, строк таблицы, содержащих значения нескольких колонок).
- Определены структуры хранения для индексов, таблиц и других объектов базы данных.

Каждый следующий уровень основан на структурах, определенных на предыдущих уровнях. Так, объект большого размера, который не может быть размещен на одной странице, обычно выносится в отдельную область, которую можно рассматривать как служебную таблицу. При этом не имеет значения, является ли этот объект, например, скалярным значением текстового типа или значением составного типа, содержащим большое количество атрибутов. Для прикладного программиста, составляющего запросы на языке SQL, такое выделение незаметно.

Важная особенность PostgreSQL состоит в том, что при обновлении данные никогда не записываются на место, использованное для хранения предыдущей версии. С одной стороны, это дает простую возможность для отмены любых изменений, с другой — приводит к накоплению участков памяти, занятых устаревшими версиями данных. Для того чтобы использовать эту память повторно, выполняется процедура «сборки мусора» (по-английски называемая vacuum), которая обычно работает как фоновый процесс.

Структуры хранения, применяемые в PostgreSQL, а также операции выборки данных более подробно рассматриваются в главе 11.

10.5. Управление процессами и оперативной памятью

Как распределение оперативной памяти, так и управление процессами существенно зависят от типа операционной системы, под управлением которой работает сервер баз данных.

Доступная серверу оперативная память состоит из нескольких областей. Общие области памяти разделяются всеми процессами сервера.

Кеш базы данных (в некоторых учебниках он называется буферным пулом). Кеш используется для временного хранения страниц базы данных, которые необходимы выполняемым в данный момент запросам на выборку или на обновление данных. С кешем работает подсистема хранения данных и диспетчер транзакций. Кеш базы данных является общим для всех запросов и для всех пользователей.

Кеш журналов. Используется для регистрации изменений в данных. Так же как и кеш базы данных, используется подсистемой хранения и диспетчером транзакций и является общим для всех транзакций.

Список активных сеансов. Хранит информацию обо всех активных сеансах работы с базами данных.

Локальные области памяти выделяются отдельно для каждого процесса.

Активные запросы. Этот раздел оперативной памяти содержит информацию о запросах, готовых к выполнению, еще выполняющихся или уже выполненных, результаты которых обрабатываются приложением. Применительно к PostgreSQL эта область памяти содержит подготовленные операторы и состояния курсоров.

Временная область памяти. Используется для хранения промежуточных результатов операций, если они могут быть размещены в оперативной памяти (иначе в качестве временной области используется пространство в базе данных).

Фактическое распределение памяти между перечисленными (и, возможно, другими) областями до некоторой степени управляется параметрами, задаваемыми при конфигурации сервера баз данных.

Параметры сервера, управляющие распределением оперативной памяти, по умолчанию установлены таким образом, чтобы система PostgreSQL могла работать даже на самых скромных конфигурациях оборудования. Скорее всего,

значения параметров по умолчанию не позволят в полной мере использовать возможности оборудования даже для небольших баз данных и небольшой нагрузке на систему. Поэтому, хотя ручная настройка этих параметров не требуется, чтобы система могла начать работу, изменение значений параметров обычно существенно улучшает характеристики производительности сервера баз данных.

Ручное вмешательство в конфигурацию сервера необходимо для баз данных большого размера и высоконагруженных серверов (выполняющих большое количество запросов или запросы, выполнение которых требует значительной доли всех доступных ресурсов). Для достижения высокой производительности администратору базы данных следует планировать использование оперативной памяти. В первую очередь это относится к определению размеров кеша. Современные вычислительные системы, используемые в качестве серверов, зачастую имеют оперативную память достаточно большого размера, чтобы поместить в кеш все содержимое базы данных. В таких конфигурациях почти все запросы выполняются без дополнительных обращений к внешним носителям, а энергонезависимая память (HDD или SSD) необходима только для обеспечения надежности хранения.

Управление распределением памяти и другими параметрами сервера входит в функции администратора базы данных. Эти и другие функции администратора обсуждаются в главе 20.

10.6. Параллельные и распределенные базы данных

Сервер баз данных называется *параллельным*, если он использует возможности параллельной работы оборудования, например несколько процессоров, ядер процессора или несколько устройств хранения данных (дисков и т. п.) Параллельный сервер баз данных может быть реализован на различных конфигурациях оборудования, кратко описанных в главе 9 (SM, SD, SN), но каждая из них, конечно, требует совершенно различной поддержки со стороны программного кода СУБД. В системе PostgreSQL поддерживается конфигурация SM — многопроцессорная архитектура с общей оперативной памятью и устройствами хранения данных.

С точки зрения приложения параллельный сервер баз данных по своим функциям не отличается от сервера, работающего на однопроцессорной конфигурации. Поэтому единственная цель, которая может быть достигнута в резуль-

тате создания параллельного сервера, — увеличение производительности. Для оценки производительности параллельных серверов баз данных можно использовать метрики масштабируемости, описанные в главе 1.

Повышение производительности не достигается автоматически: для получения высокой производительности необходимо учитывать возможности оборудования при проектировании структуры хранения и распределении данных по устройствам. При этом различные схемы размещения будут по-разному влиять на характеристики производительности. Так, масштабируемость по пропускной способности необязательно обеспечивает масштабируемость по времени отклика, и наоборот.

В отличие от параллельных серверов *распределенная* система баз данных предполагает совместную работу нескольких серверов баз данных, обычно выполняющихся на различных вычислительных системах, связанных сетью.

Распределенные системы могут создаваться с целью повышения надежности или доступности. В этом случае дополнительные серверы используются для создания и обслуживания копий основной базы данных. Переход к работе с копией базы данных на другом сервере может обеспечить работу приложений в случае недоступности или отказа основного сервера.

Другой метод использования распределенных систем дает возможность совместной обработки данных, которые невозможно или нецелесообразно размещать в одной базе данных или на одном сервере баз данных.

В распределенной системе баз данных приложение может выдавать запросы, для выполнения которых необходимы данные, размещенные в нескольких базах данных. Такие запросы называются распределенными. Чтобы сервер баз данных мог принимать и выполнять распределенные запросы, в PostgreSQL в схеме базы данных должны быть определены внешние (foreign) источники данных, оформленные как внешние таблицы. Это позволяет выполнять запросы, использующие данные, фактически размещенные на разных серверах.

В системе PostgreSQL в качестве внешних источников могут использоваться другие серверы баз данных PostgreSQL, базы данных других производителей, системы хранения данных, не основанные на реляционной модели данных, а также файлы в нескольких распространенных форматах.

Другим аспектом распределенных систем является управление распределенными транзакциями. В системе PostgreSQL обеспечена возможность применения двухфазного протокола фиксации (two-phase commit, 2PC), однако это никак не связано с распределенным выполнением запросов на основе механизма

внешних таблиц. Обычно поддержка согласованности в распределенных СУБД реализуется в системах, надстраиваемых над PostgreSQL.

Более подробно параллельные и распределенные системы рассматриваются в главах 21 и 22.

10.7. Расширения и расширяемость

Постоянное расширение областей применения систем управления базами данных неизбежно приводит к тому, что имеющиеся в составе СУБД средства оказываются недостаточными для новых классов приложений. В течение более чем трех десятилетий развития язык SQL обогатился большим разнообразием средств, расширяющих его возможности. Применение этих средств обсуждается в главе 15.

Многие развитые системы позволяют добавлять пользовательские функции и процедуры, однако такие возможности часто оказываются ограниченными. Отличительная особенность PostgreSQL состоит в том, что эта система с самого начала проектировалась как расширяемая и поэтому предоставляет широкие возможности на различных уровнях как для поддержки новых классов приложений, так и для пополнения функциональности самой системы. В PostgreSQL возможно не только определение пользовательских типов данных, функций и процедур, но и новых структур хранения, в частности методов индексирования. Важно подчеркнуть, что такие структуры могут эффективно использоваться оптимизатором запросов наравне со встроенными структурами.

Средства и возможности расширения, предусмотренные в системе PostgreSQL, более подробно обсуждаются в главе 17.

Важными инструментами расширения являются процедурные языки программирования, позволяющие реализовывать функциональность приложений, используя выполняемые на сервере баз данных функции и процедуры. Эти языки необходимы также для реализации многих видов расширений функциональности самого сервера баз данных. Применение процедурных языков обсуждается в главе 16.

Некоторые расширения включены в ядро основной распространяемой версии PostgreSQL. В частности, в составе системы имеются средства полнотекстового поиска, которые обсуждаются в главе 18.

10.8. Безопасность

Защита данных является одной из основных функций систем управления базами данных. Средства обеспечения безопасности в системе PostgreSQL включают защиту от несанкционированного доступа к базам данных и разграничение доступа как на уровне объектов схемы базы данных (таблиц, представлений, функций и др.), так и на уровне отдельных строк таблиц.

Модель защиты, применяемая в системе PostgreSQL, а также основные способы управления ролями и предоставления прав доступа к данным рассматриваются в главе 5 первой части курса.

Глава 19 содержит более детальное обсуждение средств защиты в PostgreSQL. В ней в том числе обсуждается разграничение доступа на уровне отдельных строк таблиц.

10.9. Итоги главы

В этой главе кратко охарактеризованы основные компоненты СУБД, описаны их функции и взаимодействие между ними, а также указаны некоторые особенности системы PostgreSQL.

В следующих главах детально обсуждаются теоретические основания, алгоритмы и протоколы, а также особенности реализации и применения этих модулей.

10.10. Упражнения

Упражнение 10.1. Найдите в исходном коде PostgreSQL модули, отвечающие за:

1) управление сеансами;
2) разграничение доступа;
3) диспетчеризацию транзакций;
4) ведение журнала транзакций;
5) рестарт сервера баз данных;
6) выполнение операций реляционной алгебры;

7) реализацию основных структур хранения данных;
8) оптимизацию запросов;
9) реализацию и использование индексов на основе B-деревьев;
10) реализацию индексов GiST.

Глава 11

Структуры хранения и основные алгоритмы СУБД

В этой главе обсуждаются вопросы, которые необходимо решать разработчику СУБД при выборе методов хранения данных: организация хранения таблиц, структуры индексов, а также алгоритмы, необходимые для реализации операций реляционной алгебры и алгебры SQL. Далеко не все из рассматриваемых альтернативных проектных решений применяются в системе PostgreSQL. Ни разработчикам приложений, ни администраторам базы данных обычно не нужно реализовывать что-либо из описанного в этой главе, однако знание этого материала помогает наилучшим образом использовать возможности СУБД.

11.1. Хранение объектов логического уровня

Объекты логического уровня — это объекты, в терминах которых происходит взаимодействие приложения с сервером базы данных. Поскольку PostgreSQL относится к числу SQL-ориентированных систем, основным видом логических объектов являются таблицы — коллекции объектов, каждый из которых содержит фиксированный набор атрибутов, один и тот же для всех объектов одной таблицы.

В то же время место для хранения любых видов данных выделяется в терминах блоков (или страниц) одинакового размера, как минимум в пределах одного табличного пространства. В PostgreSQL размер страницы фиксирован и обычно составляет 8 Кб. Этот размер можно изменить на 16 или 32 Кб при компиляции из исходного кода, но в любом случае размер будет одинаковым для всех табличных пространств.

11.1.1. Размещение коллекций объектов

Разработчик СУБД должен принимать решения, определяющие, каким образом объекты коллекции (например, строки таблиц) будут размещаться в блоках. Поскольку не может существовать единственного метода, оптимального для всех применений и для всех видов нагрузки, высокопроизводительные СУБД обычно предоставляют несколько альтернативных методов размещения логических объектов и коллекций. Метод размещения (из предоставляемого СУБД ассортимента) выбирается разработчиком приложения совместно с администратором базы данных для каждой таблицы на этапе проектирования схемы хранения базы данных с учетом требований приложений, использующих эту базу.

Варианты отображений логических объектов на структуры хранения

В процессе проектирования методов хранения логических объектов разработчик СУБД должен принимать решения по ряду вопросов, определяющих свойства и характеристики структуры хранения.

Фрагментация объектов определяет, каким образом и при выполнении каких условий данные одного объекта разбиваются на части, размещаемые на разных страницах хранилища. Например, в системе PostgreSQL значения атрибутов строки таблицы, длина которых превышает порог, могут подвергаться сжатию, и если после этого длина все еще превышает порог, то размещаются отдельно от значений остальных атрибутов строки таблицы. При этом стратегия сжатия и размещения зависит от типа значения и может быть переопределена для отдельных колонок.

Адресация и перемещаемость определяют, каким образом система представляет ссылки на объект, и может ли измениться положение объекта в хранилище после того, как он был размещен первоначально.

Во многих СУБД применяется обновление «на месте», т. е. старые значения заменяются на новые, возможно, другой длины. При этом фактически данные могут перемещаться, однако обычно в таких системах принимаются меры, для того чтобы адреса оставались неизменными. В системе PostgreSQL при любом изменении создается новая версия объекта, которая получает новый адрес, а предыдущая версия остается доступной по

старому адресу, до тех пор пока она не будет удалена в результате сборки мусора. При этом все объекты могут перемещаться в пределах одной страницы.

Размещение по значениям определяет, зависит ли размещение объектов коллекции от значений атрибутов объектов. В PostgreSQL размещение по значениям атрибутов не применяется.

Упорядочение определяет, учитывается ли отношение порядка для некоторого атрибута (или набора атрибутов) при размещении объектов. Такая организация хранения полезна, если достаточно часто выполняются запросы, в которых требуется выборка объектов по интервалу значений атрибута. Заметим, что размещение по значениям, намеченное в предыдущем пункте, не обязательно предполагает упорядоченность.

В системе PostgreSQL упорядочивание строк таблиц реализуется с помощью команды CLUSTER, которая реорганизует таблицу в соответствии с упорядочением в одном из индексов. Однако при обновлениях это упорядочение не поддерживается, поэтому необходима периодическая повторная реорганизация. Этот механизм применяется редко, поскольку доступ к таблице во время выполнения такой операции полностью блокируется.

Совместное размещение коллекций в общем наборе страниц хранилища позволяет собрать вместе (на одной или смежных страницах) строки разных таблиц, содержащие одинаковые значения каких-либо атрибутов. В системе PostgreSQL значения атрибутов не используются при выборе места для записи строки, и совместное размещение не применяется.

Решения по этим вопросам определяют, какие алгоритмы доступа к данным коллекции могут и будут использоваться при работе СУБД.

При проектировании структуры хранения необходимо также принимать решения, связанные с поведением структуры при внесении изменений. В частности, необходимо определить:

- стратегию поиска и выделения свободного места для новых объектов и стратегию добавления новых страниц для объектов коллекции;
- действия при переполнении страницы (при увеличении размеров объекта или при добавлении новых объектов на страницу);
- обработку удалений объектов и недостаточно заполненных страниц;

- возможность одновременного доступа нескольких транзакций, в том числе обновляющих данные на одной странице;
- реорганизацию структуры хранения с целью улучшения ее характеристик, если при обновлениях структура может деградировать (примером может служить сборка мусора, которая в PostgreSQL обычно выполняется фоновым процессом).

Адресация и перемещаемость

Доступ к объектам по адресу на уровне хранения необходим даже в тех случаях, когда модель данных не предусматривает явные ссылки на объекты. Например, в реализациях реляционной модели данных доступ к строкам таблиц может производиться с использованием индексов, и для перехода от индексной записи непосредственно к объекту используется прямая адресация. Наиболее эффективным методом адресации является использование низкоуровневого (абсолютного или относительного) адреса страницы и смещения на странице. Такая адресация, однако, несовместима с необходимостью перемещать объект при изменении его длины или длины других объектов, размещенных на той же странице.

Для того чтобы обеспечить неизменность адреса, используется ряд приемов.

- Замена смещения на порядковый номер объекта на странице, возможно, с поддержкой массива смещений объектов, размещенных на этой странице. Этот прием обеспечивает неизменность адреса объекта при его перемещении в пределах одной страницы.
- Использование «якорей» объектов фиксированной длины, содержащих ссылку на фактическое размещение объекта. Как правило, объект размещается на той же странице, что и якорь, что исключает потерю эффективности из-за косвенной адресации. Однако при увеличении длины объект может перемещаться в другое место с сохранением размещения якоря.

Как видно из структуры страницы, описываемой ниже в разделе 11.1.2, в системе PostgreSQL применяется первый из этих приемов, поэтому в качестве ссылки на объект можно использовать номер страницы, на которой размещается объект, в сочетании с номером позиции в массиве указателей на объекты этой страницы. Такая ссылка остается неизменной, если объект перемещается в пределах страницы.

Заметим, что фиксированная адресация несовместима с размещением объектов по значению. Это обстоятельство не имеет значения в системе PostgreSQL, поскольку размещение по значению для основных таблиц не используется. Конечно, размещение по значению необходимо в индексных структурах, однако нет необходимости ссылаться на объекты индекса извне самой структуры этого индекса.

Размещение по значениям и упорядоченность

Размещение объектов с учетом значений одного или нескольких атрибутов позволяет исключить использование индекса при фильтрации по такому атрибуту (или набору атрибутов). По существу, такое хранение является комбинацией индексной структуры и структуры для хранения коллекций.

Если индексная структура, на основе которой организовано такое хранение, поддерживает упорядоченность, то появляется возможность весьма эффективно выполнять запросы, требующие фильтрации по интервалу значений ключа, по которому упорядочены строки таблицы.

Подчеркнем, что такая организация хранения может применяться для размещения по ключу, который не является уникальным.

Хотя в PostgreSQL подобные структуры для хранения таблиц не используются, соответствующий алгоритм выборки данных применяется в том случае, если все данные, необходимые для выполнения запроса, содержатся в индексе и обращение к основной таблице не требуется.

Совместное размещение коллекций

Совместное хранение предполагает размещение объектов нескольких коллекций на одной странице. Такое размещение полезно, если объекты этих коллекций связаны общим значением некоторого атрибута и часто используются вместе (например, выполняется операция соединения по этому атрибуту).

В системе PostgreSQL такой способ организации хранения таблиц не поддерживается на уровне структур хранения, однако его можно моделировать на уровне логической структуры, используя атрибуты с множественными значениями (например, массивы).

11.1.2. Размещение данных на страницах

Напомним, что, поскольку PostgreSQL поддерживает объектные расширения, типы атрибутов могут быть достаточно сложными — в частности, могут содержать коллекции (представленные массивами), слабоструктурированные данные (например, в формате JSON), тексты и пр. Длины скалярных атрибутов (например, символьных строк text или varchar) могут изменяться. Размер памяти, необходимый для хранения объекта, может варьироваться в очень широких пределах как для разных коллекций, так и для разных объектов одной коллекции и даже значений одного атрибута в одной коллекции.

Для того чтобы достаточно эффективно обрабатывать такое большое разнообразие, наборы логических объектов PostgreSQL (отношения) представляются структурой хранимых таблиц. Принципиальных различий между логическими и хранимыми таблицами нет, и обычно их не различают. Если содержимое логической таблицы (отношения) удовлетворяет ограничениям, наложенным на хранимые таблицы, то для хранения этой логической таблицы используется одна хранимая и отображение становится тождественным.

Ограничение, накладываемое на хранимые таблицы, состоит в том, что длина записей в таких таблицах не может превышать размер страницы и фактически ограничивается некоторым порогом (обычно около 2 Кб), обеспечивающим возможность хранения нескольких записей на одной странице. Если размер объекта логического отношения превышает этот порог, то значения атрибутов этого объекта преобразуются и часть данных может быть вынесена в другую хранимую таблицу, которая связана с тем же логическим отношением.

Значения переменной длины

В системе PostgreSQL для представления значений переменной длины используется общий формат, не зависящий от типа значения: непосредственно перед значением записывается его длина в байтах, включая место, необходимое для хранения самой длины. В общем случае для записи длины выделяется 4 байта, из которых 2 бита используются для служебных целей; для записи длины остается 30 бит, что ограничивает максимальный размер значения одним гигабайтом.

Выделение 4 байтов для хранения длины коротких значений (скажем, строковых значений небольшой длины) привело бы к чрезмерному расходу памяти.

Поэтому для коротких значений длина записывается в один байт, старший (служебный) бит которого установлен в ноль. Оставшиеся 7 бит дают возможность хранения длины, не превышающей 127, включая один байт на саму длину, поэтому максимально возможный размер такого значения составляет 126 байт.

Похожие схемы организации хранения объектов переменной длины применяются и в других СУБД.

Значения большего размера могут быть представлены в нескольких различных форматах, и при необходимости система PostgreSQL выполняет преобразования. Формат, в котором значение без всяких изменений записывается непосредственно после длины, необходим для передачи значений между сервером базы данных и приложением (клиентом), а также применяется для временного хранения значений в оперативной памяти.

Если размер значения превышает определенный порог (обычно около 2 Кб), такое значение разрезается на фрагменты, каждый из которых оформляется как отдельная запись в служебной хранимой таблице типа TOAST (The Oversized-Attribute Storage Technique), которая представляет собой внутреннюю структуру, используемую для хранения объектов большого размера. Конечно, идентификация этих фрагментов определяет значения, из которых они получены, а также логический объект, содержащий эти значения. И, конечно, длина этих фрагментов не превышает порог.

Такая служебная таблица может быть автоматически создана для каждого логического отношения в дополнение к основной хранимой таблице, в которую записываются объекты этого логического отношения. Таким образом, при постоянном хранении в базе данных значения большого размера представлены набором фрагментов («тостов»); кроме этого, длинные значения могут храниться в сжатом виде.

Структура страниц

Организация данных на странице в системе PostgreSQL не зависит от типов хранимых данных и от типа объекта, к которому относятся эти данные: хранимые таблицы, индексы и др. Единицей хранения на странице является запись, которая может быть строкой таблицы, индексной записью, фрагментом значения большого размера (тостом). Для модулей, отвечающих за размещение данных на странице, это не имеет никакого значения.

Поскольку требования к структурам хранения довольно близки в самых различных системах, организация страниц оказывается очень похожей во многих СУБД. Например, в PostgreSQL она содержит следующие разделы:

Заголовок страницы фиксированного размера (24 байта).

Массив указателей на записи, содержащиеся на этой странице. Эти указатели содержат смещения записей относительно начала страницы, их длину и служебные биты. Массив имеет переменный размер, зависящий от числа записей (объектов) на этой странице.

Незанятый участок памяти, который может быть использован для добавления новых записей (или для расширения имеющихся в тех системах, в которых записи могут обновляться), а также для расширения массива указателей.

Область данных, в которой находятся размещенные на странице объекты или их заголовки (для объектов большого размера).

Дополнительная область, которая может по-разному использоваться для индексов разных типов и не применяется для таблиц.

Ссылки на объекты включают идентификацию страницы базы данных и номер позиции в массиве указателей, размещенном на этой странице. Такая косвенная адресация дает возможность перемещать объекты в пределах страницы без изменения ссылок.

Несмотря на то что в PostgreSQL при любых изменениях создается новая версия объекта, а ранее созданная версия не изменяется, перемещение объектов требуется для устранения фрагментации памяти на странице после удаления устаревших версий.

Вспомогательные структуры

Кроме основных файлов, используемых для хранения таблиц или индексов, в системе PostgreSQL создаются дополнительные структуры:

- файл, хранящий информацию о наличии свободного места на страницах таблицы или индекса (free space map, FSM);
- карта видимости (visibility map, VM), описывающая, какие страницы содержат записи, обрабатываемые активными транзакциями, и какие страницы могут содержать записи, подлежащие удалению.

Карта свободного пространства организована как дерево. На нижнем уровне (в листьях этого дерева) хранится один байт на каждую страницу таблицы или индекса, указывающий на возможность добавления записей на эту страницу. Для таблиц или индексов большого размера верхние уровни содержат обобщенную информацию о наличии свободного места на страницах, описываемых вершинами дерева следующего уровня. Карта свободной памяти не создается для индексов на основе хеширования.

Карта видимости содержит информацию о том, какие страницы могут содержать устаревшие версии строк. Значения битов признаков в карте видимости устанавливаются консервативно: 1 обязательно означает, что условие выполняется (устаревших версий нет), однако 0 означает только, что условие может нарушаться (устаревшие версии могут быть). Карта видимости создается для таблиц, но не для индексов.

Если страница таблицы не содержит устаревших версий строк, ее не нужно обрабатывать при сборке мусора (vacuum). Кроме того, карта видимости определяет необходимость дополнительной проверки после поиска по индексу. Дело в том, что индекс может содержать записи, указывающие на устаревшие версии строк таблиц, и поэтому для исключения таких строк из результатов поиска необходима дополнительная проверка содержимого табличной страницы. Если же известно, что страница не содержит устаревшие версии, такую проверку можно опустить.

Кроме этого, к служебным структурам можно отнести таблицы TOAST, применяемые для хранения больших объектов, потому что обычно эти таблицы не упоминаются явно в запросах пользователей. Организация хранения таблиц TOAST не отличается от хранения обычных таблиц: для них также создаются необходимые вспомогательные структуры.

11.1.3. Хранение больших объектов

В системе PostgreSQL имеется возможность создания больших объектов, не входящих в какую-либо таблицу или другую структуру (однако фактически такие объекты создаются в одной из системных таблиц). При создании большого объекта PostgreSQL возвращает значение типа oid, которое можно использовать в дальнейшем для доступа к этому объекту. Содержимое больших объектов никак не интерпретируется PostgreSQL; с точки зрения ядра СУБД это просто последовательность байтов.

Другими словами, логическая структура данных, хранимых в большом объекте, полностью определяется кодом приложения. Заметим, что такой код может находиться как в программе, выполняемой в роли клиента базы данных, так и в виде функций, размещенных на сервере базы данных и таким образом расширяющих функциональность PostgreSQL.

По историческим причинам размер типа oid составляет 4 байта, что ограничивает максимальное количество больших объектов, которые могут быть созданы в одном кластере баз данных.

Понятие большого объекта считается устаревшим и, скорее всего, будет вытеснено структурами TOAST, однако предельный размер объекта, хранимого в TOAST, составляет 1 Гб, а максимальный размер большого объекта — 4 Тб. Кроме этого, интерфейс доступа к большим объектам лучше приспособлен для обработки потоков байтов, таких, как звук или видео. Заметим, что нет никаких концептуальных препятствий, для того чтобы реализовать аналогичный интерфейс и для объектов, хранимых в структурах TOAST.

11.1.4. Строки или колонки?

В зависимости от класса задач, в которых используются данные, для каждой таблицы может быть выбран один из нескольких способов хранения, если СУБД такие альтернативы предоставляет. Так, для задач оперативной обработки (OLTP), характеризующихся тем, что каждое выполнение приложения использует очень небольшую часть объектов базы данных, а обновления этих объектов происходят относительно часто, обычно используется хранение таблицы *по строкам*. Это означает, что атрибуты одной строки таблицы хранятся в смежных областях памяти внутри страницы, и, поскольку объекты имеют небольшие размеры, на каждой странице размещается несколько объектов.

Другой способ хранения таблиц — *по колонкам* — состоит в том, что в смежных областях памяти хранятся значения одного атрибута из разных объектов (строк) отношения. Такое хранение целесообразно использовать, если запросы, как правило, выбирают значительную часть хранимых в таблице объектов, но используют небольшое количество атрибутов, и при этом обновления выполняются относительно редко. Все эти предположения, как правило, выполняются для задач аналитической обработки (OLAP).

В начале 80-х гг. было экспериментально установлено, что хранение данных по строкам дает преимущества в производительности для класса задач, который

доминировал в то время (OLTP), и на характеристиках оборудования, которое было доступно в то время. В дальнейшем, в связи с расширением области применения баз данных и изменением характеристик оборудования, оказалось, что для решения аналитических задач хранение по колонкам более предпочтительно.

Во многих случаях целесообразно применять гибридные структуры: хранить в столбцах значения, составленные из нескольких атрибутов, часто используемых вместе в одном запросе. Своего рода экстремальным решением является хранение каждого атрибута как отдельной коллекции.

В системе PostgreSQL хранение по колонкам не реализовано, хотя возможно использование внешних хранилищ через механизм оберток сторонних данных (foreign data wrappers). Можно моделировать хранение по колонкам, заменяя таблицу на совокупность нескольких таблиц с малым числом колонок в каждой, однако такое решение не будет эффективным, поскольку большой объем памяти будут занимать заголовки строк. В версиях системы PostgreSQL, начиная с 12, возможно подключение альтернативных методов хранения таблиц. Среди таких методов, несомненно, будет доступна и организация хранения по колонкам.

11.2. Индексы

В контексте структур хранения индексом называется вспомогательная (избыточная) структура данных, предназначенная для ускорения некоторых классов запросов. Более конкретно — индексы позволяют применять эффективные алгоритмы обработки запросов, содержащих условия на значения атрибутов, для которых построен индекс.

Любой индекс можно рассматривать как коллекцию, состоящую из объектов специального вида (индексных записей), включающих два атрибута:

индексный ключ, представляющий поисковый образ (как правило, значения атрибута или атрибутов, для которых построен индекс);

информацию об объектах, которую в той или иной форме задает указатель (или указатели) на объекты, содержащие значения атрибутов, совпадающие со значениями в индексном ключе.

В случае если значение индексного ключа содержится в нескольких объектах, в системе PostgreSQL индексные записи, как правило, дублируются; в других системах вместо этого могут храниться списки указателей. Очевидно, в первом случае возникает избыточность из-за дублирования ключей, во втором — необходимо специальным образом обрабатывать списки, которые могут быть очень длинными. Конечно, для атрибутов, на значения которых наложено ограничение целостности UNIQUE, дублирование ключей не требуется. Известны разновидности индексов, в которых вместо списков указателей используются битовые карты.

Особенность индексов по сравнению с другими коллекциями состоит в том, что для индексов имеет смысл только выполнение запросов весьма ограниченного вида, а именно выполняющих фильтрацию по значениям ключа (не обязательно по совпадению значений, но в любом случае проверяется истинность некоторого предиката, связывающего поисковый критерий со значениями индексного ключа). Благодаря этому для индексов оказывается целесообразным применять значительно более развитые конструкции, чем для коллекций общего вида.

Изменение значения атрибута в таблице приводит к необходимости удаления из индекса старого значения и вставки нового. Ассортимент операций обновления индексов обычно включает только эти две операции: вставка новой записи (или ссылки) и удаление существующей. Как известно, в PostgreSQL операции UPDATE также выполняются как удаление старой и вставка новой версии, но в других СУБД обновление таблиц может выполняться иначе.

Перечислим основные требования, предъявляемые к индексным структурам, и критерии их оценки.

Поскольку основное назначение индексов — ускорение поиска данных, наиболее важным критерием их полезности является время поиска в индексе. Традиционно для оценки времени поиска применяется количество страниц, которые необходимо обработать, для того чтобы выполнить запрос на поиск в индексе. Применительно к ранним системам важность этого критерия связана с тем, что при размещении страниц на диске время ожидания обмена существенно превосходит суммарное время выполнения всех остальных операций, необходимых для выполнения запроса. В связи с ростом объемов оперативной памяти и распространением устройств хранения SSD значение критериев, основанных на времени доступа к вращающимся дискам, снизилось, однако оценки, основанные на количестве обрабатываемых страниц, по-прежнему дают некоторое представление о сложности индексных структур.

Можно предполагать, что количество операций, выполняемых процессором при обработке одной страницы индекса, примерно одинаково для всех страниц (другими словами, время процессора не будет существенно различаться при обработке разных индексных страниц). Количество обработанных страниц можно поэтому применять для оценки времени поиска в индексе, даже если все требуемые страницы уже находятся в буферах в оперативной памяти.

Другим важным критерием является сложность модификации. Индекс называется *динамическим*, если при добавлении или удалении записей не требуется его глобальная перестройка и производительность индекса не деградирует при многократных изменениях.

Еще один критерий — доля памяти, занятой данными, по отношению к общему размеру памяти, выделенной для индекса. Само по себе заполнение памяти не так важно, как время доступа, однако при низкой заполненности непомерно увеличивается количество страниц, занимаемых индексом, что косвенно влияет и на время доступа.

11.2.1. Одномерные индексы

Индекс называется *одномерным*, если значения индексного ключа рассматриваются как скалярные. Особенность одномерных ключей состоит в том, что для них обычно существует естественное упорядочение. Например, такое упорядочение присуще числовым типам, текстовым строкам и моментам времени. Составные ключи, построенные из значений нескольких атрибутов с лексикографическим упорядочиванием, также являются одномерными.

B-деревья

Упорядоченные одномерные индексы, организованные в некоторое дерево, являются наиболее часто используемым видом индексов. Такие индексы обеспечивают поиск индексных записей по значению атрибута на совпадение и на попадание в заданный интервал значений. Можно также рассматривать поиск по префиксу, однако его реализация по существу не отличается от поиска по интервалу значений.

Рассмотрим структуру B^+-дерева. Напомним, что эта структура отличается от структуры B-дерева тем, что индексные записи размещаются только в листовых вершинах.

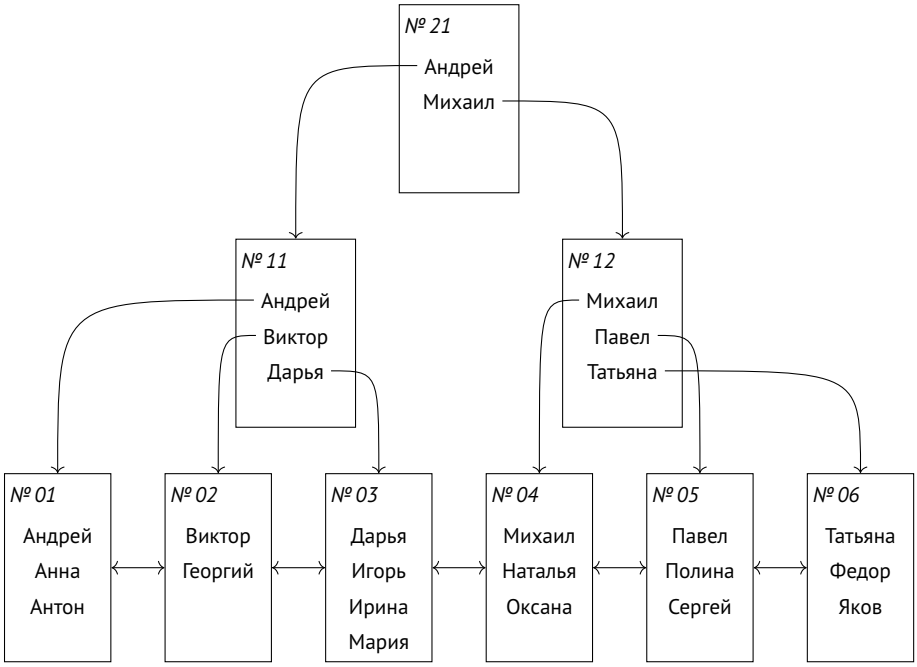


Рис. 11.2.1. Структура B^+-дерева

Пример B^+-дерева показан на рис. 11.2.1. Каждая вершина дерева соответствует странице в пространстве, выделенном для хранения индекса. Индексные записи, включенные в дерево, размещаются в соответствии с отношением порядка для индексных ключей. Если все индексные записи не помещаются в одну страницу, используется необходимое количество логически упорядоченных страниц, и при этом поддерживается отношение порядка для всех индексных ключей. Таким образом, каждая страница содержит некоторый интервал значений ключей; все ключи, попадающие в этот интервал, размещаются на этой странице. Если ключи индексных записей уникальны, то интервалы, соответствующие разным страницам, не пересекаются, иначе граничные значения интервалов в соседних страницах могут совпадать. Совокупность этих страниц составляет нижний (нулевой) уровень дерева.

Если количество страниц на уровне i превышает 1, то для каждой такой страницы создается вспомогательная индексная запись, размещаемая на уровне $i + 1$. Она включает границу интервала ключей, содержащихся на странице уровня i, и указатель на эту страницу. Различные реализации B^+-дерева могут исполь-

зовать минимальный или максимальный ключ в качестве граничного; в системе PostgreSQL используется минимальный. Полученные таким образом записи образуют уровень $i + 1$ и если этот уровень также содержит более одной страницы, то строится следующий уровень.

При добавлении новая запись вставляется в ту страницу нулевого уровня, которой соответствует значение индексного ключа. Если на странице нет места, то выполняется процедура расщепления, которая создает новую страницу нулевого уровня и переносит на нее примерно половину записей с переполненной страницы, а затем добавляет новую запись на следующий уровень (на котором также при необходимости может произойти расщепление). На рис. 11.2.2 показано состояние дерева после вставки записи и расщепления блока № 03.

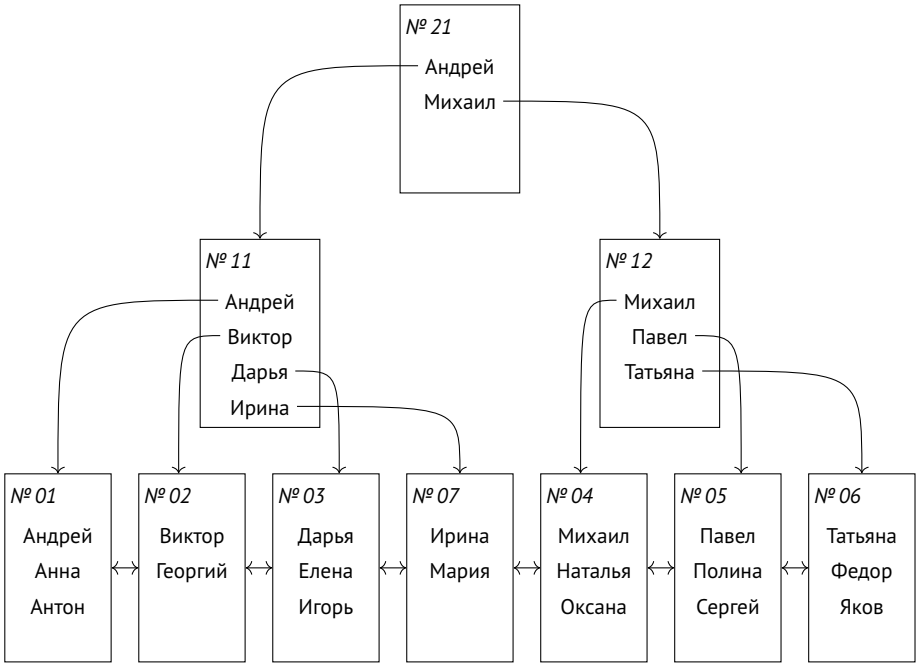


Рис. 11.2.2. Расщепление вершины B^+-дерева при добавлении записи

Легко видеть, что дерево оказывается сбалансированным, т. е. длина всех путей от корня дерева до листьев одинакова. Такая процедура гарантирует, что все страницы индекса будут заполнены не менее чем на 50 %. Можно доказать, что при поступлении новых записей в случайном порядке ожидаемая средняя заполненность блоков составит $\ln 2 \approx 0{,}69$.

Существуют варианты алгоритма, обеспечивающие более высокую степень заполнения за счет более сложных трансформаций дерева при его модификациях. Например, если вместо расщепления одной страницы на две распределять содержимое двух соседних переполнившихся страниц на три (примерно поровну), то можно получить заполнение страницы не ниже чем $^2/_3$. При этом, однако, реструктуризации становятся более частыми, и, кроме того, усложняются механизмы, обеспечивающие корректность сканирования дерева параллельно выполняемыми запросами на поиск в индексе.

Условие заполненности страниц на 50 % может нарушаться при удалениях записей. В этом случае необходимо слияние страниц, содержащих недостаточное количество индексных записей, с соседними страницами. Если процедура слияния страниц предусмотрена, то структура B^+-дерева становится динамической (в определенном выше смысле) и не деградирует при любых изменениях в данных.

Многие практические реализации, в том числе PostgreSQL, однако, слияние не делают. В результате структура индекса может деградировать и потребуется ее реорганизация, при которой все дерево строится заново. По-видимому, отказ от динамического сжатия связан с тем, что необходимость в нем возникает относительно редко: базы данных обычно имеют тенденцию расти, а не сокращаться. Тем не менее деградация дерева большого размера может существенно ухудшить производительность системы, поэтому администратору базы данных необходимо следить за состоянием индексов.

Алгоритм поиска в B^+-дереве начинает работу с единственной страницы верхнего уровня (корня дерева), выбирая на ней ключ, указывающий на страницу следующего уровня, которая может содержать искомый ключ. Если страницы нижележащего уровня представляются минимальными ключами (как в PostgreSQL), то это — наибольший ключ, не превосходящий искомый, а если максимальным — то наименьший, превосходящий искомый. Структура страниц, применяемая в PostgreSQL, дает возможность внутри страницы использовать алгоритм бинарного поиска. Если запрос предполагает поиск по интервалу, то выполняется поиск левого (или правого, в зависимости от вида условия) конца интервала.

Далее выбирается страница следующего уровня, на которую ссылается указатель, связанный с найденным ключом. Спуск повторяется до тех пор, пока не будет достигнут нулевой уровень, на котором поиск по точному значению заканчивается (положительным или отрицательным результатом), а поиск по интервалу продолжается последовательным просмотром нулевого уровня вплоть

до ключа, который окажется больше правого (меньше левого) конца интервала. Для того чтобы упростить последовательный просмотр, обычно страницы одного уровня связываются двухсторонними указателями.

Например, при поиске значения «Дмитрий» в корневом блоке № 21 будет выбран ключ «Андрей», указывающий на блок № 11, затем ключ «Дарья», указывающий на блок № 03, в котором искомый ключ отсутствует, и будет возвращен результат, указывающий, что искомого ключа в индексе нет.

Количество страниц, которое будет просмотрено при спуске, равно количеству уровней и оценивается логарифмом от числа индексных записей N. Если среднее количество записей на одной странице равно m, то количество уровней в дереве будет примерно равно $\log_m N$.

Для доказательства следует заметить, что при каждом подъеме на один уровень количество страниц сокращается в m раз. Поэтому количество уровней равно наименьшему числу k, такому, что $N \leqslant m^k$.

При поиске по интервалу к этому значению прибавляется количество страниц нулевого уровня, занятых индексными записями, входящими в интервал.

Теоретически структура B-дерева является динамической, т. к. при обновлениях могут перестраиваться только страницы, находящиеся на пути к корню, что составляет небольшую часть дерева. Для реализаций, которые не делают слияние недостаточно заполненных страниц, это не совсем так, но даже в этом случае структура B-дерева обладает очень хорошими эксплуатационными свойствами.

Важно заметить, что простые варианты алгоритмов, описанные выше для индексов на основе B-деревьев и ниже для других типов индексов, недостаточны для применения в промышленных системах, в которых требуется учитывать много дополнительных требований, в первую очередь возможность конкурентной (параллельной или псевдопараллельной) обработки. В частности, в системе PostgreSQL реализация B-деревьев использует алгоритмы конкурентного доступа, описанные в [42].

Индексы на основе хеширования

Идея любого варианта метода хеширования состоит в том, что адрес, по которому размещена запись, вычисляется некоторым преобразованием ключа. Функция, выполняющая такое преобразование, называется *функцией хеширования*. Можно сказать, что функция хеширования отображает пространство

ключей (обычно потенциально очень большое, но редко заполненное) в пространство адресов значительно меньшего размера. При таком преобразовании неизбежно возникновение коллизий — разные ключи могут отображаться в один и тот же адрес.

Поскольку функция хеширования почти никогда не сохраняет упорядочение, индексы на основе хеширования можно использовать только для проверки на равенство значений атрибута. Исключением является метод многомерного хеширования LSH (locality sensitive hashing), однако этот метод крайне редко встречается в системах управления базами данных, и мы не будем его рассматривать.

Чтобы определить индексную структуру на основе хеширования, необходимо:

- задать функцию хеширования;
- определить адресное пространство (обычно в СУБД в качестве адресов используются номера страниц в файле, в котором хранится индекс, при этом каждая страница может хранить несколько записей);
- выбрать метод размещения записей переполнения (тех записей, которые не могут быть размещены по адресу, вычисленному функцией хеширования вследствие коллизий).

Если количество записей переполнения невелико, то индексы на основе хеширования обеспечивают очень быстрый доступ по точному значению ключа: функция хеширования сразу указывает страницу, на которой находится запись, поэтому сложность (и время) поиска в таком индексе не зависит от количества записей в нем. Однако сложность поиска зависит от количества записей переполнения: чем их больше, тем больше времени понадобится для поиска.

Статические варианты индексов на основе хеширования предполагают, что адресное пространство (и, следовательно, память, выделяемая для индекса) должны быть определены при создании индекса. При малой заполненности количество записей переполнения оказывается небольшим, но плохо используется выделенная память. С ростом количества записей увеличивается и количество записей переполнения, поэтому возрастает время поиска и требуется реорганизация индекса (при которой содержимое полностью переносится в новую область памяти большего размера).

Известны методы доступа на основе хеширования, допускающие динамическое расширение адресного пространства: расширяемое хеширование [24] и

линейное хеширование [43]. В системе PostgreSQL применяется вариант расширяемого хеширования.

Метод расширяемого хеширования не требует предварительного выделения памяти, блоки могут добавляться по мере роста объема данных. Кроме блоков памяти для хранения данных, для работы метода необходима небольшая таблица переадресации, которая хранится в оперативной памяти. Пусть в некотором состоянии размер файла составляет N блоков, тогда размер этой таблицы составляет 2^k позиций, где k такое, что $2^{k-1} < N \leqslant 2^k$. Другими словами, k равно количеству битов, необходимых для записи текущего размера файла, выраженного количеством блоков данных. Каждая позиция таблицы, определяемая k младшими битами функции хеширования, содержит адрес некоторого блока данных.

Если при добавлении новой записи оказывается, что для нее нет места в блоке, на который указывает таблица переадресации, то выполняется расщепление этого блока. Процедура расщепления использует дополнительный бит функции хеширования, для того чтобы распределить записи, находившиеся в расщепляемом блоке, между двумя блоками, и вносит соответствующие изменения в таблицу переадресации. При этом может потребоваться увеличение размеров таблицы переадресации в два раза.

В начальном состоянии $k = 0$, файл содержит один блок, а таблица переадресации — одну позицию, содержащую адрес этого блока. Все объекты данных, независимо от значения функции хеширования, размещаются в этом блоке. После первого расщепления $k = 1$, таблица переадресации содержит $2^k = 2$ позиции, в которых хранятся адреса двух блоков данных. Следующее расщепление также приводит к удвоению размеров таблицы, после этого $k = 2$. На рис. 11.2.3 показано, как при этом меняется таблица переадресации.

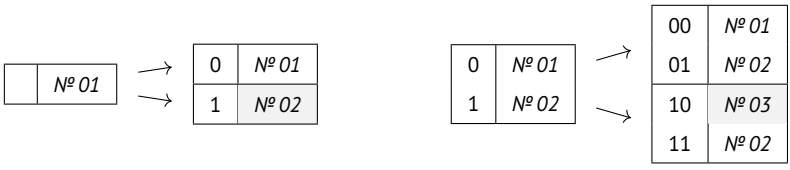


Рис. 11.2.3. Расширяемое хеширование, первые два расщепления

Конечно, в реализации хранить значения функции хеширования, показанные в левой колонке, не нужно: на рисунке они приводятся только для пояснения. Цветом на этом рисунке помечены адреса блоков, состояние которых изменено процедурой расщепления.

Если при дальнейшем добавлении записей переполнится блок № 01 (или блок № 03), то произойдет новое удвоение размеров таблицы. Если же переполнится блок, адрес которого содержится в нескольких позициях таблицы переадресации, то удвоение таблицы не понадобится: достаточно только изменить адрес в одной из позиций, содержащих адрес этого блока.

На рис. 11.2.4 показаны эти дальнейшие изменения таблицы переадресации: сначала расщепляется блок № 03 (с удвоением таблицы), затем — блок № 01 (удвоения таблицы не происходит).

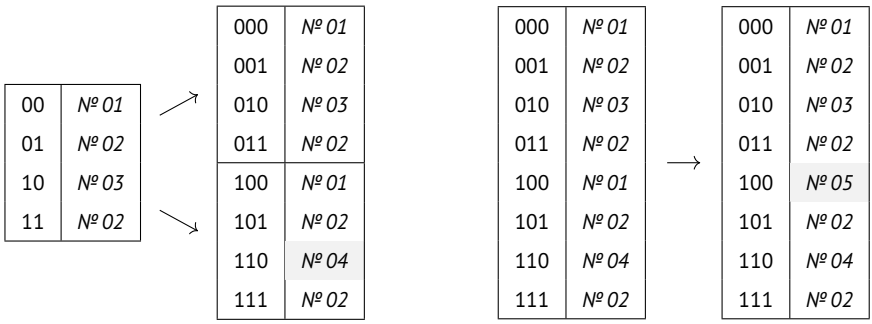


Рис. 11.2.4. Расширяемое хеширование, следующие два расщепления

Ожидаемый коэффициент заполнения блоков данных этого файла, как и для B-деревьев (и по тем же вероятностным причинам) составляет $\ln 2 \approx 0{,}69$. Существуют многочисленные разновидности метода расширяемого хеширования, позволяющие ограничить рост таблицы переадресации или увеличить заполненность файла.

Метод расширяемого хеширования гарантирует доступ к любой записи за одно обращение к блокам данных, записи переполнения не возникают. Однако этот метод чувствителен к качеству функции хеширования: при неравномерном распределении данных по блокам таблица переадресации растет слишком быстро.

В системе PostgreSQL хеширование используется как для построения индексов, так и для выполнения алгебраических операций соединения, группировки и устранения дубликатов.

11.2.2. Пространственные индексы

В отличие от одномерных многомерные индексные структуры характеризуются тем, что ключ индексной записи состоит из нескольких полей, не связанных отношением порядка. Критерии поиска могут задаваться для любого непустого подмножества ключей. Одним из распространенных классов объектов, для которых целесообразно применение многомерных индексов, являются пространственные объекты. Например, для множества точек на плоскости или в трехмерном пространстве компонентами ключа являются координаты точек.

Конечно, совсем не обязательно, чтобы поля многомерного ключа были вещественными числами (как в случае координат), — можно использовать любые скалярные типы данных, имеющие естественное упорядочение. Необходимо также, чтобы соответствующий домен содержал достаточно много различных значений, и требуется, чтобы в этом домене существовала метрика, т. е. способ вычисления расстояния между значениями.

Все многомерные индексные структуры используют разбиение множества ключей на подмножества, содержащие ключи, расположенные в определенном смысле близко друг от друга. Обычно такие подмножества организуются в некоторое дерево.

Наиболее известной, реализованной во многих СУБД и широко применяемой многомерной индексной структурой является R-дерево (R — rectangle). Эта структура предназначена для индексирования точек или прямоугольников со сторонами, параллельными осям координат (в пространствах размерности больше 2 вместо прямоугольников используются параллелепипеды). Если необходимо индексировать объекты другой формы, то вокруг них описывается прямоугольник нужной ориентации.

Прямоугольники (параллелепипеды) такого вида полностью определяются координатами концов самой большой диагонали (скажем, такой, у которой по каждой размерности значение координаты в начальной точке меньше, чем в конечной). Координаты этой пары точек являются полями индексного ключа. Таким образом, для пространства размерности k будет использовано $2k$ полей ключа.

Каждая вершина дерева представляется блоком, содержащим индексные записи, характеризующие прямоугольники, и сама описывается прямоугольником, содержащим все прямоугольники, лежащие внутри блока. Структура в целом

представляет собой сбалансированное дерево, в котором блоки, расположенные в листьях, содержат прямоугольники индексируемых объектов данных. Нелистовые вершины содержат прямоугольники, описывающие вершины следующего уровня вместе со ссылками на эти вершины.

Структура примерного R-дерева проиллюстрирована на рис. 11.2.5.

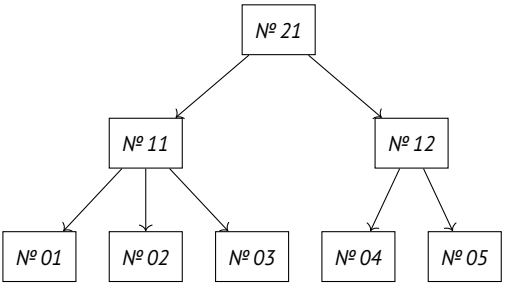


Рис. 11.2.5. Структура R-дерева

На рис. 11.2.6 для того же R-дерева показано расположение объектов (представленных геометрическими фигурами) и охватывающих прямоугольников на плоскости. Так, листовая страница № 01 содержит записи, относящиеся к объектам 1, 2 и 3, страница № 12 (на уровне 1) содержит записи, описывающие прямоугольники страниц № 04 и № 05, а корневая вершина дерева содержит описания прямоугольников страниц № 11 и № 12.

Запрос на поиск в R-дереве представляется прямоугольником такого же вида. Результатом выполнения запроса является множество объектов, прямоугольники которых имеют непустое пересечение с прямоугольником запроса. На рис. 11.2.6 запрос изображен прямоугольником, стороны которого показаны пунктиром.

Работа алгоритма поиска начинается с корневой вершины, из которой выбираются все прямоугольники, пересекающиеся с прямоугольником запроса. Далее для каждого из выбранных прямоугольников выполняется аналогичный поиск на следующих уровнях вплоть до листьев дерева. В отличие от одномерного случая (B-деревьев) спуск выполняется не по одному пути, а по нескольким.

Подчеркнем, что индексируемые объекты могут иметь любую форму или могут быть точечными (в этом случае их охватывающие прямоугольники также вырождаются в точку), однако ключ в индексной структуре R-дерева всегда будет прямоугольником. Это приводит к тому, что поиск в R-дереве может возвращать объекты, которые не должны входить в результат запроса, в тех случаях,

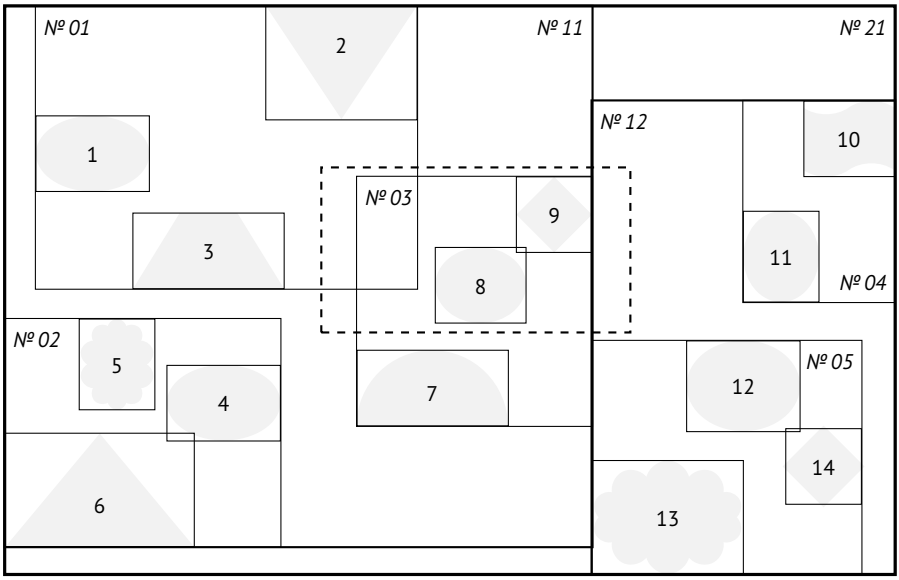


Рис. 11.2.6. Расположение объектов R-дерева

когда сам объект не пересекается с запросом, но охватывающий его прямоугольник — пересекается. Аналогично, запрос на поиск объектов в некоторой непрямоугольной области заменяется на прямоугольный запрос, охватывающий эту область, что также может вызвать включение лишних объектов в результат. Важный пример таких запросов — поиск объектов, находящихся не дальше, чем на заданном расстоянии от указанной точки. Этот запрос представляется кругом, но для поиска в R-дереве круг заменяется на описанный вокруг него квадрат.

Для того чтобы добавить новый объект, выполняется процедура поиска с охватывающим прямоугольником этого объекта в качестве прямоугольника запроса, причем поиск останавливается на предпоследнем уровне (т. е. выполняется поиск блоков, а не объектов). Прямоугольники разных блоков могут пересекаться, поэтому в результате может быть найдено несколько блоков. Поскольку охватывающие прямоугольники блоков на всех уровнях, кроме корневого, обычно не покрывают все пространство. может оказаться, что новый объект не пересекается ни с одним из имеющихся блоков. В любом случае процедура вставки выбирает один из блоков и, если объект не помещается в охватывающий прямоугольник полностью, то охватывающий прямоугольник блока расширяется.

Для того чтобы R-дерево не вырождалось при модификациях, задаются минимальное и максимальное количество записей в каждой вершине (кроме корневой, в которой ограничение снизу не применяется). Если при добавлении нового объекта в вершине нет места, выполняется процедура расщепления, которая распределяет объекты из старой вершины между двумя новыми, затем удаляет старую запись и добавляет две новые в блок более высокого уровня. Этот блок в случае необходимости также расщепляется, а при расщеплении корня добавляется новый уровень дерева.

В отличие от одномерного случая (B-дерево), в котором процедура расщепления переносит в новый блок примерно половину записей с большими ключами, расщепление для R-деревьев весьма нетривиально. Известны различные эвристические критерии качества полученного разбиения, в том числе:

- минимальный суммарный периметр;
- минимальная суммарная площадь;
- минимальная площадь пересечения;
- минимальная площадь, не занятая объектами следующего уровня.

В течение более чем 30 лет развития было предложено большое количество алгоритмов расщепления для разнообразных модификаций R-дерева, различающихся по вычислительной сложности и по качеству получаемого результата.

Также как B-деревья, R-деревья являются динамической индексной структурой при условии, что применяется процедура слияния недостаточно заполненных блоков. Если же, как в большинстве реализаций, слияние не применяется, то R-дерево может деградировать. Кроме возникновения недостаточно заполненных блоков может возникать и другой вид деградации: охватывающие прямоугольники, занимающие бóльшую область пространства, чем необходимо (эта ситуация может возникать при удалении объекта, находящегося на краю охватывающего прямоугольника блока, содержавшего этот объект).

Упомянем один из дополнительных эвристических приемов, позволяющих улучшить структуру R-дерева. Идея повторной принудительной вставки (forced re-insert) состоит в том, что объект, отстоящий слишком далеко от других объектов, находящихся в том же блоке (и, следовательно, сильно увеличивающий размеры охватывающего прямоугольника) удаляется из этого блока и выполняется его повторная вставка, при которой он, возможно, попадет в другой блок. Неформальное объяснение этого приема состоит в том, что на начальных фазах заполнения дерева, когда объектов (и, следовательно, блоков) немного,

объекты могут находиться на относительно большом расстоянии друг от друга, и поэтому необходимы охватывающие прямоугольники большого размера. С ростом дерева количество блоков и плотность размещения объектов в пространстве увеличиваются, поэтому размеры прямоугольников могут уменьшаться.

Индексы на основе R-деревьев хорошо работают с объектами небольшого размера (или точечными) в пространствах небольшой размерности, однако могут оказаться малополезными по одной или нескольким из следующих причин:

- Охватывающие прямоугольники индексируемых объектов имеют большие пересечения. В этом случае никакая структура, использующая прямоугольники в качестве поисковых образов, не сможет различить такие объекты.
- Объекты большого размера, занимающие малую часть площади охватывающего прямоугольника (например, участки дорог). В этом случае индекс будет возвращать большое количество объектов, на самом деле не пересекающихся с поисковым прямоугольником.
- В задачах поиска объектов, находящихся на расстоянии, не превышающем заданное, естественным поисковым запросом является круг, а не прямоугольник, поэтому результат применения индекса требует дополнительной фильтрации.

Последняя из перечисленных причин становится решающей при увеличении размерности пространства. Дело в том, что отношение объема гиперсферы к объему охватывающего ее гиперкуба очень быстро падает с ростом размерности, соответственно, возрастает доля нерелевантных объектов, возвращенных индексом. Известно, что любая индексная структура на основе дерева или любого разбиения пространства деградирует с ростом размерности, так что поиск становится более медленным, чем полный просмотр всей коллекции объектов. Этот факт образно называют «проклятьем размерности». В литературе описаны структуры, которые могут работать в бóльших размерностях, чем R-деревья, однако ни одна из известных структур такого типа не может эффективно работать в пространствах, размерности которых измеряются сотнями.

Имеются также структуры, позволяющие выполнить запрос приближенно (т. е. некоторые объекты, удовлетворяющие условию поиска, могут быть не найдены с помощью такого индекса). Такие структуры, однако, не реализуются в составе систем управления базами данных.

11.2.3. Инвертированные индексные структуры

Во многих классах задач объекты, среди которых выполняется поиск, характеризуются набором поисковых признаков (ключей), причем размер этого набора не фиксирован, т. е. разные объекты могут иметь различное количество ключей. Для поиска таких объектов используются *инвертированные индексы*. Наиболее широко известные применения этой структуры связаны с задачами поиска текстов на естественных языках, но, конечно, возможные применения этим не ограничиваются. Можно, например, использовать эту структуру для ускорения поиска множеств по заданному подмножеству, а также для решения некоторых задач на графах. Первые структуры данных такого типа применялись еще в конце 50-х гг. и получили название *инвертированных файлов*.

Описывать структуру инвертированного индекса удобнее всего на упрощенном примере индексирования текстовых документов.

В задачах поиска текстов запросы строятся на основе отдельных слов, содержащихся в тексте, их комбинаций или наборов, но не текста целиком. Структура прямого и инвертированного файлов представлена на рис. 11.2.7.

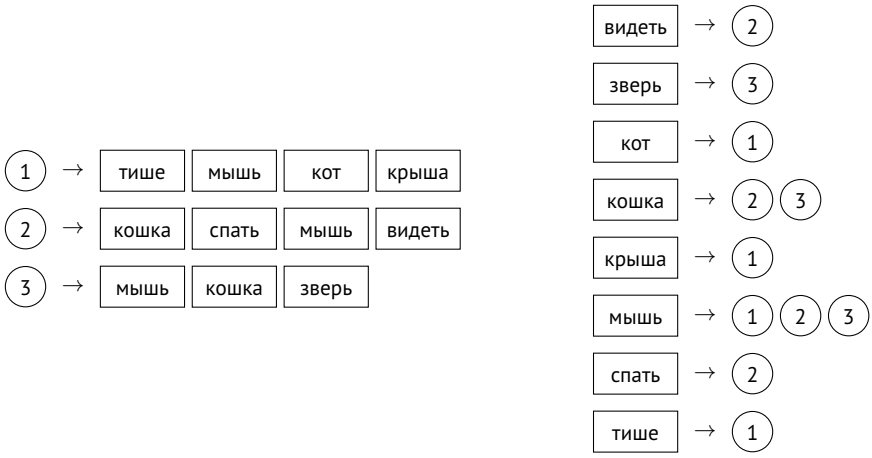


Рис. 11.2.7. Прямой и инвертированный файлы

На этом рисунке числа в кружках обозначают идентификаторы (номера) документов, а слова в прямоугольниках — идентификаторы поисковых термов (слов, характеризующих документ для поиска). В ранних системах в качестве поисковых термов могли использоваться только ключевые слова или слова,

входящие в аннотацию, а не все слова, составляющие полный текст документа. Прямой файл, таким образом, сопоставляет каждому документу последовательность слов, характеризующих этот документ, а инвертированный — наоборот, позволяет по слову найти список документов, в которых встречается это слово, т. е. является обратным (обращенным, инвертированным) по отношению к прямому файлу. Конечно, инвертированный файл не является обратной функцией в математическом смысле.

Списки документов в подобных структурах принято называть *пост-листами*. Заметим, что порядок слов в прямом файле может отражать порядок этих слов в документе, а относительное расположение номеров документов в пост-листах не может иметь никакой семантической нагрузки, поэтому пост-листы обычно упорядочиваются по возрастанию номеров документов.

Запрос, содержащий набор слов, обрабатывается следующим образом: из инвертированного файла считываются пост-листы, соответствующие этим словам, и затем вычисляется их пересечение. Поскольку все пост-листы упорядочены одинаково, для вычисления пересечения можно использовать алгоритм слияния, что требует однократного прохода по пост-листам. Для более сложных запросов может понадобиться вычисление объединения или разности, однако в любом случае алгоритм остается однопроходным.

Большое количество поисковых термов в запросе требует обработки значительного числа пост-листов. Поскольку теоретико-множественные операции обладают свойствами коммутативности и ассоциативности, обычно применяются эвристические алгоритмы для определения последовательности, в которой обрабатываются пост-листы. Например, для вычисления пересечения может быть целесообразно начинать с самых коротких пост-листов, чтобы как можно быстрее сократить размер промежуточного результата.

В литераторе описано большое количество вариантов и модификаций индексных структур на основе инвертированных файлов. Для текстовых документов на естественном языке построение инвертированных файлов из слов, непосредственно содержащихся в тексте, не дает удовлетворительные результаты по качеству поиска. Поэтому обычно слова, извлеченные из документов, подвергаются различным видам лексической обработки: приведению к канонической форме, выделению основы и т. п.

Реализация текстового поиска в PostgreSQL содержит для этого ряд библиотечных функций для лингвистической обработки. Более детально методы полнотекстового поиска и средства их реализации, включенные в состав PostgreSQL, обсуждаются в главе 18.

11.2.4. Разреженные индексы

Индекс называется *плотным*, если он содержит ссылки на отдельные объекты индексируемой коллекции. Все индексы, рассмотренные выше, являются плотными.

Хранение коллекции может быть организовано таким образом, что объекты, содержащие совпадающие или близкие значения некоторого атрибута (или группы атрибутов), размещаются рядом, на одной или нескольких соседних страницах, выделенных для хранения коллекции. В таких случаях в индексной записи вместо ссылок на отдельные объекты достаточно поместить только ссылку на область памяти, содержащую объекты с ключом, включенным в индекс. Такие индексы называются *неплотными* или *разреженными*. Как правило, разреженный индекс содержит одно значение ключа для каждой страницы данных.

11.2.5. Сигнатурные индексы

Рассматриваемая в этом подразделе индексная структура предназначена для ускорения *запросов включения* (containment queries). Предполагается, что значения некоторого атрибута таблицы представляют собой конечные множества. Например, если строка таблицы описывает текстовый документ, то значением этого атрибута может быть множество слов, встречающихся в документе.

Предполагается также, что все значения этого атрибута являются подмножествами некоторого универсального множества U. Для текстовых документов таким множеством будет множество всех слов, которые могут встречаться в документах.

Пусть задано некоторое множество $Q \subset U$. Любое такое множество определяет запрос включения, результат выполнения которого состоит из строк, в которых значение рассматриваемого атрибута содержит Q (как подмножество). Если запрос представляет собой множество слов, то ответом на этот запрос будет набор всех документов, каждый из которых содержит все слова, включенные в запрос.

Для того чтобы проверить, что множество A входит в результат выполнения запроса Q, нужно вычислить теоретико-множественную разность $Q \setminus A$. Множество A удовлетворяет запросу тогда и только тогда, когда эта разность является

пустым множеством. Чтобы не выполнять (ресурсоемкое) вычисление разности для каждого значения нашего атрибута, строится *сигнатурный индекс*.

Сигнатурой множества A называется битовая карта фиксированной длины $s(A)$, которая строится таким образом, что если два множества связаны отношением включения $A \subset B$, то $s(A) \leqslant s(B)$ в каждом бите сигнатуры. Другими словами, если сигнатура $s(B)$ в некоторой позиции содержит 0, то $s(A)$ тоже должна содержать 0 в этой позиции.

Сигнатура множества вычисляется побитовой логической операцией OR, применяемой к сигнатурам всех элементов множества (например, слов документа), а сигнатуры элементов множества вычисляются с помощью функции хеширования.

Для того чтобы метод хорошо работал, функция хеширования должна вырабатывать сигнатуру, содержащую малое количество ненулевых битов (иначе в результате применения операции OR сигнатуры множеств будут содержать слишком много единичных битов и поэтому множества будут плохо различимы). Обычно количество ненулевых битов задается заранее, а функция хеширования вырабатывает номера битов, которые устанавливаются в единицу. В реализации сигнатурных индексов в системе PostgreSQL сигнатуры элементов содержат ровно один ненулевой бит.

Сигнатурный индекс в системе PostgreSQL содержит все сигнатуры индексируемого атрибута, организованные в дерево с помощью обобщенной индексной структуры GiST. Результатом поиска в таком индексе является набор строк таблицы, для которых $s(Q) \leqslant s(A)$ (для каждого бита сигнатуры).

При фильтрации по сигнатурному индексу не гарантируется включение $Q \subset A$, потому что разные множества могут отображаться в одинаковые сигнатуры. Поэтому после применения индекса обычно необходима дополнительная проверка того, что включение множеств действительно имеет место. Проверка не требуется, если каждому биту сигнатуры соответствует не более одного слова; ее необходимость определяется в PostgreSQL автоматически.

Если функция хеширования элементов (слов) вырабатывает, как в PostgreSQL, сигнатуру с одним единичным битом, то разные элементы могут отображаться в одинаковые сигнатуры. Если количество различных элементов велико, то такие коллизии будут частыми, и поэтому индекс будет давать большое количество ложных кандидатов.

Если функция хеширования вырабатывает больше одного ненулевого бита, то ложные кандидаты могут появляться также при наличии общих единичных битов в сигнатурах разных слов. Например, допустим, что слово «кошка» имеет сигнатуру 1001, слово «мышь» — сигнатуру 0101, а слово «собака» — сигнатуру 1100. Тогда документ, описывающий отношения кошек и мышей, будет иметь сигнатуру 1101 и будет найден по запросу «собака».

Доля ложных срабатываний может оказаться очень большой, что ограничивает применимость сигнатурных индексов. Во многих ситуациях применение инвертированных индексов на основе GIN может оказаться предпочтительным.

11.2.6. Особенности реализации индексов в PostgreSQL

Реализация индексов в системе PostgreSQL решает две задачи: с одной стороны, эта реализация обеспечивает высокоэффективное выполнение запросов и учет особенностей отдельных типов индексов при выборе плана выполнения запроса и, с другой стороны, обеспечивает возможность расширения набора функций системы путем реализации новых индексных структур. Для достижения этих трудно совместимых целей потребовались инженерные решения, включающие многоуровневую систему моделей и понятий.

С точки зрения исполнителя запросов применение любого индекса сводится к получению адреса или набора адресов строк, удовлетворяющих критерию поиска, переданному в индексную структуру. Адреса строк могут использоваться для выборки данных непосредственно из таблиц или для построения битовых карт, но в любом случае внутренняя структура индекса не имеет никакого значения. Битовые карты полезны при выполнении запросов, в которых выбирается значительная доля кортежей, а также в тех случаях, когда целесообразно применение нескольких различных индексов.

Индексы могут также использоваться для выборки значений индексных ключей — в этом случае значения можно взять из самого индекса, не переходя к таблице. Однако в системе PostgreSQL применяется хранение множественных версий строк таблиц, поэтому при выборке значений из индексов, вообще говоря, требуется дополнительная проверка того, что эти значения принадлежат версиям строк таблицы, которые видны текущей транзакции. Проверка видимости всех выбираемых значений сделала бы бессмысленным такое

использование индексов. Для того чтобы сократить затраты на эти проверки, в PostgreSQL используется вспомогательная структура данных (карта видимости), которая позволяет обойтись без проверок для строк, размещенных на страницах, на которых ни одна из строк не обновлялась в течение достаточно длительного времени, и поэтому все строки на таких страницах видны всем транзакциям.

Внутренняя организация индекса определяется методом доступа, отвечающим за поддержку структуры хранения, а также выдающим информацию о свойствах и стоимости выполнения операций (в основном — поиска) в этом индексе. Эта информация необходима для выбора оптимального плана выполнения запроса. Кроме этого, метод доступа отвечает за реализацию транзакционных свойств индекса, обеспечивающих корректность работы при одновременном доступе к индексу нескольких транзакций.

Реализация новых методов доступа является, таким образом, достаточно сложной задачей. Для того чтобы упростить ее решение, в PostgreSQL реализованы обобщенные методы доступа.

Возможность добавления новых типов методов доступа на основе деревьев обеспечивается в PostgreSQL механизмами GiST и SP-GiST. По существу, эти средства обеспечивают поддержку структур типа дерева, согласованных с другими частями PostgreSQL: управлением памятью, транзакционностью и пр. Отличие GiST от SP-GiST в том, что GiST поддерживает сбалансированные деревья (такие, как R-деревья), а SP-GiST позволяет строить несбалансированные деревья (такие, как k-d-деревья или k-d-B-деревья [52]). GiST также обеспечивает выполнение запросов на поиск ближайших соседей (k-nn, k-nearest neighbours).

Обобщенными методами доступа являются также GIN, обеспечивающий построение инвертированных индексов (раздел 11.2.3). Как и в случае обобщенных индексов GiST и SP-GiST, создание нового типа индекса сводится к разработке ряда функций, реализующих операции, зависящие от индексируемого типа данных.

Создание типов неплотных индексов обеспечивается обобщенным методом доступа BRIN, входящей в состав PostgreSQL. Также, как для GiST, SP-GiST и GIN, реализация типа индекса на основе BRIN сводится к разработке нескольких функций, определяющих способы выполнения операций поиска и модификации индекса для конкретного типа данных.

Наконец, метод доступа может быть применен для различных типов данных, в том числе определенных пользователем. Для этого описываются функции,

представляющие семантику типа данных, подлежащего индексированию. Набор таких функций, связывающих метод доступа с типом данных, называется *классом операторов*. Так, для того чтобы обеспечить работу индексов на основе B-деревьев, необходимо определить операторы проверки на равенство и упорядочение значений типа данных, подлежащего индексированию.

Особенности организации и применения индексов в системе PostgreSQL обсуждаются в [69]. Возможности и применение обобщенных индексных структур в PostgreSQL рассматриваются в главе 17.

11.3. Выполнение алгебраических операций

11.3.1. Алгебраические операции и алгоритмы

Выполнение любого запроса начинается с операций выборки хранимых данных. Поскольку реляционная модель данных не занимается вопросами хранения, эти операции не могут считаться частью реляционной алгебры, однако обычно при их выполнении учитываются условия, которым должны удовлетворять данные, необходимые для выполнения запроса.

Реляционная алгебра и в особенности алгебра SQL предусматривает большой ассортимент бинарных операций (т. е. операций с двумя аргументами), однако для их выполнения используются модификации одних и тех же основных алгоритмов. В этом разделе рассматриваются три алгоритма просмотра отношений, которые применимы для вычисления:

- декартова произведения;
- внутреннего и внешних соединений;
- теоретико-множественной разности;
- пересечения.

Модификации тех же алгоритмов можно применять для операций группировки и устранения дубликатов. Эти операции не являются в точном смысле бинарными, но для их выполнения требуется сопоставлять разные кортежи из аргумента. Поэтому для них применимы те же алгоритмы, в которых в качестве второго аргумента используется частично вычисленный результат операции.

Отметим, что для всех алгоритмов бинарных операций, рассматриваемых в этом разделе, существуют эффективные параллельные версии, которые обсуждаются в главе 22.

11.3.2. Операции выборки данных

Основной операцией доступа к коллекциям является фильтрация по некоторому условию на атрибуты. При отсутствии такого условия результатом выполнения операции является множество всех элементов коллекции. Во многих руководствах эта операция называется *селекцией* (например, в литературе по теоретической реляционной модели эта операция обычно обозначается буквой σ). Однако термин «селекция» может вызвать некорректные ассоциации с ключевым словом SELECT языка SQL. Наиболее точно эта операция может быть охарактеризована выражением «выборка данных, возможно, по условию».

Здесь мы рассматриваем фильтрацию применительно к коллекциям, хранимым в базе данных. Для фильтрации промежуточных результатов выполнения запроса применяются другие алгоритмы. В зависимости от выбранного метода хранения и условий фильтрации могут быть применимы некоторые из перечисленных ниже алгоритмов. Не все алгоритмы применимы для любой ситуации, не все алгоритмы реализованы и используются во всех СУБД, и в некоторых СУБД могут использоваться и другие алгоритмы.

Полный просмотр. Последовательно считываются все страницы, выделенные для коллекции, с проверкой условия фильтрации для всех объектов на каждой странице. Пример плана запроса, использующего полный просмотр, в системе PostgreSQL:

```
demo=# EXPLAIN (costs off)
SELECT * FROM flights;
     QUERY PLAN
--------------------
 Seq Scan on flights
```

Доступ по адресу. Для каждого адреса объекта, полученного из индекса по критерию фильтрации, выбирается страница, содержащая объект, и, возможно, выполняется просмотр всех объектов на этой странице для выборки других объектов, удовлетворяющих условию фильтрации.

```
demo=# EXPLAIN (costs off)
SELECT * FROM flights
WHERE flight_id = 12345;
```

```
                QUERY PLAN
-----------------------------------------
 Index Scan using flights_pkey on flights
   Index Cond: (flight_id = 12345)
```

Просмотр страницы нужен, для того чтобы исключить повторный доступ к той же странице (например, если условие, для которого применяется индекс, задает диапазон значений). Просмотр страницы также необходим при использовании неплотных индексов, которые локализуют блоки, содержащие искомые записи, а не отдельные записи. В системе PostgreSQL эти задачи решаются другим способом с применением битовых карт, которые строятся при просмотре индексов:

```
demo=# EXPLAIN (costs off)
SELECT * FROM boarding_passes
WHERE ticket_no = '0005435212351';
                        QUERY PLAN
-----------------------------------------------------------
 Bitmap Heap Scan on boarding_passes
   Recheck Cond: (ticket_no = '0005435212351'::bpchar)
   ->  Bitmap Index Scan on boarding_passes_pkey
         Index Cond: (ticket_no = '0005435212351'::bpchar)
```

Поскольку в системе PostgreSQL на странице могут находиться несколько версий одного объекта, при доступе по адресу выполняются дополнительные проверки, чтобы выбирались только корректные версии.

Доступ по ключу. Если размещение учитывает значения атрибута или атрибутов, входящих в условия фильтрации, то доступ к нужной странице возможен без использования индекса, построенного для этой таблицы.

Просмотр интервала. Если объекты размещены в соответствии с упорядоченностью атрибута, входящего в условие фильтрации, для выборки требуемых объектов достаточно выполнить просмотр только последовательно расположенных страниц, содержащих требуемые значения атрибута.

Выбор данных (только) из индекса. Выбор данных из индекса вместо чтения таблицы возможен в том случае, если индекс содержит все значения, которые должны быть выбраны из таблицы для вычисления результата запроса. Этот метод может применяться в различных вариантах:

- выборка данных по значению ключа;
- выборка диапазона значений;
- полный просмотр индекса.

Наиболее широко известный случай применения этого метода состоит в использовании составного индекса, содержащего все необходимые колонки (такой индекс называется *покрывающим*). Этот же метод может применяться и в других случаях, например для вычисления агрегатных функций (count, min, max). Напомним, что в системе PostgreSQL индексы содержат информацию обо всех версиях объекта, поэтому для получения корректных результатов требуется использовать карту видимости и в некоторых случаях обращаться к страницам данных.

Далеко не каждый тип индекса из большого разнообразия типов, имеющихся в системе PostgreSQL, пригоден для использования в этой группе алгоритмов. Пример плана запроса для индекса на основе B-дерева:

```
demo=# EXPLAIN (costs off)
SELECT ticket_no, flight_id FROM boarding_passes
WHERE ticket_no = '0005435212351';
                          QUERY PLAN
---------------------------------------------------------------
 Index Only Scan using boarding_passes_pkey on boarding_passes
   Index Cond: (ticket_no = '0005435212351'::bpchar)
```

Если при размещении объектов использована горизонтальная фрагментация, т. е. таблица хранится в виде нескольких секций (partitions), то структура хранения и алгоритм выбираются отдельно для каждой секции.

Вертикальная фрагментация, т. е. разнесение групп атрибутов по нескольким таблицам, не поддерживается явным образом в языке SQL, поскольку для этого достаточно использовать несколько таблиц (скорее всего, с одинаковыми первичными ключами).

11.3.3. Сортировка

Несмотря на то что в теоретической реляционной модели данных упорядочение кортежей в отношениях не имеет никакого значения, операции сортировки являются неотъемлемой частью любой системы управления базами данных. Необходимость упорядочивания (сортировки) данных возникает по нескольким причинам.

- Последовательность, в которой выводятся строки результата, может быть важна для приложения (и для пользователя). В языке SQL упорядочение результата запроса задается предложением ORDER BY.

- Некоторые алгоритмы выполнения алгебраических операций основаны на предположении об упорядоченности аргументов, в частности это алгоритмы на основе слияния, обсуждаемые ниже.
- Сортировка может быть необходима для построения некоторых структур хранения.

Алгоритмы сортировки можно подразделить на две группы:

Алгоритмы внутренней сортировки упорядочивают коллекции, полностью размещенные в оперативной памяти. К этой группе относятся, в частности, пирамидальная сортировка (heap sort) и быстрая сортировка (quicksort).

```
demo=# EXPLAIN (analyze, costs off, timing off)
SELECT * FROM seats
ORDER BY seat_no;
                      QUERY PLAN
-----------------------------------------------------
 Sort (actual rows=1339 loops=1)
   Sort Key: seat_no
   Sort Method: quicksort  Memory: 111kB
   ->  Seq Scan on seats (actual rows=1339 loops=1)
```

Алгоритмы внешней сортировки упорядочивают данные, которые размещены во внешней памяти (на дисках). Наиболее часто применяемым алгоритмом этой группы является алгоритм на основе слияния.

```
demo=# EXPLAIN (analyze, costs off, timing off)
SELECT * FROM bookings
ORDER BY book_date;
                        QUERY PLAN
-----------------------------------------------------------
 Sort (actual rows=262788 loops=1)
   Sort Key: book_date
   Sort Method: external merge  Disk: 8480kB
   ->  Seq Scan on bookings (actual rows=262788 loops=1)
```

Можно доказать, что вычислительная сложность задачи сортировки не может быть ниже чем $O(N \log N)$, где N — количество сортируемых объектов. Упомянутые выше алгоритмы имеют именно такую сложность. Это означает, что создать алгоритм, который работал бы существенно быстрее известных, невозможно.

Алгоритмы сортировки детально изучены в 1960–1970-х гг. и описаны практически во всех учебниках по программированию, алгоритмам и структурам данных, поэтому мы воздержимся от их изложения.

11.3.4. Алгоритм вложенных циклов

Простейший вариант алгоритма вложенных циклов показан на рис. 11.3.1 и может быть описан следующим псевдокодом:

```
FOR r IN R LOOP
    FOR s IN S LOOP
        обработать_кортежи(r,s);
    END LOOP;
END LOOP;
```

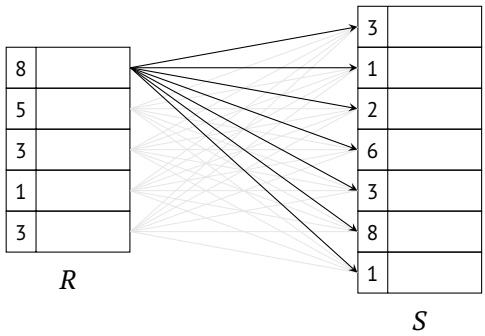


Рис. 11.3.1. Алгоритм вложенных циклов

Если функция обработать_кортежи записывает в результат кортеж, содержащий все атрибуты всех входных кортежей, то результатом этого алгоритма будет декартово произведение. Сложность этого алгоритма, очевидно, пропорциональна произведению кардинальностей множеств: $\mathrm{card}(R)\,\mathrm{card}(S)$. Важный факт состоит в том, что, поскольку размер результата декартова произведения равен произведению размеров аргументов, получить декартово произведение алгоритмом меньшей сложности невозможно.

Другие варианты функции обработать_кортежи реализуют другие алгебраические операции. Так, если эта функция выводит строку результата только в том случае, если выполняется некоторое условие, то будет получен результат операции соединения по этому условию. Если результат выводится только при полном совпадении входных строк, то будет вычислено теоретико-множественное пересечение, и т. п.

Если в качестве второго аргумента используется накопленный в ходе выполнения операции результат (начиная с пустого), то алгоритм вложенных циклов

можно использовать для удаления дубликатов из первого аргумента. Для этого необходимо, чтобы строка первого аргумента добавлялась в результат в том и только в том случае, если она не обнаружена при просмотре частично сформированного результата.

В любом случае, однако, сложность алгоритма остается пропорциональной произведению размеров аргументов, т. к. алгоритм будет просматривать все пары независимо от того, формируется из них строка результата или нет. Важно, однако, заметить, что, в отличие от вычисления декартова произведения, для других операций (соединения, теоретико-множественных, устранения дубликатов) существуют другие алгоритмы, обладающие лучшими теоретическими оценками сложности. Такие алгоритмы обсуждаются в следующих разделах этой главы. Другими словами, алгоритм вложенных циклов можно применять для реализации многих операций, но для операций, отличающихся от декартова произведения, этот алгоритм может быть (и часто оказывается) не оптимальным.

Можно, однако, сократить количество обращений к дискам, если считывать входные данные большими порциями. Приведенный далее псевдокод содержит не два вложенных цикла, а четыре, при этом два внешних цикла считывают в оперативную память страницы первого и второго аргументов операции, а внутренние два цикла представляют собой простой алгоритм, применяемый к части аргументов, находящейся (в результате работы внешних циклов) в оперативной памяти.

```
FOR Br IN страницы(R) LOOP
    FOR Bs IN страницы(S) LOOP
        FOR r IN Br LOOP
            FOR s IN Bs LOOP
                обработать_кортежи(r,s);
            END LOOP;
        END LOOP;
    END LOOP;
END LOOP;
```

В простейшем варианте алгоритма количество просмотров второго аргумента равно мощности первого аргумента $\mathrm{card}(R)$, а в блочном варианте — количеству повторений самого внешнего цикла. Чтобы сократить это количество, следует считывать сразу несколько блоков первого аргумента и выделить для этой цели максимально возможное количество оперативной памяти.

Такой вариант алгоритма вложенных циклов в PostgreSQL не применяется.

Наиболее важный вариант алгоритма вложенных циклов применим, если:

1) выполняется операция соединения, второй аргумент которой (*S*) является хранимым отношением, а не промежуточным результатом;

2) существует индекс по атрибуту соединения для этого аргумента.

В этом случае полный просмотр второго аргумента можно заменить на доступ через индекс, и внутренний цикл ограничивается просмотром только тех объектов из *S*, которые необходимы для вычисления результата соединения.

Можно охарактеризовать этот алгоритм другим способом. Как известно из теории, операция соединения эквивалентна операции вычисления декартова произведения и последующей фильтрации (селекции) по условию соединения. В общем случае алгоритм вложенных циклов буквально реализует это тождество: вложенные циклы по существу вычисляют декартово произведение, к которому «на лету» (по мере получения отдельных кортежей, входящих в прямое произведение) применяется фильтрующее условие.

Вместо этого можно на каждой итерации внешнего цикла (по первому аргументу операции соединения) применить операцию фильтрации ко второму аргументу. Предикат этой фильтрации получается из предиката соединения подстановкой значения атрибута из первого аргумента. Таким образом, предикат фильтрации для каждой итерации внешнего цикла получается разный и сравнивает значение атрибута второго аргумента с константой, полученной из первого. После выполнения такой фильтрации внутренний цикл применяется не ко всему второму аргументу, а только к результату фильтрации, т. е. к строкам, которые должны соединиться с текущей строкой первого аргумента.

Конечно, такое изменение алгоритма имеет смысл только в том случае, если для реализации операции фильтрации можно использовать индекс, а не полный просмотр, и, следовательно, второй аргумент является хранимым отношением, а не промежуточным результатом выполнения запроса. Очевидно, что применение алгоритма селекции на основе полного просмотра фактически означало бы проверку условий фильтрации одновременно с проверкой условий соединения, что по существу эквивалентно обычному алгоритму вложенных циклов.

В приведенном ниже примере вешний цикл выполняется по отношению aircrafts_data, которое просматривается полностью. Фильтрация второго аргумента соединения (отношения seats) выполняется для каждой строки первого аргумента с помощью индекса.

```
demo=# EXPLAIN (costs off)
SELECT *
FROM aircrafts_data a
   JOIN seats s ON a.aircraft_code = s.aircraft_code;
                      QUERY PLAN
------------------------------------------------------
 Nested Loop
   ->  Seq Scan on aircrafts_data a
   ->  Index Scan using seats_pkey on seats s
         Index Cond: (aircraft_code = a.aircraft_code)
```

Этот вариант алгоритма оказывается наиболее эффективным способом соединения, если размеры первого аргумента малы (возможно, после фильтрации). Однако при больших размерах аргументов другие алгоритмы соединения оказываются более эффективными, чем алгоритм вложенных циклов.

11.3.5. Алгоритм соединения на основе сортировки и слияния

Алгоритм на основе слияния применим для операций, в которых необходима проверка на равенство или неравенство (больше, меньше и т. п.) значений атрибутов входных кортежей. К таким операциям относятся соединение (с подходящими предикатами), группировка и удаление дубликатов, а также теоретико-множественные операции объединения, пересечения и разности.

Работа алгоритма состоит из двух фаз:

Сортировка. Входные отношения сортируются по атрибутам, по котором будет производиться проверка на равенство или неравенство (т. е. по атрибутам соединения, группировки или по всем атрибутам для удаления дубликатов). При этом объекты, атрибуты которых удовлетворяют предикату с одним и тем же значением второго аргумента предиката, окажутся после сортировки расположенными рядом.

Слияние. Выполняется вариант процедуры слияния, зависящий от выполняемой операции.

При выполнении соединения входные аргументы просматриваются в порядке возрастания атрибутов сортировки, при этом для групп объектов из первого и второго операндов, для которых условие соединения истинно, выполняется алгоритм вложенных циклов. Если размеры этих групп относительно малы по сравнению с размерами отношений, то выполнение этих вложенных циклов потребует значительно меньше ресурсов, чем полный алгоритм вложенных циклов. При этом дополнительная фильтрация внутри вложенных циклов не

потребуется, так как все обрабатываемые пары объектов должны быть включены в результат (потому что условие соединения для них выполняется). Корректность этого алгоритма гарантируется тем, что все объекты, которые должны быть соединены с некоторым объектов другого аргумента, оказываются расположенными рядом после сортировки.

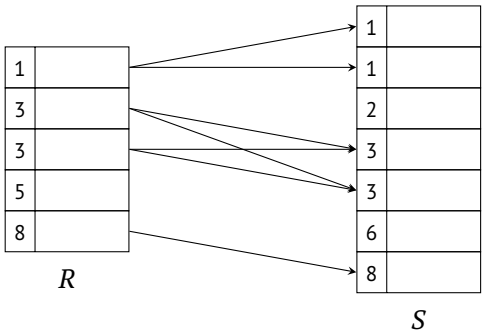


Рис. 11.3.2. Алгоритм соединения на основе слияния

Алгоритм слияния показан на рис. 11.3.2. Приведенный ниже псевдокод описывает частный случай алгоритма слияния, реализующего операцию внутреннего эквисоединения в предположениях о том, что:

1) кортежи аргументов (*R* и *S*) упорядочены по ключам соединения;

2) ключ соединения имеет уникальные значения во втором аргументе (*S*).

```
r := следующий(R);
s := следующий(S);
WHILE NOT последний(r) AND NOT последний(s) LOOP
    IF r < s THEN r := следующий(R);
    ELSIF s < r THEN s := следующий(S);
    ELSIF s = r THEN
        LOOP -- вложенный цикл
            обработать_кортежи(r,s);
            s := следующий(S);
            EXIT WHEN последний(s) OR s != r;
        END LOOP;
    END IF;
END LOOP;
```

Важная особенность алгоритма слияния состоит в том, что результат получается упорядоченным по значению ключа слияния.

В приведенном ниже примере плана запроса сортировка входных отношений выполняется в узлах Sort плана, слияние — в узле Merge Join, а сортировка результата соединения не требуется:

```
demo=# EXPLAIN (costs off)
SELECT *
FROM bookings b
   JOIN tickets t ON b.book_ref = t.book_ref
ORDER BY b.book_ref;
               QUERY PLAN
----------------------------------------
 Merge Join
   Merge Cond: (t.book_ref = b.book_ref)
   ->  Sort
         Sort Key: t.book_ref
         ->  Seq Scan on tickets t
   ->  Sort
         Sort Key: b.book_ref
         ->  Seq Scan on bookings b
```

Алгоритм на основе сортировки и слияния оказывается особенно эффективным, если один или оба входных отношения уже упорядочены так, как нужно для процедуры слияния — в этом случае фаза сортировки, естественно, исключается. Результат может оказаться отсортированным, в частности, в том случае, если предыдущая операция также была выполнена этим алгоритмом. Другой важный случай, в котором результат получается упорядоченным, — выборка диапазона значений из упорядоченного индекса.

```
demo=# EXPLAIN (costs off)
SELECT *
FROM bookings b
   JOIN tickets t ON b.book_ref = t.book_ref
ORDER BY b.book_ref;
                        QUERY PLAN
-----------------------------------------------------------
 Merge Join
   Merge Cond: (b.book_ref = t.book_ref)
   ->  Index Scan using bookings_pkey on bookings b
   ->  Index Scan using tickets_book_ref_idx on tickets t
```

Если алгоритм применяется для операций группировки или устранения дубликатов, для каждой группы объектов (из единственного в этом случае операнда) вырабатывается ровно один объект результата, поэтому вместо вложенных циклов достаточно выполнить только один цикл по группе объектов, имеющих одинаковые значения атрибутов группировки.

Теоретико-множественные операции объединения, пересечения и разности вообще не требуют вложенных циклов, поскольку решение о включении объек-

та в результат принимается на основе сравнения объектов из первого и второго аргументов (в предположении, что входные аргументы операций не содержат дубликатов).

11.3.6. Соединение на основе хеширования

Алгоритм на основе хеширования применим для тех же операций, для которых работает алгоритм на основе сортировки и слияния, с дополнительным условием на предикат соединения, который в этом случае должен быть условием равенства значений атрибутов. Другими словами, алгоритм применим для тех операций, где требуется проверка на совпадение значений всех или некоторых атрибутов входных отношений.

Базовый вариант алгоритма включает две фазы:

Распределение. Объекты первого аргумента размещаются в корзинах в соответствии со значениями функции хеширования.

Проверка. Объекты второго аргумента поочередно хешируются и сопоставляются со всеми объектами первого аргумента, хранящимися в соответствующей корзине. В случае совпадения значений атрибутов формируется объект результата.

Алгоритм можно описать следующим псевдокодом:

```
-- распределение
FOR r IN R LOOP
    B = номер_корзины(r);
    вставить r в корзину B хеш-таблицы;
END LOOP;

-- проверка
FOR s IN S LOOP
    B = номер_корзины(s);
    FOR b IN B LOOP
        обработать_кортежи(s,b);
    END LOOP;
END LOOP;
```

Выполнение любой версии алгоритма на основе хеширования включает применение функции хеширования к атрибутам входных объектов, подлежащих сравнению, распределение объектов по корзинам в соответствии со значениями функции хеширования и применение алгоритма вложенных циклов в каждой корзине. Корректность такого алгоритма гарантируется тем, что объекты,

попадающие в разные корзины, имеют разные значения функции хеширования и, следовательно, значения сравниваемых атрибутов также различаются. Поэтому объекты, попавшие в разные корзины, не могут образовать пару, подлежащую включению в результат.

Оценка сложности этого алгоритма включает однократный полный просмотр обоих аргументов для распределения по корзинам и сумму стоимостей применения алгоритма вложенных циклов в каждой корзине. При большом количестве корзин и достаточно равномерном распределении объектов по корзинам эта сложность будет существенно ниже, чем сложность алгоритма вложенных циклов.

Схема работы алгоритма хеширования представлена на рисунке 11.3.3.

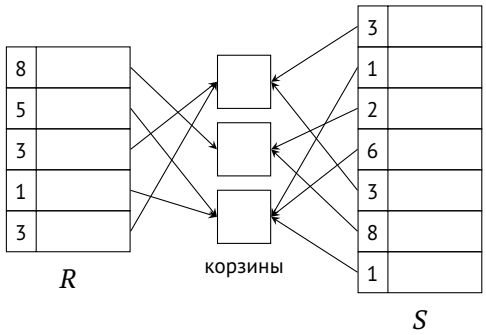


Рис. 11.3.3. Алгоритм соединения на основе хеширования

В приведенном ниже примере плана запроса, использующего соединение на основе хеширования, распределение выполняется в узле Hash плана, а проверка — в узле Hash Join:

```
demo=# EXPLAIN (costs off)
SELECT *
FROM bookings b
   JOIN tickets t ON b.book_ref = t.book_ref;
               QUERY PLAN
-----------------------------------------
 Hash Join
   Hash Cond: (t.book_ref = b.book_ref)
   ->  Seq Scan on tickets t
   ->  Hash
         ->  Seq Scan on bookings b
```

Более сложные варианты алгоритма предназначены для операций с таблицами очень большого размера.

Так, алгоритм гибридного соединения на основе хеширования сначала распределяет входные аргументы по корзинам и объединяет корзины в группы таким образом, чтобы каждая группа корзин могла быть одновременно загружена в оперативную память. Далее для каждой группы корзин выполняется обычный алгоритм соединения на основе хеширования. При этом второй аргумент можно либо предварительно распределить по группам, либо применить многократный просмотр второго аргумента (отдельный просмотр для каждой группы корзин).

Базовый алгоритм хеширования начинает вырабатывать результаты после полного считывания первого аргумента. В некоторых случаях такая задержка оказывается нежелательной. Симметричный алгоритм соединения на основе хеширования использует два набора корзин, по одному для каждого из аргументов. Считывание обоих входных аргументов начинается одновременно (поочередно или параллельно). К каждому прочитанному кортежу применяется функция хеширования, затем этот кортеж помещается в корзину своего аргумента и сопоставляется со всеми кортежами из соответствующей корзины другого аргумента.

Предположим, что пара кортежей *r, s* удовлетворяет предикат соединения, т. е. должна войти в результат. Если *r* прочитан раньше, чем *s,* то пара будет найдена при обработке *s,* потому что *r* уже записан в корзину первого аргумента. Если же *r* поступит позже, чем *s,* то пара будет найдена при просмотре корзины второго аргумента. Следовательно, этот алгоритм корректно строит результат операции соединения.

По сравнению с базовым алгоритмом симметричный требует значительно больше оперативной памяти (т. к. в корзинах накапливаются оба аргумента, а не только меньший по размеру), однако он начинает возвращать результаты, как только будет найдена первая пара подходящих входных кортежей.

11.3.7. Многопотоковое соединение

Все приведенные выше алгоритмы реализуют операцию соединения двух отношений. Однако во многих запросах необходимо соединять большее количество отношений, поэтому на первый взгляд кажется полезным ввести также алгоритмы, выполняющие соединение более чем двух входных отношений.

Действительно, расширить алгоритмы (например, вложенных циклов и на основе сортировки и слияния) таким образом, чтобы они могли обрабатывать

большее количество входных отношений, совсем несложно. Однако в промышленных системах управления базами данных такие модификации применяются редко.

Причина состоит в том, что количество операций сравнения, которые необходимо выполнить для вычисления результата, растет экспоненциально с ростом числа входных потоков. Поскольку время работы любых алгоритмов соединения обычно определяется временем процессора, необходимого для их выполнения (а не временем обмена с дисками), применение многопоточных операций оказывается менее эффективным.

Однако в случае алгоритма слияния можно поддерживать упорядочение входных отношений по значениям ключа последнего прочитанного объекта из каждого отношения. В этом случае экспоненциальный рост сложности не происходит. Поэтому многопоточное соединение для алгоритма слияния может иметь смысл, и оно действительно применяется в системе PostgreSQL при выполнении внешней сортировки.

11.4. Итоги главы и библиографические комментарии

В главе представлены основные структуры хранения, применяемые в системах управления базами данных, структуры индексов и алгоритмы поиска, а также базовые варианты алгоритмов выполнения основных алгебраических операций (соединения, произведения, удаления дубликатов, группировки и теоретико-множественного пересечения, объединения и разности). Структуры хранения, используемые в настоящее время в системе PostgreSQL, во многом основаны на идеях, реализованных в раннем прототипе [57].

Структура B-дерева, впервые описанная в [8], является первой динамической индексной структурой, наиболее известной и широко применяемой в самых различных СУБД. Публикации, описывающие улучшения для некоторых специальных случаев, продолжают появляться до настоящего времени.

Индексные структуры на основе методов хеширования применялись начиная с ранних СУБД, однако динамические варианты индексов на основе хеширования появились только в конце 70-х гг.: расширяемое хеширование [24] и линейное хеширование [43]. Заметим, что, несмотря на существенные преимущества этих структур, динамическое хеширование применяется на практике достаточно редко.

Среди пространственных индексных структур наиболее значительное распространение получили R-деревья, введенные в [33]. Улучшенный вариант этой структуры, R*-деревья, предложен и исследован в [60]. Исследования различных алгоритмов расщепления переполненных вершин для R-деревьев продолжаются до настоящего времени. Несбалансированные структуры для многомерного индексирования (k-d-B-деревья) предложены в [52].

Анализ различных алгоритмов выполнения алгебраических операций можно найти в [35] и в [46], а также в ранней работе [32].

11.5. Упражнения

Упражнение 11.1. Создайте таблицу, содержащую географические координаты и момент времени (такие атрибуты могут быть, например, у фотографии). Заполните таблицу искусственно сгенерированными значениями.

Упражнение 11.2. Для созданной таблицы постройте двумерный индекс по координатам. (Воспользуйтесь типом данных point и индексом GiST.)

Упражнение 11.3. Постройте еще один индекс по всем колонкам созданной таблицы. (Используйте расширение btree_gist.)

Упражнение 11.4. Напишите запрос, находящий фотографии, сделанные на расстоянии не более двух километров от заданной точки в течение заданного дня. Определите, как зависит скорость выполнения запроса от наличия одного или другого индекса из двух предыдущих упражнений.

Упражнение 11.5. Постройте индекс триграмм, встречающихся в индексируемом тексте. (Воспользуйтесь индексом GIN и расширением pg_trgm.)

Упражнение 11.6. С помощью команды EXPLAIN изучите планы выполнения запросов из упражнений к главе 4. Определите узлы плана, выполняющие выборку данных, соединение, сортировку. Укажите способы выполнения этих операций, которые выбрал планировщик запросов.

Глава 12

Выполнение и оптимизация запросов

12.1. Стадии обработки запроса

Напомним, что взаимодействие клиентского приложения с реляционной базой данных реализуется посредством передачи декларативных запросов, записанных на языке SQL.

Обработка любого поступившего на сервер баз данных запроса, как указано в главе 10, включает следующие этапы:

Синтаксический разбор. На этом этапе проверяется корректность синтаксиса запроса с учетом информации о схеме базы данных. Идентификаторы, встречающиеся в запросе, связываются с объектами базы данных (отношениями, атрибутами, функциями и т. п.). Результатом синтаксического разбора корректного (не содержащего ошибок) запроса является его представление в виде алгебраического выражения, эквивалентного исходному декларативному запросу.

Переписывание. На этапе переписывания выполняются эквивалентные преобразования запроса, упрощающие его дальнейшую обработку. Например, если запрос содержит вложенные подзапросы, его можно преобразовать к форме, в которой вложенных подзапросов нет.

Оптимизация. На этапе оптимизации среди множества эквивалентных алгебраических выражений, вычисляемых запросом, выбирается такое, которое имеет лучшие оценки необходимых для его вычисления затрат. Кроме этого для каждой операции выбирается алгоритм, который будет использован для вычисления выражения. Полученное в результате оптимизации выражение называется физическим планом выполнения запроса.

Интерпретация. На этом этапе выполняется интерпретация плана, т. е. вычисляется результат выполнения запроса. Этот этап часто совмещается со следующим.

Пересылка результата. Организация пересылки зависит от того, какой интерфейс используется клиентским приложением (источником запроса).

В реальных системах выполнение этих этапов может потребовать нескольких взаимодействий между приложением-клиентом и сервером баз данных. Например, это необходимо для обработки подготовленных запросов, рассматриваемых ниже в разделе 12.2.

Полученное в результате синтаксического разбора выражение содержит операции некоторой алгебры (обычно это алгебра операций SQL) и может быть представлено в виде дерева. Вершинами этого дерева являются операции, а ребра описывают структуру выражения. Каждое подвыражение представляется поддеревом, корень которого связывается ребром с той операцией, для которой это подвыражение является аргументом. Исключением является операция, находящаяся в корне, т. е. вырабатывающая окончательный результат выполнения всего запроса.

Может оказаться, что структура дерева недостаточна для представления запроса. Это происходит, если в запросе имеются общие подвыражения. В этом случае запрос невозможно представить в виде одного алгебраического выражения со скобками, а для его представления необходимо использовать направленный ациклический (т. е. не содержащий контуров) граф. Оптимизация таких запросов значительно сложнее, чем оптимизация обычных запросов, и пока мы такие запросы рассматривать не будем.

Представление запроса в виде дерева называется *планом выполнения запроса*. В некоторых системах, в том числе в PostgreSQL, планом принято называть только план, который передается на выполнение после оптимизации. Нам, однако, понадобится говорить не только об окончательном результате оптимизации, но и о других видах деревьев, представляющих запрос на различных фазах его подготовки к выполнению.

Принято различать *логические планы* выполнения, содержащие абстрактные операции, близкие к операциями реляционной алгебры, и *физические планы*, содержащие конкретные алгоритмы выполнения логических операций. Неполный список логических операций, которые могут использоваться в различных системах, включает:

- фильтрацию;
- проекцию;

- теоретико-множественные операции (объединение, пересечение, разность);
- соединение, внешнее соединение, полусоединение, антисоединение;
- группировку и удаление дубликатов;
- сортировку;
- декартово произведение.

В качестве примеров физических операций можно назвать:

- сканирование (полный просмотр) хранимой таблицы;
- сканирование индекса;
- поиск в индексе;
- доступ к строке хранимой таблицы по указателю;
- соединение алгоритмом вложенных циклов;
- сортировка;
- соединение алгоритмом слияния;
- соединение алгоритмом хеширования.

Очевидно, что перечисленные физические операции обеспечивают только часть из перечисленных логических, однако в любой СУБД предусмотрены физические операции или их комбинации для всех логических операций. Кроме этого, для разных типов индексов операции могут быть различными. Таким образом отображение логических операций на физические зависит не только от самой логической операции, но и от схемы базы данных, а в некоторых случаях и от других операций, встречающихся в том же запросе. Например, если требуется, чтобы результат был упорядочен, обычно в план включается операция сортировки, но если предыдущая операция вырабатывает результат в нужном порядке, то операция сортировки исключается из плана.

В ряде случаев одной логической операции может соответствовать несколько физических. Например, операция фильтрации данных, находящихся в хранимой таблице, может быть реализована операциями поиска по индексу с последующей выборкой данных с помощью полученных из индекса указателей.

В системе PostgreSQL операция соединения с использованием алгоритма хеширования фактически реализуется как две отдельных операции физического уровня, одна из которых выполняет распределение, а другая — проверку.

12.2. Подготовка и выполнение

Выполнение подготовительных этапов обработки запроса (от синтаксического разбора до завершения оптимизации) может требовать относительно много времени (в основном — времени процессора). Это может быть в особенности заметно для запросов, время выполнения которых невелико и которые при этом выполняются многократно. Может показаться, что многократное выполнение одного и того же запроса не имеет смысла, поскольку результат получится идентичный. Однако запросы могут быть параметризованы, и значение параметра (например, константы, используемой в запросе для фильтра) при разных выполнениях может быть разным, и, конечно, разным будет результат.

Системы управления базами данных предоставляют возможность исключить многократную подготовку запроса. Для этого обработка запроса разделяется на две фазы.

На фазе подготовки, обычно запрашиваемой клиентом с помощью функции prepare (в разных интерфейсах эта функция может иметь разные названия), выполняются все этапы обработки запроса, которые в данной системе можно выполнить без знания фактических значений параметров. Результатом подготовки является некоторое представление плана запроса, ссылка на которое возвращается клиенту.

На фазе выполнения подготовленного запроса клиент должен передать значения параметров (если запрос параметризован), и после этого сервер базы данных выполняет все оставшиеся этапы обработки.

Многие СУБД (но не PostgreSQL) применяют неявное разделение фаз подготовки и выполнения даже в том случае, если клиент запрашивает немедленное выполнение запроса. Подготовленный запрос кешируется и в дальнейшем используется, если новый запрос текстуально совпадает с исходным текстом подготовленного. Для того чтобы такое совпадение случалось чаще, константы, использованные в запросе, могут заменяться на параметры, в качестве значений которых, конечно, используются значения констант из запроса.

Фактическое распределение этапов обработки запроса между фазами подготовки и выполнения в различных СУБД может быть разным. Если оптимизация полностью завершается на фазе подготовки, то оптимизатор не может использовать значения констант при выборе плана, хотя оптимальный план может зависеть от этих значений. Например, для фильтрации по редко встречающимся в базе данных значениям целесообразно использовать индексы, а для часто встречающихся более эффективен полный просмотр. Не имея фактических значений параметров, оптимизатор не может обоснованно выбрать один из этих вариантов, если распределение значений атрибута фильтрации существенно отличается от равномерного.

В системе PostgreSQL фаза подготовки, как правило, включает этапы, которые не зависят от значений параметров запроса, а оптимизация выполняется непосредственно перед выполнением, когда все значения параметров уже известны. Но в некоторых случаях, если это представляется безопасным, PostgreSQL может отказаться от учета значений параметров и оптимизировать запрос уже на фазе подготовки.

Практически во всех высокопроизводительных системах управления базами данных выполнение запросов организовано таким образом, что отношения, получаемые в результате выполнения операций, передаются на вход следующей операции по мере их получения, строка за строкой. Выполнение плана начинается с листовых вершин (операций выборки хранимых данных), и как только эти операции начинают вырабатывать строки результата, начинается выполнение следующих (родительских) операций, и т. д. до корня дерева.

Такая организация интерпретации плана исключает необходимость хранения промежуточных результатов и открывает возможности для параллельного выполнения операций. Конечно, некоторые операции не могут обойтись без временного хранения промежуточных результатов и фактически не могут выполняться параллельно с другими операциями. Например, операция сортировки не может начать вырабатывать результат, до тех пор пока не получены все строки ее аргумента. Однако в подобных случаях хранение промежуточных данных реализуется в самой операции, а не при передаче данных между операциями.

Важным свойством такой организации исполнения является то, что затраты на передачу данных между операциями оказываются пренебрежимо малыми по сравнению с затратами ресурсов на выполнение операций, и поэтому стоимость выполнения плана можно оценивать как сумму стоимостей выполнения отдельных операций в этом плане.

Это утверждение верно в том случае, если весь запрос выполняется на вычислительной системе с общей памятью. Для распределенных систем ситуация несколько сложнее, и мы вернемся к этому вопросу позже.

В некоторых случаях (но не в PostgreSQL) передача результатов по одной строке может приводить к понижению эффективности из-за слишком частых переключений между контекстами операций. Например, это может происходить в случае хранения данных по колонкам, когда передаваемые строки являются очень короткими. В подобных ситуациях применяется буферизация данных между операциями.

12.3. Оптимизация запросов

12.3.1. Задача оптимизации

Неформально задача оптимизатора состоит в том, чтобы выбрать наилучший план выполнения запроса из нескольких (чаще — из многих) эквивалентных по результату. Существование различных эквивалентных планов определяется следующим:

- Для операций логического уровня справедливы алгебраические тождества (коммутативность, ассоциативность, дистрибутивность), позволяющие изменять структуру дерева. Конечно, не для всех операций справедливы все тождества (например, операция теоретико-множественного вычитания не коммутативна), но, как показано в главе 2, количество таких тождеств для операций реляционной алгебры весьма велико.
- Операции логического уровня могут отображаться на операции физического уровня различными способами.

Для того чтобы найти лучший план, необходим инструмент, позволяющий сравнивать разные планы. Поскольку нас интересует нахождение вычислительно эффективного плана, в качестве критерия можно использовать количество вычислительных ресурсов, необходимое для выполнения плана. Существует много различных видов вычислительных ресурсов, например можно рассматривать общее время выполнения запроса, процессорное время, затраченное на его выполнение, количество операций ввода-вывода и др. План, лучший по одному из критериев, не обязательно будет (и, скорее всего, не будет) лучшим по другим.

Обычно оптимизаторы СУБД используют некоторую взвешенную линейную комбинацию различных критериев. При этом имеется возможность влиять на выбор этой комбинации с помощью параметров сервера баз данных. Такой подход применяется и в системе PostgreSQL, в которой стоимость определяется как линейная комбинация затрат процессора и работы дисков, а относительное значение разных видов ресурсов задается параметрами сервера.

Количество ресурсов, необходимых для выполнения плана, определяет его качество с точки зрения эффективности работы сервера баз данных. Другой возможный вид критериев качества и функций стоимости предполагает оценку времени отклика. При этом обычно различают время, необходимое для получения первой строки результата, и время полного выполнения запроса. Эти два варианта времени отклика оцениваются разными функциями стоимости, и оптимальные планы могут получаться разными.

Во многих системах имеется способ выбрать, что именно будет оптимизироваться: получение первой строки или полное выполнение запроса. В системе PostgreSQL вычисляются обе функции, а в качестве критерия оптимизации применяется их комбинация, зависящая от параметра cursor_tuple_fraction.

Важно заметить, что алгоритмы оптимизации никак не зависят от выбранного критерия. Единственное, что требуется, это возможность сравнивать планы и определять, какой из них лучше по выбранному критерию. Конечно, при использовании разных критериев будут выбираться разные планы, но на построение оптимизатора это никак не влияет.

Далее в этом разделе предполагается, что определена функция, сопоставляющая каждому плану некоторое число, которое называется *стоимостью* этого плана, а сама функция называется *функцией стоимости*. Методы вычисления функции стоимости рассматриваются ниже в разделе 12.4.

Формально задача оптимизации запросов формулируется следующим образом: среди всех эквивалентных планов выполнения запроса найти такой, на котором достигается минимальное значение функции стоимости.

План, на котором достигается минимум, называется *оптимальным планом*, а процесс поиска этого плана называется *оптимизацией*. Заметим, что использование термина «оптимизация» в этом контексте отличается от его использования в обычной разговорной речи. При решении задачи оптимизации запросов предполагается, что для найденного оптимального плана доказано, что плана с лучшим значением функции стоимости не существует. При использовании слова «оптимизация» в разговорной речи оно обычно просто означает

«улучшение». Например, когда мы оптимизируем код какой-нибудь программы, мы его улучшаем, но чаще всего никому не приходит в голову утверждать, что никто никогда не сможет улучшить еще что-нибудь в нашем коде.

В предыдущем абзаце речь идет о точном решении задачи оптимизации в ее математической постановке. Эта задача, как и многие другие задачи дискретной оптимизации, является вычислительно сложной, поэтому точные алгоритмы ее решения, в том числе обсуждаемые ниже в этой главе, неприменимы для запросов большого размера. Поэтому в реальных системах (в том числе в PostgreSQL) используются и приближенные алгоритмы решения этой задачи. Заметим также, что точное решение основано на значениях функции стоимости, которая оценивает затраты ресурсов неточно, поэтому абсолютно точное решение математической задачи не всегда оказывается лучшим в реальности.

Необходимо также заметить, что оптимизация запросов имеет очень большое значение для высокопроизводительных систем. Разница во времени выполнения неудачных планов и планов, близких к оптимальному, может достигать нескольких порядков. Время выполнения может варьироваться от долей секунды до нескольких лет или десятилетий (конечно, выполнение таких планов в реальности никогда не завершается).

12.3.2. Сокращение пространства планов

Задача оптимизации запросов в формальной математической постановке относится к классу задач дискретного программирования, поскольку пространство эквивалентных планов дискретно. Как и большинство других задач этого класса, оптимизация запросов оказывается вычислительно сложной из-за больших размеров пространства планов.

Для того чтобы сократить объем и упростить работу алгоритмов оптимизации, перед запуском основных алгоритмов обычно выполняются преобразования логического плана выполнения запроса, полученного в результате синтаксического разбора. Иногда совокупность таких преобразований называется логической оптимизацией, но мы не будем использовать этот термин. Английский термин *query rewriting* намного лучше описывает происходящее на этом этапе обработки запроса. Можно выделить несколько классов таких трансформаций.

Нормализация. К этой группе относятся трансформации, делающие структуру дерева более регулярной или расширяющие запрос информацией из схемы базы данных.

Примерами таких трансформаций могут служить:

- подстановка в запрос текстов представлений;
- преобразование подзапросов в явные операторы соединения;
- проталкивание (push down) условий фильтрации внутрь подзапросов, к которым по каким-либо причинам не применяется предыдущее преобразование.

Безусловные улучшения. Некоторые трансформации заведомо не могут ухудшить значение функции стоимости, и их можно применять всегда, когда справедливы соответствующие тождества. Например, операции фильтрации и проекции требуют для своего выполнения мало ресурсов (на каждую обрабатываемую строку). Поэтому эти операции можно переставлять так, чтобы они выполнялись раньше других операций (если, конечно, они не исключают атрибуты, необходимые для последующих операций). Обе эти операции сокращают объем данных, поэтому целесообразно их перемещать как можно ближе к листьям дерева. Поскольку при этом объем данных, обрабатываемых другими операциями, сокращается, эти преобразования не могут ухудшить значение функции стоимости.

Эвристики. Некоторые трансформации и приемы, применяемые при оптимизации, существенно сокращают размеры пространства планов, но могут при этом приводить к потере оптимального решения. Применение эвристик имеет смысл, если потеря оптимального плана происходит относительно редко или если значение функции стоимости для получаемых планов не очень существенно отличается от оптимального.

Примером эвристики может служить оптимизация подзапроса отдельно от всего запроса с последующей подстановкой полученного оптимального плана для подзапроса в план основного запроса. В некоторых случаях это может приводить к планам, весьма далеким от оптимальных.

Полученный в результате переписывания запрос обрабатывается алгоритмом оптимизации.

12.3.3. Алгоритмы оптимизации

Алгоритмы оптимизации запросов могут быть точными или приближенными. Точные алгоритмы выполняют исчерпывающий просмотр пространства планов. Подчеркнем: «исчерпывающий» означает, что будет найден оптимальный

план, но не означает, что для этого будет выполнен полный просмотр пространства планов. Приближенные алгоритмы могут быть основаны на применении эвристик или на вероятностных методах.

По способу конструирования оптимального плана алгоритмы оптимизации можно разделить на следующие типы:

Снизу вверх. Такие алгоритмы начинают работу с построения оптимальных планов для небольших подзапросов и затем, конструируя оптимальные планы для все более сложных подзапросов, заканчивают работу, когда получен оптимальный план.

Трансформационные. Алгоритмы трансформационного типа начинают работу с некоторого полного плана и, применяя к нему допустимые трансформации, обходят некоторую часть пространства планов. Такие алгоритмы, как правило, являются приближенными.

Сверху вниз. Алгоритмы этого типа начинают работу с выбора завершающей операции в плане (корня дерева), а затем конструируют оптимальные планы для все более мелких подзапросов. Такие алгоритмы используются крайне редко, хотя у них есть важные привлекательные особенности.

Далее при обсуждении алгоритмов оптимизации мы будем предполагать, что запросы уже нормализованы и операции фильтрации и проекции продвинуты к листьям настолько, насколько это возможно. Заметим также, что эти операции применяются к каждой строке независимо от обработки других строк таблицы. Поэтому эти операции можно выполнять «на лету» при передаче результатов одной операции в другую в качестве аргумента.

Оптимизация снизу вверх

При обсуждении алгоритмов оптимизации «снизу вверх», следуя примеру многих других учебников, ограничим класс запросов. Будем предполагать, что запрос после нормализации содержит только операции фильтрации, проекции и соединения. Такие запросы принято называть SPJ-запросами (select-project-join). Алгоритмы могут работать и с другими классами запросов, однако включение других операций сделало бы изложение чрезмерно громоздким, не добавляя информации об алгоритме.

В основе алгоритмов, конструирующих оптимальный план, лежит следующее утверждение: *любое поддерево оптимального плана для некоторого запроса является оптимальным планом для соответствующего ему подзапроса*.

Стоимость плана равна сумме, включающей стоимость корневой операции, стоимости подзапросов, вычисляющих ее аргументы, и стоимости пересылки данных между операциями. Если бы план подзапроса оптимального плана не был оптимальным, можно было бы заменить план подзапроса на оптимальный, что понизило бы стоимость всего плана, а это противоречит предположению о его оптимальности.

Необходимо заметить, что здесь эквивалентность планов (и подпланов) рассматривается в алгебре тех операций, которые используются в плане, а не, скажем, реляционной алгебры. В частности, если некоторая операция требует, чтобы ее аргумент был упорядоченным, то эквивалентными являются только подпланы, вырабатывающие необходимое упорядочение своего результата, передаваемого в качестве этого аргумента.

Например, операция MERGE JOIN, применяемая в PostgreSQL, предполагает, что ее аргументы упорядочены. Если для аргумента этой операции имеются два плана, вырабатывающие одинаковое множество значений (т. е. одинаковые отношения), но один из планов не гарантирует требуемое упорядочение результата, то такие подпланы не будут эквивалентны и поэтому замена невозможна, даже если подплан, не гарантирующий упорядоченность, имеет более низкую оценку стоимости. Чтобы сделать такие планы эквивалентными, нужно включить операцию сортировки, которая, конечно, увеличит его стоимость.

Будем предполагать, что операции фильтрации и проекции выполняются вместе с предшествующими операциями, как описано выше. В этом предположении остается только определить порядок выполнения операций соединения. Поскольку операция соединения коммутативна и ассоциативна, возможна любая последовательность этих операций, и поэтому пространство планов получается огромным (его размер экспоненциально зависит от количества таблиц, участвующих в операциях соединения).

Заметим, что выражения $R \bowtie S$ и $S \bowtie R$ (знак $\bowtie$ обозначает операцию соединения, как было определено в разделе 2.2.2) в силу коммутативности эквивалентны по результату, но алгоритмы вычисления соединения могут быть несимметричными, и стоимость этих планов может различаться. Например, для алгоритма соединения на основе хеширования стоимость будет ниже, если первый аргумент меньше второго по размеру.

Поиск оптимального плана начинается с выбора метода доступа к хранимым данным для каждой из таблиц, упомянутых в запросе. Алгоритм полного просмотра хранимой таблицы можно использовать в любом случае, однако, если за операцией чтения в плане запроса непосредственно следуют операции

фильтрации и для некоторых из атрибутов, по которым выполняется фильтрация, в базе данных существуют индексы, то использование таких индексов может оказаться предпочтительным. Оценка стоимости операции чтения данных с помощью индекса зависит в первую очередь от селективности этого индекса: чем меньше строк таблицы останется после фильтрации, тем ниже оценка стоимости такой операции.

Если кроме фильтрации имеется операция проекции такая, что все необходимые атрибуты могут быть выбраны из индекса, то обращение к основной таблице, возможно, не потребуется, как описано в разделе 11.3.2. Это также повлияет на оценку стоимости. Существуют также алгоритмы, использующие несколько различных индексов прежде, чем происходит обращение к хранимой таблице.

Метод доступа может выбирать хранимые данные в определенном порядке. Такая упорядоченность может снизить стоимость последующих операций, потому что исключается необходимость сортировки. Такие упорядочивания принято называть *интересными*.

Для каждой таблицы, упомянутой в запросе, и для каждого интересного упорядочивания оптимизатор выбирает методы доступа наименьшей стоимости. Заметим, что одна таблица может быть упомянута в запросе несколько раз; в таких случаях выбор метода доступа делается отдельно для каждого вхождения таблицы в запрос, поскольку операции фильтрации для разных вхождений (скорее всего) различаются.

Выбор методов доступа к хранимым данным, описанный выше, является исчерпывающим, но не полным перебором, т. к. не рассматриваются всевозможные комбинации методов для разных таблиц.

После выбора методов доступа для отдельных таблиц начинается построение дерева соединений. Наиболее известным алгоритмом для этого является алгоритм динамического программирования, дающий точное решение задачи оптимизации, если не применяются дополнительные эвристики.

Для иллюстрации работы алгоритма будем использовать запрос, в котором требуется вычислить соединение трех таблиц *a, b, c*. Полный список возможных планов для этого запроса включает:

$$(ab)c,\ a(bc),\ (ac)b,\ a(cb),$$
$$(ba)c,\ b(ac),\ (bc)a,\ b(ca),$$
$$(ca)b,\ c(ab),\ (cb)a,\ c(ba).$$

Будем называть длиной подзапроса количество включенных в него таблиц. После выбора методов доступа к отдельным таблицам будут найдены оптимальные планы для подзапросов длины 1.

На каждом шаге алгоритма предполагается, что уже построены оптимальные планы для всех возможных подзапросов длины не более чем $i - 1$, и строятся оптимальные планы для подзапросов длины i. Для этого каждый подзапрос длины $i - 1$ пополняется запросом длины 1, выбирающим данные из таблицы, которая в нем еще не участвует (т. е. к подзапросу длины $i - 1$ присоединяется еще одна таблица).

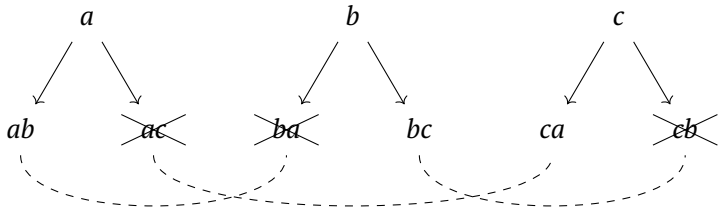


Рис. 12.3.1. Алгоритм динамического программирования после шага 2

Для каждого из полученных планов длины i вычисляется функция стоимости, и затем для каждой комбинации таблиц выбирается один план, имеющий минимальную стоимость. (В реальности дело обстоит несколько сложнее: план минимальной стоимости выбирается отдельно для каждого интересного упорядочивания, если в результате выполнения операции получается упорядоченный результат.)

На рис. 12.3.1 показано состояние после шага 2. Для упрощения на рисунках не показаны варианты, учитывающие упорядочение результатов. Дугами соединены планы, содержащие (возможно, в разном порядке) одни и те же таблицы, поэтому из каждой пары таких планов оставлен только один, представляющий наилучший план для подзапроса, содержащего эти таблицы.

Работа алгоритма завершается, когда в построенный план оказываются включены все таблицы. Результаты работы алгоритма после его завершения показаны на рис. 12.3.2.

Можно заметить, что на последнем шаге выполняется сравнение только 6 полных планов, а не 12, что потребовалось бы при полном просмотре пространства планов.

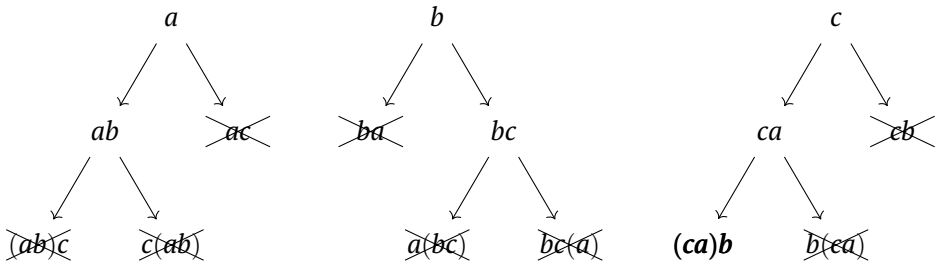


Рис. 12.3.2. Алгоритм динамического программирования после завершения

Алгоритм динамического программирования можно ускорить, ограничивая пространство рассматриваемых планов. Обычно из рассмотрения исключаются планы, содержащие вычисление декартова произведения, если только такое вычисление не требуется в исходном запросе. Для вычисления прямого произведения необходимо большое количество ресурсов, поэтому планы, содержащие такое вычисление, скорее всего, не являются оптимальными. Для реализации этого правила при конструировании новых подзапросов длины i достаточно проверить, что в исходном запросе имеется условие, связывающее атрибуты вновь присоединяемой таблицы с одной или несколькими таблицами, входящими в уже построенный подзапрос длины $i - 1$. Если таких условий нет, но есть условия, связывающие с еще не включенными таблицами, то новый план (длины i) исключается из рассмотрения.

Существенное сокращение пространства планов обеспечивает другой эвристический прием, согласно которому для любой операции соединения по крайней мере один из аргументов должен быть хранимой таблицей. Получаемые при этом планы представляют собой вырожденное дерево, в котором существует путь от корня к листу длины $n - 1$, где через n обозначено количество вершин. Такие планы называются *односторонними*, в противоположность кустистым планам (общего вида). Преимущество односторонних планов состоит в том, что для операций соединения, один из аргументов которых является хранимой таблицей, возможно использование индексов. Тем не менее ограничение односторонними планами может привести к потере оптимального плана.

Алгоритм динамического программирования в худшем случае оказывается экспоненциальным как по времени выполнения, так и по памяти, необходимой для хранения частичных планов. По этим причинам его можно применять только для запросов относительно небольшого размера, обычно содержащих не более 10–11 таблиц.

Существуют приближенные алгоритмы конструирования оптимальных планов снизу вверх. *Жадный алгоритм* начинает работу так же, как алгоритм динамического программирования: выбирает оптимальный план для выборки данных из каждого отношения, необходимого для вычисления результата запроса. Все эти частичные планы необходимы в любом плане выполнения данного запроса. Далее на каждом шаге, начиная с построения соединений двух отношений, выбирается только один частичный план с наименьшей стоимостью для последующего расширения.

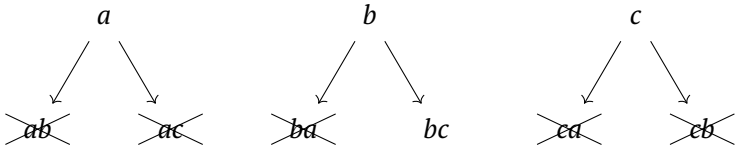


Рис. 12.3.3. Жадный алгоритм после шага 2

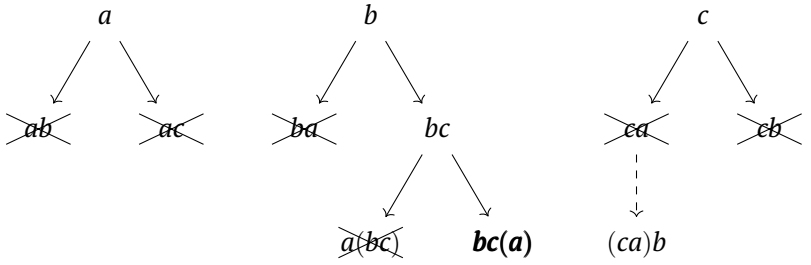


Рис. 12.3.4. Жадный алгоритм после завершения

На рис. 12.3.3 показано состояние после второго шага этого алгоритма, а на рис. 12.3.4 — состояние после его завершения (пунктиром показан оптимальный план, который не был найден).

Жадный алгоритм вырабатывает планы слишком низкого качества, поэтому он никогда не применяется в высокопроизводительных системах управления базами данных.

Семейство алгоритмов *итеративного динамического программирования* [40] дает возможность выбора между затратами на поиск оптимального плана и качеством приближенного результата. Для этих алгоритмов задается предельный размер памяти или пороговое значение времени. Алгоритм начинает работу

как обычный алгоритм динамического программирования. Когда объем памяти, занятый частичными планами, или выделенное время оказываются исчерпанными, выполняется один шаг жадного алгоритма, т. е. выбирается лучшее полученное к этому времени частичное решение, и далее алгоритм продолжает работу в режиме динамического программирования, начиная с этого частичного решения.

Трансформационные алгоритмы

Поскольку точные методы оптимизации неприменимы для запросов, содержащих большое количество операций, для их оптимизации применяются другие методы. Как правило, такие методы основаны не на конструировании планов, а на их трансформации. Все пространство планов можно представить в виде графа, вершины которого соответствуют различным планам выполнения запроса, а дуги соединяют вершины, так что план, соответствующий одной вершине, можно преобразовать в другой применением одного правила.

Граф планов для нашего простого примера представлен на рис. 12.3.5. Дуги на этом графе соединяют планы, которые можно преобразовать один в другой применением тождеств коммутативности и ассоциативности.

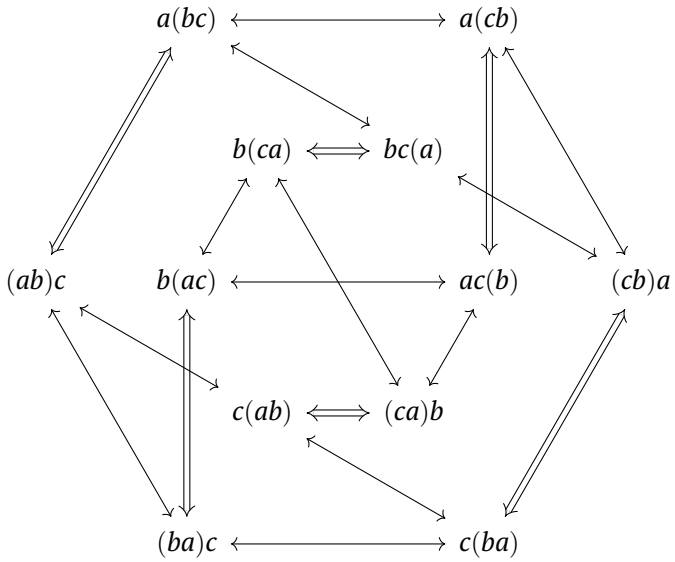


Рис. 12.3.5. Переходы в пространстве планов

Приближенные алгоритмы оптимизации строятся на основе случайных блужданий по графу трансформаций. Поиск оптимального плана начинается с некоторого плана, например с плана, полученного в результате синтаксического разбора и нормализации. Любой алгоритм представляет собой последовательность шагов, на каждом из которых выполняется переход к другому плану, при этом для всех просматриваемых планов вычисляется оценка их стоимости. Поскольку приближенные алгоритмы применяются тогда, когда размеры пространства планов не позволяют применить точные алгоритмы, полный просмотр пространства планов невозможен. Поэтому работа алгоритма заканчивается тогда, когда истекает выделенное для работы оптимизатора время. Результатом оптимизации становится тот из просмотренных планов, оценка стоимости которого минимальна.

На каждом шаге *алгоритма случайного спуска*, одного из первых стохастических алгоритмов оптимизации, вычисляется оценка стоимости для всех планов, достижимых из текущего за один шаг по графу планов (т. е. таких, которые могут быть получены применением одной трансформации). Если среди этих планов есть такие, оценка которых ниже оценки текущего плана, то один из них выбирается случайным образом и становится следующим текущим. Если же планов с лучшей оценкой среди соседних планов нет, то текущий является локальным минимумом. Лучший из найденных планов запоминается как приближенный оптимальный план, и поиск продолжается с другого начального плана.

Некоторое улучшение результатов алгоритма случайного спуска может достигаться применением *метода отжига*, который состоит в том, что на первых шагах спуска с некоторой вероятностью допускается переход к плану с худшей оценкой, причем вероятность такого перехода уменьшается по мере продвижения по графу. Если представить значения функции стоимости как поверхность, то метод отжига дает шансы выбраться из неглубоких «ям».

В системе PostgreSQL для приближенной оптимизации применяется *генетический алгоритм*, который также является стохастическим, но отличается тем, что в качестве трансформации плана используются так называемые мутации, которые представляют собой комбинации обычных трансформаций.

Эвристические приемы

Рассматриваемые в этом подразделе приемы не являются самостоятельными алгоритмами, а применяются в сочетании с другими алгоритмами для сокращения времени их работы.

Большинство эвристических приемов, не упомянутых ранее, сводится к разбиению запроса на части меньшего размера, каждая из которых оптимизируется отдельно. В качестве таких частей довольно часто используются подзапросы, явно выписанные в тексте запроса или полученные после подстановки текстов представлений.

Легко построить запросы, для которых такая эвристика приведет к очень плохим результатам. Примером может служить выборка из представления с указанием условий фильтрации в другой части основного запроса. В такой ситуации может помочь другая эвристика — проталкивание условий внутрь подзапроса, однако она не всегда применима.

В случае если оптимизация некоторого запроса выполняется с применением этих эвристик (это можно определить по плану выполнения запроса) и полученный план неудовлетворителен, может помочь ручное переписывание запроса в другой форме.

Оптимизации сверху вниз

Работа алгоритмов «сверху вниз» начинается с выбора завершающей операции плана, за которой следует (рекурсивно) построение оптимальных планов для подзапросов, представляющих аргументы этой операции. Для нахождения оптимального плана необходимо выполнить перебор всех операций, которые могут быть завершающими для этого запроса. Если для некоторого подзапроса удается получить оценку снизу для стоимости всех его планов, и эта оценка такова, что итоговая оценка плана всего запроса будет заведомо хуже, чем оценка лучшего из уже найденных планов, то весь подзапрос можно не рассматривать. Другими словами, возможно применение *метода ветвей и границ*.

В отличие от алгоритмов «снизу вверх» можно исключить построение планов для подзапросов, которые не могут быть частью основного запроса. Например, операция внешнего соединения некоммутативна, что существенно сокращает количество планов, которые необходимо просмотреть, однако алгоритм «снизу вверх» не может учесть такую информацию при генерации подпланов.

Основная трудность в реализации алгоритмов «сверху вниз» связана с построением оценки снизу для стоимости множества планов (т. е. оценки снизу для всех планов, соответствующих подзапросу).

Несмотря на наличие теоретических проработок (например, [30]), алгоритмы «сверху вниз» крайне редко применяются на практике.

12.4. Модели стоимости

12.4.1. Функции и модели стоимости

Работа всех алгоритмов оптимизации запросов предполагает применение функции стоимости для оценки качества плана. Качество работы оптимизатора поэтому очень существенно зависит от того, насколько точно функция стоимости отражает количество ресурсов, необходимых для выполнения запроса (с учетом выбранных целей). Очевидно, что можно абсолютно точно определить необходимое количество ресурсов, просто выполнив план запроса. Конечно, такое апостериорное вычисление функции стоимости бессмысленно для оптимизации. Поэтому оптимизаторы используют функции стоимости, основанные на моделях стоимости, позволяющих оценить необходимые ресурсы, не выполняя операцию или запрос. Основой для получения таких оценок являются статистические характеристики хранимых данных и промежуточных результатов.

Стоимость плана складывается из стоимостей выполнения отдельных операций, входящих в план, и стоимости передачи данных между операциями. Однако для запросов, выполняемых на одном сервере при использовании потоков данных между операциями (как описано выше), стоимость передачи данных оказывается пренебрежимо малой по сравнению со стоимостью операций.

Оценка стоимости отдельной операции зависит от алгоритма, выполняющего ее, а также от статистических свойств операндов (аргументов) этой операции, в первую очередь от количества строк и от суммарного объема данных (в байтах). Для некоторых алгоритмов выборки данных стоимость может зависеть не от объема хранимых данных, а от количества и объема результата операции.

Кроме оценки стоимости операции, модель стоимости обеспечивает получение оценок статистических свойств результата операции, которые необходимы для того, чтобы оценить стоимость (и результаты) последующих операций плана. Статистические свойства хранимых данных обычно представляются гистограммами, однако сколько-нибудь надежное предсказание параметров распределений для результатов большинства операций невозможно. Так, предсказать распределение значений атрибута соединения невозможно, даже если распределение этого атрибута — равномерное для обоих аргументов операции соединения. Поэтому все промышленные системы, в том числе и PostgreSQL, ограничиваются только оценками кардинальности (т. е. количества объектов) результата.

12.4.2. Модели стоимости для алгоритмов бинарных операций

В этом подразделе мы покажем, как можно вычислять простые оценки для алгоритмов выполнения бинарных операций на примере операций декартова (прямого) произведения и соединения.

Для отношения R будем использовать следующие характеристики:

T_R — количество кортежей (строк) в R;

B_R — размер памяти в блоках, необходимый для хранения отношения R, если бы оно было записано на диск;

b — размер блока;

l_R — средняя длина одного кортежа, равная $\frac{bB_R}{T_R}$.

Поскольку промежуточные результаты обычно передаются в следующую операцию как поток данных, фактическая запись на диск, как правило, не требуется, однако оценка размера данных все равно необходима, потому что стоимость операций зависит от размеров аргументов. Для оценки размера не имеет значения, присутствует ли в блоках свободное пространство и какая доля памяти занята вспомогательной информацией, определяющей структуру блоков (заголовками, таблицами переадресации и т. п.). Для оценок можно просто считать, что значение b учитывает только ту часть памяти, которая занята строками отношения.

Для оценки стоимости не имеет значения абсолютная величина единиц, в которых выражается оценка, важно только то, насколько быстро или медленно растет стоимость операции с ростом размеров ее аргументов. Для того чтобы описать, как именно зависит рост функции $f(x)$ от ее аргумента, ее сравнивают с другой (обычно более простой) функцией.

Нам понадобится обозначение O (читается «О большое»), используемое в математике для указания таких соотношений. Запись

$$f(x) = O(g(x))$$

обозначает, что существует некоторая константа C такая, что $f(x) < Cg(x)$ для любых значений x. Другими словами, f растет не быстрее, чем g.

Оценки для декартова произведения

Количество кортежей в произведении $R \times S$, очевидно, равно произведению количеств кортежей в аргументах:

$$T_{R \times S} = T_R \, T_S.$$

Длина кортежа произведения будет равна сумме длин кортежей аргументов, поэтому общий объем памяти, занимаемый произведением, оценивается как

$$B_{R \times S} = T_R \, T_S \, \frac{(l_R + l_S)}{b} = T_R \, B_S + T_S \, B_R.$$

Оценка размеров результата важна по двум причинам. Во-первых, результат надо вычислить, и на это понадобятся ресурсы. Во-вторых, результат операции может использоваться как аргумент в следующей операции, для оценки стоимости которой необходимы размеры аргументов.

Заметим, что сложность алгоритма вложенных циклов составляет $O(T_R \, T_S)$. Поскольку любой алгоритм, вычисляющий произведение, должен вычислить $T_R \, T_S$ кортежей, сложность любого такого алгоритма не может быть лучше, чем сложность алгоритма вложенных циклов (с точностью до постоянного множителя).

Оценки алгоритмов соединения

Пусть I обозначает количество различных значений атрибута (или группы атрибутов), по которым выполняется операция эквисоединения $R \bowtie S$. Предположим, что все значения этого атрибута встречаются в обоих операндах операции соединения и распределены равномерно (эти предположения практически никогда не выполняются в реальности, поэтому получаемые оценки не очень точны). Конечно, одно из отношений может содержать значения атрибутов, которые не встречаются в другом отношении; в этом случае для оценки размеров результата нужно исключить строки, содержащие такие значения, потому что они не влияют на результаты операции внутреннего соединения.

Если эти предположения справедливы, то количество кортежей, в которых атрибут принимает некоторое фиксированное значение, равно

$$\frac{T_R}{I} \quad \text{и} \quad \frac{T_S}{I}$$

для первого и второго аргумента соответственно.

Поскольку для этого фиксированного значения каждый кортеж из R образует пару с каждым кортежем из S, имеющим такое же значение атрибута, фактически будет вычислено прямое произведение, мощность которого составит

$$\frac{T_R}{I}\frac{T_S}{I} = \frac{T_R T_S}{I^2}$$

для каждого из I значений атрибута соединения. Поэтому общее количество кортежей в результате соединения оценивается формулой

$$T_{R \bowtie S} = \frac{T_R T_S}{I}.$$

Размер памяти, необходимый для хранения результата соединения, оценивается выражением

$$B_{R \bowtie S} = \frac{T_R B_S + T_S B_R}{I}.$$

Вычисление размеров результатов операции соединения проиллюстрировано на рис. 12.4.1. Значения атрибута соединения a5 встречаются только в первом аргументе, а значения a6 — только во втором, поэтому кортежи, содержащие эти значения, не дают вклад в результат операции.

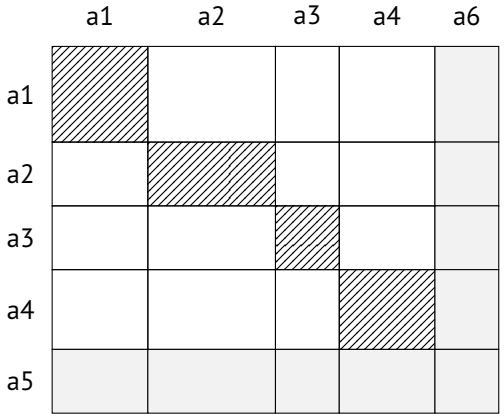


Рис. 12.4.1. Оценка размеров для операции соединения

Оценка стоимости для алгоритма вложенных циклов будет такой же, как для вычисления прямого произведения, т. е. $T_R T_S$, потому что при выполнении алгоритма вложенных циклов будут вычислены все кортежи произведения и для каждого из них будет проверено условие равенства атрибутов соединения.

Оценка алгоритма на основе сортировки и слияния складывается из:

- оценки размеров результата
$$\frac{T_R T_S}{I};$$
- оценки стоимости сортировки (если она необходима) для каждого из входных отношений
$$T_R \log_2(T_R) + T_S \log_2(T_S);$$
- оценки однократного просмотра отношений
$$T_R + T_S.$$

Наконец, оценка алгоритма на основе хеширования включает:

- стоимость однократного просмотра входных отношений
$$T_R + T_S;$$
- оценку алгоритма вложенных циклов для каждой из h корзин, которая составляет
$$\frac{T_R T_S}{h}.$$

Заметим, что $h \leqslant I$, поскольку все кортежи с одинаковыми значениями атрибута соединения должны попасть в одну корзину и выделение большего числа корзин не имеет смысла.

12.4.3. Оценки селективности

Приведенные выше простые модели стоимости для бинарных операций показывают, что их стоимость, а следовательно, и стоимость всего плана, существенно зависят от размеров и количества данных, обрабатываемых этими операциями. Вычисление оценок начинается с оценки размеров и количества данных, выбираемых из хранимых таблиц.

Количество строк, передаваемых для обработки последующим операциям, определяется тем, какие операции фильтрации выполнены при выборке хранимых данных (а размер выбираемых строк зависит от операций проекции). Отношение количества выбранных строк к общему количеству входных строк фильтра называется его *селективностью*.

Очевидно, что селективность фильтра зависит от предиката, по которому выполняется фильтрация. Для предикатов вида $A = c$, содержащих отношение равенства некоторого атрибута константе, простую оценку селективности можно получить, предполагая, что значения атрибута распределены равномерно. Пусть I обозначает количество различных значений атрибута. Тогда для фильтра, выбирающего одно значение этого атрибута, оценки селективности и количества строк, полученных после фильтрации, будут равны соответственно

$$\frac{1}{I} \text{ и } \frac{T_R}{I}.$$

Оценку для предикатов, содержащих отношение неравенства (больше, меньше и т. п.) можно получить, зная минимальное и максимальное значение атрибута, по которому выполняется фильтрация, используя линейную интерполяцию. Пусть атрибут A принимает числовые значения, его максимальное и минимальное значения равны a_{max} и a_{min}, а условие фильтрации имеет вид $A > c$. В предположении о равномерности распределения значений оценка селективности такого фильтра будет равна

$$\frac{a_{max} - c}{a_{max} - a_{min}},$$

и, как и ранее, для получения оценки количества строк на выходе фильтра нужно умножить селективность на T_R.

12.4.4. Статистические характеристики данных

Для того чтобы оптимизатор мог вычислять оценки стоимости для операций и планов, система управления базами данных собирает и хранит в системных таблицах статистические характеристики. Как минимум для вычисления оценок необходимы размеры всех хранимых объектов данных (таблиц, индексов, материализованных представлений и т. д.) и количество элементов (строк или индексных записей) в этих объектах.

Эти характеристики можно применять, предполагая равномерность распределения значений, однако это предположение почти никогда не выполняется для реальных данных. Поэтому все высокопроизводительные системы поддерживают дополнительную информацию о статистических свойствах данных. Эта информация должна быть представлена компактно (т. е. иметь относительно небольшой объем), для того чтобы оптимизатор мог ее использовать, не затрачивая на это слишком много ресурсов.

В системе PostgreSQL такая дополнительная информация представляется гистограммами и списками наиболее частых значений атрибутов.

Гистограммы строятся для отдельных атрибутов и представляют собой небольшие таблицы, строки которых соответствуют интервалам значений атрибута, встречающихся в таблице. Известно очень много разных типов гистограмм, однако в СУБД применяются только некоторые из них.

Для каждого интервала указывается количество строк таблицы, значение атрибута которых попадает в этот интервал. Полезность гистограмм основана на том, что распределение значений в пределах одного интервала, как правило, значительно ближе к равномерному, чем распределение значений атрибута на всем диапазоне. При наличии гистограмм оценки селективности и кардинальности результатов операций можно вычислять отдельно на каждом из интервалов и затем обобщать результаты.

Для оценки кардинальности результата фильтрации по условию на равенство атрибута константе достаточно разделить общее количество строк в интервале, содержащем эту константу, на количество различных значений атрибута в том же интервале. Для условий, содержащих отношения неравенства, используется линейная интерполяция в интервале, содержащем константу, плюс суммарное количество строк во всех предшествующих (или следующих) интервалах.

Если для атрибута разрешены неопределенные значения, их количество влияет на селективность любых фильтров, содержащих условия на этот атрибут. Это обстоятельство делает оценки селективности немного более громоздкими: строки, содержащие неопределенные значения, не могут удовлетворять условиям таких фильтров, поэтому при оценках селективности количество неопределенных значений необходимо вычитать из общего количества строк в таблице. Заметим, что подсчет количества неопределенных значений нужен также для оценки селективности фильтров, проверяющих неопределенное значение.

Одним из широко распространенных отклонений распределений значений от равномерного является наличие очень часто встречающихся значений. Наличие таких значений существенно ухудшает качество получаемых оценок селективности даже при использовании гистограмм. Кроме этого, если в условии фильтрации используются такие значения, то, скорее всего, оптимальным будет другой алгоритм выборки данных из хранимой таблицы.

Чтобы избежать этих негативных явлений, для неуникальных атрибутов строятся *списки наиболее часто встречающихся значений*, в которых для каждого из значений указывается частота строк таблицы, содержащих это значение. Для

фильтров, не содержащих часто встречающиеся значения, эти строки должны быть исключены из подсчетов при оценках селективности, а для условий, содержащих такие значения, селективность просто выбирается из списка.

Сбор статистики требует просмотра значительного количества хранимых данных и является поэтому достаточно ресурсоемкой операцией. Для очень больших таблиц может иметь смысл вычисление статистики на основе случайной выборки (sample) из таблицы. Такая статистика, конечно, не может быть абсолютно точной, однако ее точность обычно бывает достаточна для получения надежных оценок селективности (и кардинальности) результата, а время вычисления существенно меньше, чем время полной обработки всей таблицы. Размер случайной выборки, которая считается достаточно представительной, в системе PostgreSQL определяется параметром сервера. В других системах размер выборки может зависеть от размеров таблицы.

Постоянная поддержка статистической информации в актуальном состоянии создавала бы слишком большую нагрузку на СУБД и поэтому вряд ли целесообразна. Статистические свойства данных обычно изменяются незначительно даже при частых изменениях отдельных строк таблиц, в особенности — больших. Поэтому процедура обновления статистики запускается только когда количество изменений в таблице превысит некоторое пороговое значение.

В системе PostgreSQL обновление статистики выполняется одновременно с реструктуризацией таблиц процедурой очистки от устаревших записей (vacuum). После выполнения этой процедуры статистика приводится в соответствие с актуальной репрезентативной выборкой (а для небольших таблиц становится точной). Имеется возможность проверки (по системным таблицам), насколько актуальна статистическая информация для любого хранимого объекта.

Обновить статистику можно также с помощью оператора ANALYZE. При отсутствии параметров этот оператор SQL заново вычисляет статистическую информацию для всех таблиц (а также материализованных представлений) базы данных, а при указании таблицы — только для этой таблицы. Можно, кроме этого, указать, статистика каких колонок требует перевычисления.

12.5. Другие подходы к оптимизации запросов

В этом разделе кратко характеризуются перспективные подходы к оптимизации и выполнению запросов, которые, как правило, пока не используются или используются в ограниченной форме в промышленных СУБД.

12.5.1. Адаптивное выполнение запросов

В большинстве случаев оптимизатор хорошо справляется со своей работой, однако бывают ситуации, когда построенный оптимизатором план оказывается далеким от оптимального. Как правило, причиной этого является неточность оценок кардинальности промежуточных результатов выполнения запроса, получаемых на фазе оптимизации на основе моделей стоимости. Во многих случаях получение точных оценок невозможно даже при использовании детальной статистической информации. Однако очевидно, что после выполнения операции необходимые оценки могут быть получены очень точно.

Идея адаптивной оптимизации состоит в том, что после выполнения некоторой части запроса, когда уже становятся известны значения статистических свойств результатов, можно выполнить повторную оптимизацию и продолжить выполнение запроса по новому плану. Такое уточнение или изменение плана может выполняться многократно, до тех пор пока запрос не будет выполнен полностью, т. е. фазы оптимизации и выполнения могут чередоваться.

Реализация этой идеи, однако, оказывается довольно сложной. Подробный анализ известных на момент написания методов адаптивной оптимизации содержится в обзоре, представленном на конференции VLDB [23], и в расширенном виде в журнале [22].

Возможность адаптивной оптимизации зависит от свойств операций, входящих в план запроса. Говорят, что операция *не имеет состояния*, если обработка очередного входного кортежа не зависит от результатов обработки предыдущих кортежей. Примерами операций без состояния являются фильтры и проекции. Операция называется *блокирующей*, если вывод ее результатов начинается после полного считывания одного или нескольких входных аргументов. Так, операция сортировки является блокирующей. Последнее свойство может зависеть не только от логической операции, но и от алгоритма. Например, алгоритм вложенных циклов для операции соединения является неблокирующим, алгоритм соединения на основе хеширования — блокирующим (потому что результат может появиться только после полного считывания первого аргумента), а алгоритм симметричного хеширования — неблокирующим.

Для запросов, содержащих исключительно операции без состояния, может применяться **схема динамической маршрутизации**. Рассмотрим эту схему на примере запроса, содержащего конъюнкцию нескольких критериев фильтрации для одного отношения. При вычислении такого запроса критерии применяются последовательно. Первым следует применять тот критерий фильтрации,

который оставляет для последующей обработки наименьшую долю строк, минимизируя объем работы для других фильтров.

Оценки селективности, получаемые на основе статистики, неточны, поэтому может потребоваться изменение порядка применения критериев фильтрации после уточнения оценок селективности. Схема алгоритма маршрутизации кортежей представлена на рис. 12.5.1.

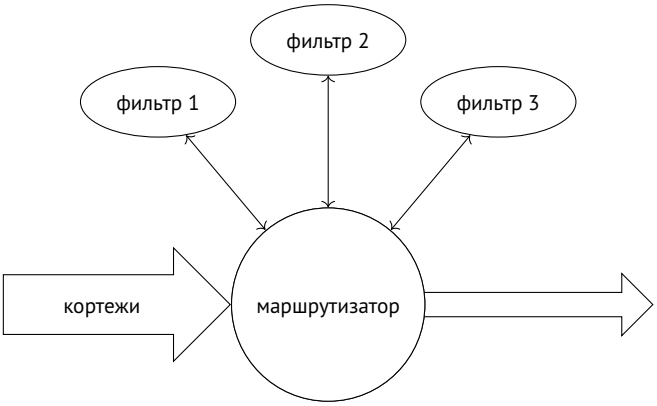


Рис. 12.5.1. Адаптивная маршрутизация кортежей

Маршрутизатор направляет входящие кортежи последовательно на все фильтры (если, конечно, кортеж возвращается после фильтра). Последовательность применения фильтров определяется статистическими характеристиками, которые пересчитываются по мере обработки кортежей и, если необходимо, учитываются при определении порядка применения фильтров для последующих входных кортежей.

Пересмотр порядка применения фильтров нетривиален и требует применения алгоритмов оптимизации (например, динамического программирования или жадного, подобного упомянутому выше), поэтому изменение порядка выполняется не после каждого кортежа, а после обработки некоторого количества, которое могло изменить оценки селективности отдельных фильтров.

Для операций с состояниями схема маршрутизации неприменима. Другой метод адаптивной оптимизации, представленный на рис. 12.5.2, состоит в использовании **точек материализации**.

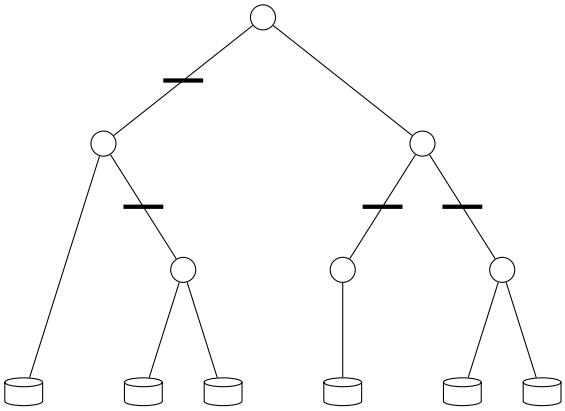


Рис. 12.5.2. Создание точек материализации

Точки материализации, условно показанные на рисунке жирными горизонтальными отрезками, прерывают потоковую передачу кортежей между операциями и записывают промежуточный результат (на диск или в память). При этом вычисляются уточненные оценки кардинальности (этого и последующих) промежуточных результатов, на их основе выполняется оптимизация оставшейся части плана, и затем выполнение продолжается с точек материализации, как с хранимых таблиц.

Использование точек материализации можно сочетать с маршрутизацией кортежей: в плане выполнения запроса выделяются сегменты, не содержащие операций с состоянием, для которых может применяться маршрутизация. При этом точки материализации могут устанавливаться после (некоторых из) операций с состоянием.

В литературе можно найти и другие варианты адаптивного подхода к выполнению запросов, предполагающие частичную потерю промежуточных результатов. Детальное описание таких методов и весьма обширная библиография имеется в обзоре [22].

Все варианты методов адаптивной оптимизации приводят к некоторым дополнительным расходам ресурсов, поэтому применение адаптивной оптимизации всегда оказывается дороже, чем выполнение оптимального плана. Однако адаптивная оптимизация может предотвратить использование очень плохих планов, которые могут быть выработаны обычным оптимизатором на основе неточных оценок кардинальности.

12.5.2. Параметрическая оптимизация

Выбор алгоритма для выборки хранимых данных определяется селективностью критериев фильтрации: если количество выбираемых строк мало, скорее всего будут использованы индексы, иначе более низкую стоимость, вероятно, имеет полный просмотр. От селективности, конечно, зависит и кардинальность промежуточных результатов, передаваемых в последующие операции, и, следовательно, выбор алгоритмов и порядок выполнения всех остальных операций плана.

При неравномерном распределении значений атрибута может оказаться, что селективность фильтрации зависит от фактического значения константы, которая может быть неизвестна во время оптимизации (в случае выполнения оптимизации при подготовке запроса). Если при выполнении подготовленного запроса в качестве критерия передается часто встречающееся значение, то план, подготовленный для редко встречающихся значений, не будет оптимальным, и наоборот, план, подготовленный для часто встречающихся значений, не будет оптимальным, если передается редкое значение.

Идея параметрической оптимизации состоит в том, чтобы во время оптимизации найти несколько планов, которые будут оптимальными для разных диапазонов значений селективности. Перед выполнением запроса, когда фактическое значение критерия фильтрации становится известным, оценивается его селективность и выбирается наиболее подходящий из ранее найденных оптимальных планов.

При наличии нескольких параметров пространство всех значений параметров в процессе оптимизации разбивается на регионы, для каждого из которых выбирается оптимальный план. Количество таких регионов и сложность их формы очень быстро растет с ростом числа параметров, поэтому необходимы методы приближенного описания регионов. Однако даже при искусственном сокращении количества регионов параметрическая оптимизация оказывается слишком сложной для широкого применения на практике.

Заметим, что в системе PostgreSQL проблема зависимости оптимального плана от фактических значений параметров, как правило, не стоит, потому что обычно этап оптимизации выполняется, после того как становятся известны параметры запроса. Однако при подготовке запроса планировщик может переключиться на использование плана, построенного без учета фактических значений параметров. В этом случае параметрическая оптимизация была бы полезна, но она не применяется.

12.5.3. Семантическая оптимизация

Методы оптимизации на основе стоимости, рассмотренные выше, неявно предполагают, что запрос не содержит избыточных операций. На практике, однако, далеко не все запросы написаны оптимальным образом. В частности, избыточность обычно появляется в генерируемых запросах, а также в запросах, использующих представления. Зачастую запросы могут быть переписаны так, чтобы устранить избыточные действия. Энтузиасты этого подхода обычно подчеркивают, что, если в запросе имеется тождественно ложное условие или подзапрос, всегда вырабатывающий пустой результат, то такие подзапросы или даже запрос в целом можно не выполнять.

Некоторые из условий фильтрации, включенные в запрос, могут оказаться несовместимыми с ограничениями целостности, заданными для таблицы. Поэтому при выполнении семантической оптимизации следует такие ограничения учитывать так же, как и явно указанные в тексте запроса. К сожалению, в общем случае задача проверки того, что некоторый запрос возвращает пустой результат для любого состояния базы данных является алгоритмически неразрешимой. Эта задача может быть решена за полиномиальное время только для ограниченного класса запросов к одной таблице, содержащих условия равенства или линейные неравенства для отдельных атрибутов. По этой причине все попытки реализации идеи семантической оптимизации ограничиваются задачей упрощения системы линейных условий на атрибуты одного отношения.

Семантические преобразования запросов (из этого ограниченного класса) могут выполняться и планировщиком системы PostgreSQL.

12.5.4. Многокритериальная оптимизация

Идея рассматривать различные составляющие функции стоимости (процессорное время, ввод-вывод и т. п.) как независимые функции немедленно приводит к задаче многокритериальной оптимизации, т. е. к задаче, в которой целевая функция является вектором.

Поскольку в векторных пространствах не существует естественного упорядочивания, в качестве решения многокритериальной задачи рассматривается множество Парето: такое множество решений (планов выполнения для задачи оптимизации запросов), для любого элемента которого невозможно найти другой план, который превосходил бы рассматриваемый по всем критериям. Иначе говоря, для планов, входящих в множество Парето, улучшение значения

одной из функций стоимости возможно только при ухудшении значений (по крайней мере, одной из) других.

После нахождения множества Парето выбирается план с учетом относительной важности функций стоимости.

Исследовательские реализации методов многокритериальной оптимизации запросов показывают, что вычислительная сложность оказывается слишком большой для практического применения этого подхода.

12.6. Итоги главы

В этой главе рассмотрены методы выполнения запросов, которые применяются в большинстве высокопроизводительных СУБД, и описаны компоненты оптимизатора запросов и их основные функции. Представлены основные алгоритмы оптимизации, функции стоимости и модели, используемые для оценки стоимости отдельных операций.

Кратко обсуждаются подходы к оптимизации, описанные в исследовательской литературе, но пока не применяемые в промышленных СУБД.

12.7. Упражнения

Упражнение 12.1. В демонстрационной базе данных постройте индексы для поиска бронирований по имени пассажира и интервалу дат вылета.

Упражнение 12.2. В демонстрационной базе данных постройте индексы для поиска вариантов перелета по названию пунктов отправления и прибытия и дате вылета.

Упражнение 12.3. В контексте предыдущих упражнений отключите использование индексов и сравните планы выполнения запросов с индексами и без них.

Упражнение 12.4. Найдите в демонстрационной базе данных статистическую информацию.

Упражнение 12.5. Установите параметры оптимизатора таким образом, чтобы изменение порядка операций соединения никогда не производилось. Получите разные планы выполнения запросов, содержащих 3, 5, 8 и 13 операций соединения, изменяя порядок соединений вручную. Изучите и сравните оценки стоимости для различных порядков операций соединения каждого запроса.

Упражнение 12.6. В контексте предыдущего упражнения получите фактические размеры промежуточных результатов и сравните их с оценками, используемыми оптимизатором.

Глава 13

Управление транзакциями

В этой главе обсуждаются теоретические модели, описывающие управление транзакциями, протоколы, используемые в диспетчерах, и особенности реализации поддержки транзакций в системе PostgreSQL.

Напомним, что транзакцией называется конечный набор операций над базой данных, который переводит одно согласованное состояние базы данных в другое при условии, что все операции выполнены полностью и без помех со стороны других транзакций. Неформально требования к транзакционным системам характеризуются свойствами ACID (атомарность, согласованность, изоляция и долговечность), описанными в главе 6.

Отвечая за согласованность базы данных, средства управления транзакциями решают две задачи:

- поддержка базы данных в согласованном состоянии при нормальной работе системы;
- восстановление согласованного состояния при отказах (транзакций, системы или носителя данных).

При этом предполагается, что атомарность отдельных операций, входящих в транзакции, обеспечивается на другом уровне. Фактически же даже самые простые операции, рассматриваемые в теории транзакций, реализуются в многоуровневой системе, включающей как программные, так и аппаратные компоненты. Поэтому предположение об атомарности этих операций может оказаться далеким от реальности. Тем не менее проблемы, которые удается теоретически решить на основе этого предположения, потенциально встречаются на много порядков чаще, чем проблемы, связанные с фактической неатомарностью базовых операций. Далее в этой главе считается, что атомарность операций имеет место.

13.1. Критерии корректности конкурентного выполнения

Этот раздел содержит краткое изложение теории конкурентного выполнения транзакций. Идеальная картина, представленная здесь, существенно отличается от регламентированной стандартом SQL и отличается от реализаций в СУБД, в том числе и в PostgreSQL. С точки зрения теории самое важное отличие состоит в том, что теория основана на анализе значений, записываемых в базу данных и вырабатываемых в результате выполнения операций, а требования стандарта SQL формулируются в терминах запрета на различные аномалии, которые потенциально возможны при конкурентном выполнении. Мы рассмотрим это более детально при обсуждении ослабленных критериев корректности.

13.1.1. Формальные модели корректности

Классическая теория транзакций использует очень простую модель: считается, что база данных состоит из независимых и никак между собой не связанных элементов данных, которые обозначаются *x, y,* … Над этими элементами можно выполнять операции чтения *r* и записи *w*, при этом каждая операция выполняется в рамках какой-либо транзакции, идентификатор которой указывается индексом операции. Так, операция $r_1(x)$ относится к транзакции 1 и читает *x,* а операция $w_i(y)$ записывает *y* в транзакции *i.*

Нам понадобится несколько бинарных отношений. Математическое понятие отношения обсуждалось в главе 2. Отношение называется бинарным, если оно имеет два атрибута. В нашем случае эти атрибуты будут принимать значения из одного домена. Для таких отношений знак, обозначающий отношение, обычно записывается между атрибутами.

Простой пример такого отношения — отношение равенства $a = b$. Несколько более широкий класс отношений — отношения *эквивалентности*. Неформально можно сказать, что отношение эквивалентности разбивает множество объектов на группы, внутри каждой из которых объекты обладают совпадающими свойствами, а в разных группах эти свойства различаются. Такие группы часто называют классами эквивалентности.

Отношение *R* называется *симметричным*, если из $a\ R\ b$ следует, что верно и $b\ R\ a$ (т. е. отношение сохраняется при перестановке атрибутов), и называется *антисимметричным*, если из $a\ R\ b$ следует, что $b\ R\ a$ не выполняется.

Отношение R называется *транзитивным*, если из $a\ R\ b$ и $b\ R\ c$ следует, что $a\ R\ c$. Например, отношение равенства является транзитивным и симметричным.

Отношение называется *отношением порядка* или *упорядочением*, если оно антисимметрично и транзитивно. Такие отношения часто обозначаются каким-либо знаком, похожим на знак «меньше», например $<$ или $\prec$. Запись $a < b$ можно читать как «a предшествует b».

Упорядоченность называется *полной*, если для любой пары несовпадающих объектов либо $a < b$, либо $b < a$. Если могут быть пары, для которых не верно ни то ни другое, то упорядочение называется *частичным*. Например, упорядочение вещественных чисел является полным, а отношение вложенности подмножеств $A \subset B$ является частичным упорядочением.

Предполагается, что операции транзакции частично упорядочены отношением предшествования, а операции над каждым элементом данных упорядочены полностью. Отказ от полной упорядоченности всех операций дает возможность применять теорию в случае, когда по логике транзакции допустимо и технически возможно параллельное (или псевдопараллельное) выполнение.

В примерах, показанных на рис. 13.1.1, операции чтения $r_1(x)$, $r_1(y)$ и операции записи $w_1(x)$, $w_1(y)$ не связаны отношением упорядоченности и поэтому могут выполняться параллельно; во второй транзакции операции $r_2(x)$, $r_2(y)$ также могут выполняться параллельно. Однако все операции чтения должны предшествовать всем операциям записи в этих транзакциях.

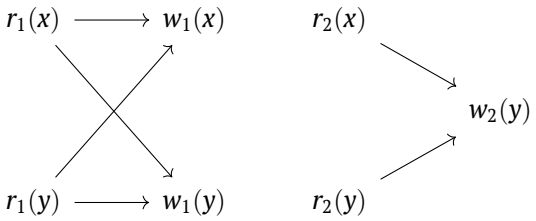


Рис. 13.1.1. Частичное упорядочение операций в транзакциях

Можно записать эти транзакции как полностью упорядоченные (несколькими разными способами), например для первой транзакции:

$$r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$$
$$r_1(x)\ r_1(y)\ w_1(y)\ w_1(x)$$

и для второй:

$$r_2(x)\ r_2(y)\ w_2(y)$$
$$r_2(y)\ r_2(x)\ w_2(y).$$

В такой записи предполагается, что операция, расположенная слева, предшествует любой операции, расположенной справа от нее, а стрелки, определяющие отношение предшествования, опускаются.

Историей для конечного набора транзакций называется множество всех операций этих транзакций, снабженное отношением частичного порядка, согласованным с отношением порядка операций внутри транзакций и таким, что все операции над одним элементом данных полностью упорядочены. В историю включаются также операции завершения транзакций фиксации *c* (commit) или обрыва *a* (abort), которые следуют (в смысле частичной упорядоченности) за всеми операциями транзакции, которую они завершают. При этом каждая транзакция завершается ровно одной из операций *c* или *a*.

Понятие истории формализует описание того, в каком порядке выполнялись операции нескольких транзакций. Операции любого набора, состоящего из более чем одной транзакции, можно организовать в истории несколькими различными способами. При этом некоторые истории, возможно, будут некорректными.

Пример частично упорядоченной истории для транзакций, показанных выше на рис. 13.1.1, приведен на рис. 13.1.2.

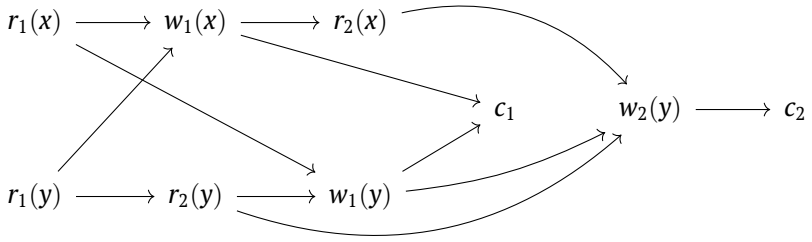


Рис. 13.1.2. Частично упорядоченная история

Полностью упорядоченная история для тех же транзакций может быть, например, такой:

$$r_1(x)\ r_1(y)\ w_1(x)\ r_2(x)\ r_2(y)\ w_1(y)\ c_1\ w_2(y)\ c_2.$$

Этот пример дополняет частично упорядоченную историю, приведенную выше, новыми упорядоченностями. Например, в частично упорядоченной истории не определен порядок, в котором выполняются операции фиксации c_1 и c_2, а в полностью упорядоченной истории $c_1 < c_2$.

Заметим, что как частично, так и полностью упорядоченная истории некорректны, поскольку в них присутствует аномалия потерянного обновления элемента данных y.

Префиксом частично упорядоченного множества называется подмножество, которое вместе с каждым элементом содержит все элементы, предшествующие ему в смысле частичного порядка.

Префикс истории называется *расписанием*. Пример префикса для истории, показанной на рис. 13.1.2, представлен на рис. 13.1.3.

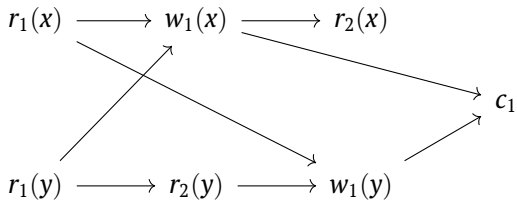


Рис. 13.1.3. Частичное упорядоченное расписание

Упорядоченное расписание может быть, например, таким:

$$r_1(x)\ r_1(y)\ w_1(x)\ r_2(x)\ r_2(y)\ w_1(y)\ c_1.$$

Заметим, что оба приведенных расписания корректны, т. к. в них операция $w_2(y)$ не включена.

Все утверждения в этом разделе, кроме тех, для которых оговорено иначе, справедливы для частично упорядоченных транзакций и расписаний. Однако, поскольку корректность выполнения определяется в зависимости от предшествования операций, все примеры расписаний будут полностью упорядочены.

Основная цель этого раздела — определить, какие расписания являются корректными. Теория, обсуждаемая в этом разделе, строится в предположении, что каждая транзакция читает и записывает любой элемент данных не более одного раза, при этом если элемент записывается, то запись следует за чтением. Кроме этого, предполагается, что все операции оборванных транзакций

уже исключены из расписания. Из этих предположений следует, что в рамках обсуждаемой теории невозможно рассмотреть некоторые виды аномалий, например аномалию неповторяющегося чтения, поскольку каждая операция чтения выполняется только один раз.

Расписание называется *серийным*, если для любой пары транзакций в этом расписании все операции одной транзакции предшествуют всем операциям другой. Отношение предшествования для операций остается отношением частичного порядка, но транзакции в таком расписании полностью упорядочены. По определению транзакции, серийное расписание является корректным, т. к. каждая транзакция в нем выполняется полностью и без помех со стороны других транзакций.

Неформально, расписание будет корректно, если оно дает такие же результаты, как какое-нибудь серийное. Осталось совсем немного: научиться сравнивать результаты и подобрать подходящий критерий эквивалентности. Конечно, эквивалентными могут быть только расписания, содержащие одинаковые наборы транзакций. Далее в этом разделе при сравнении расписаний всегда предполагается, что множества транзакций и множества операций в этих расписаниях совпадают.

Вычисление результатов выполнения расписаний основано на предположении о том, что любая операция чтения возвращает результат, записанный последней предшествующей операцией записи того же элемента данных. В наших предположениях такая предшествующая операция записи всегда принадлежит другой транзакции.

Для того чтобы такое определение можно было использовать для всех операций чтения, вводится начальная транзакция t_0, которая записывает начальное состояние всех элементов базы данных и все операции которой предшествуют всем операциям других транзакций. Аналогично можно ввести транзакцию t_∞, которая считывает все элементы базы данных и все операции которой следуют за всеми операциями других транзакций в истории.

Значение, которое использует операция записи, определяется кодом приложения, выполняющего транзакцию, и зависит от элементов базы данных, которые транзакция прочитала до выполнения операции записи. Это значение может зависеть и от входных параметров приложения (например, введенных пользователем с клавиатуры). Такие параметры можно также считать частью кода приложения.

Эти рассуждения формально описываются функцией h, которая называется *семантикой Эрбрана* и определяется (рекурсивно) следующим образом:

- $h(r_i(x)) = h(w_j(x))$, где $w_j(x)$ — последняя операция записи, предшествующая $r_i(x)$;
- $h(w_i(x)) = f_{i,x}(h(r_i(y^1)), h(r_i(y^2)), \ldots)$, где функция $f_{i,x}$ выражает семантику приложения, а ее аргументы — операции чтения, предшествующие $w_i(x)$ в той же транзакции. Верхние индексы обозначают, конечно, не показатель степени, а порядковый номер аргумента функции f.

Используя функцию h, можно записать выражения, определяющие считываемые или записываемые значения для любой операции в расписании. Например, для расписания

$$r_1(x)\ r_2(x)\ r_1(y)\ w_1(x)\ w_2(x)\ w_1(y) \tag{13.1}$$

семантики Эрбрана имеют вид:

$$\begin{aligned} h(r_1(x)) &= f_{0x}, \\ h(r_2(x)) &= f_{0x}, \\ h(r_1(y)) &= f_{0y}, \\ h(w_1(x)) &= f_{1x}(f_{0x}, f_{0y}), \\ h(w_2(x)) &= f_{2x}(f_{0x}), \\ h(w_1(y)) &= f_{1y}(f_{0x}, f_{0y}). \end{aligned}$$

Заметим, что рассматриваемое расписание 13.1 некорректно, т. к. в нем присутствует аномалия потерянного обновления: операция $w_2(x)$ записывает значение, не учитывая результат первой транзакции, записанный операцией $w_1(x)$, который в этом расписании теряется.

Корректное расписание выполнения тех же транзакций может иметь вид

$$r_1(x)\ r_1(y)\ w_1(x)\ r_2(x)\ w_2(x)\ w_1(y).$$

Семантики для операций второй транзакции этого расписания будут такими:

$$\begin{aligned} h(r_2(x)) &= f_{1x}(f_{0x}, f_{0y}), \\ h(w_2(x)) &= f_{2x}(f_{1x}(f_{0x}, f_{0y})). \end{aligned}$$

Истории называются *эквивалентными по конечному состоянию*, если построенные по ним семантики Эрбрана совпадают для t_∞, и *эквивалентными по видимому состоянию*, если совпадают семантики Эрбрана для всех операций.

Расписание называется *сериализуемым по конечному состоянию* (FSR) или *сериализуемым по видимому состоянию* (VSR), если оно эквивалентно по конечному состоянию или видимому состоянию соответственно какому-нибудь серийному расписанию.

Сериализуемость по конечному состоянию является слишком слабой (например, не предотвращает аномалию несогласованного чтения) и поэтому используется только теоретиками для сравнения с другими классами расписаний. Очевидно, что VSR $\subset$ FSR, т. е. любое расписание, сериализуемое по видимому состоянию, будет сериализуемо по конечному.

Для доказательства того, что включение — строгое, достаточно предъявить хотя бы одно расписание, входящее в FSR, но не VSR. Например, расписание

$$r_1(x)\ r_1(y)\ r_2(x)\ w_1(x)\ w_1(y)\ r_2(y)$$

принадлежит FSR, потому что изменения вносятся только одной транзакцией, но не принадлежит VSR. Для формализации доказательства следует выписать семантики Эрбрана для этого расписания и для обоих серийных расписаний транзакций t_1 и t_2.

Сериализуемость по видимому состоянию, напротив, дает идеальные результаты с точки зрения семантической корректности: результаты всех операций неотличимы от результатов при некотором серийном выполнении и, следовательно, корректны. К сожалению, проверка сериализуемости по видимому состоянию является NP-полной задачей. В этом курсе мы не будем обсуждать точное значение этого термина. Неформально это значит, что время выполнения любого алгоритма, решающего эту задачу, будет очень быстро расти с увеличением размеров задачи.

В нашем случае размер задачи определяется числом операций в расписании, поэтому критерий корректности, основанный на VSR, неприменим на практике. Тем не менее сериализуемость по видимому состоянию важна для доказательства корректности других, более узких классов расписаний: если можно доказать, что расписания из некоторого класса сериализуемы по видимому состоянию, то они будут семантически корректными.

Широко известные эффективно проверяемые критерии корректности основаны на понятии *конфликта*.

Конфликтом называется пара операций $p_i(x) \to q_j(x)$, такая, что:

1) операции p и q применяются к одному и тому же элементу данных x;

2) транзакции i и j различны;

3) по крайней мере одна из операций p и q является операцией записи w.

Заметим, что операции в конфликте всегда упорядочены, поскольку операции над одним элементом данных упорядочены в любом расписании. Для определенности мы будем считать, что первая операция конфликта всегда предшествует второй. Важно подчеркнуть, что определение конфликта не требует, чтобы пересекались интервалы времени, в которые выполняются транзакции, содержащие конфликтующие операции, и не имеет значения, выполнялись ли какие-либо операции над этим элементом данных между конфликтующими операциями. Конфликт не является ни препятствием для выполнения транзакций, ни признаком некорректности расписания.

Расписания называются *эквивалентными по конфликтам*, если совпадают множества конфликтов, имеющихся в этих расписаниях. Расписание называется *сериализуемым по конфликтам* (CSR), если оно эквивалентно по конфликтам некоторому серийному.

Пусть расписания S_1 и S_2 одного и того же множества транзакций неэквивалентны по видимому состоянию. Тогда найдется операция $r_i(x)$, такая, что в S_1 последней предшествующей операцией записи будет $w_j(x)$, а в S_2 — операция $w_k(x)$. В расписании S_1 операция $w_k(x)$ будет либо предшествовать $w_j(x)$, либо следовать за $r_i(x)$. Аналогичные, но другие соотношения будут иметь место в S_2. Из этого следует, что операции будут располагаться в конфликтах в разном порядке, следовательно S_1 и S_2 не эквивалентны по конфликтам и, следовательно, CSR $\subset$ VSR, т. е. эквивалентность по конфликтам обеспечивает семантическую корректность.

Графом сериализуемости для расписания называется ориентированный мультиграф (граф, в котором любая пара вершин может быть соединена несколькими дугами), вершины которого соответствуют транзакциям, входящим в расписание, а дуги — конфликтам между операциями этих транзакций. Если для некоторой пары транзакций существует несколько конфликтов, то граф будет содержать несколько дуг, соединяющих соответствующие вершины. На рис. 13.1.4 показан граф сериализуемости для расписания 13.1.

Существует эффективно проверяемый критерий сериализуемости расписаний по конфликтам: расписание сериализуемо по конфликтам тогда и только тогда,

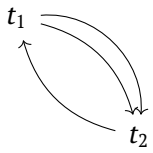


Рис. 13.1.4. Пример графа сериализуемости

когда граф сериализуемости не содержит контуров (т. е. обладает свойством ацикличности).

Набросок доказательства. Если граф не содержит контуров, эквивалентное серийное расписание строится с помощью процедуры топологической сортировки. Обратно, для любого пути в графе сериализуемости последовательность вершин вдоль пути должна быть совместима с упорядоченностью соответствующих транзакций в эквивалентном серийном расписании. Поэтому при наличии контура построение серийного расписания невозможно.

Важно отметить, что предшествование операций и порядок, в котором сериализуются транзакции, не обязательно связаны с упорядочением операций по времени, даже если расписание полностью упорядочено.

Рассмотрим пример:

$$r_2(x)\ r_1(x)\ w_1(x)\ r_3(y)\ w_3(y)\ r_2(y). \tag{13.2}$$

Граф сериализуемости для расписания 13.2 имеет вид

$$t_3 \to t_2 \to t_1.$$

Этот граф не содержит контуров и поэтому расписание сериализуемо, однако порядок, в котором транзакции t_1 и t_3 сериализуются, противоположен тому порядку, в котором выполнялись их операции. Таким образом, в эквивалентном серийном расписании порядок, в котором выполняются эти транзакции, отличается от порядка в исходном.

В некоторых случаях подобные изменения порядка транзакций нежелательны. Подкласс класса CSR, в котором порядок транзакций, упорядоченных в исходном расписании, сохраняется при сериализации, обозначается OCSR. Этот подкласс важен для таких методов управления транзакциями, в которых части транзакций могут сериализоваться независимо. Например, такие методы могут использоваться в распределенных системах.

Понятие конфликта можно заменить на понятие *коммутативности операций*. Две операции коммутируют, если выполняется хотя бы одно из следующих условий:

- операции не упорядочены в расписании (и, следовательно, могут выполняться в любом порядке);
- операции являются операциями чтения;
- операции выполняются над разными элементами данных разными транзакциями;
- операции выполняются над разными элементами данных в одной транзакции, но их порядок не определен в этой транзакции.

Очевидно, что операции, находящиеся в конфликте, не коммутируют.

Если операции коммутируют, то их можно поменять местами в расписании, а если они не упорядочены, то можно упорядочить в любом порядке. Если в результате применения таких трансформаций расписание может быть преобразовано в серийное, оно называется *сериализуемым по коммутативности*.

Легко доказать, что сериализуемость по коммутативности эквивалентна сериализуемости по конфликтам, однако понятие коммутативности легче обобщить на другие виды операций, кроме чтения и записи. Это дает возможность использовать теорию в более широких контекстах.

13.1.2. Изоляция мгновенных снимков

Несмотря на то что проверка сериализуемости по конфликтам может выполняться эффективно, оказалось, что широко известные протоколы управления транзакциями на основе блокировок, гарантирующие все свойства корректных расписаний, слишком сильно ограничивают возможности конкурентного выполнения. Транзакции слишком часто оказываются в состоянии ожидания, в результате существенно ограничивается пропускная способность СУБД. Необходимость глобальных блокировок делает эти протоколы немасштабируемыми и, следовательно, неприменимыми в распределенных системах.

В связи с этим в языке SQL предусмотрена возможность использования ослабленных критериев корректности (задаваемых уровнями изоляции), а реализации СУБД применяют протоколы, не гарантирующие корректность, но обеспечивающие более высокую пропускную способность. Один из таких протоколов,

широко используемых на практике, в том числе в PostgreSQL, называется *изоляцией снимков* (snapshot isolation, SI).

В этом разделе кратко излагается теория этого протокола (которая появилась значительно позже, чем сам протокол), а различные варианты реализации рассмотрим ниже в разделе 13.2.7.

Для того чтобы определить, какие расписания считаются корректными в соответствии с протоколом SI, необходимо связать с каждой транзакцией:

1) метку времени начала транзакции $\mathrm{STS}(t)$;
2) метку времени фиксации транзакции $\mathrm{CTS}(t)$;
3) интервал времени, в течение которого выполнялась транзакция $\tau(t) = (\mathrm{STS}(t), \mathrm{CTS}(t))$;
4) множество элементов данных, которые записываются транзакцией $\mathrm{WS}(t)$ (writeset).

При использовании протокола SI операции чтения возвращают значения, которые имели элементы данных на момент начала транзакции $\mathrm{STS}(t)$. Если транзакция модифицировала элемент данных, то она сама может прочитать новое значение, но другие транзакции смогут его получить только после фиксации транзакции, записавшей это значение.

Выполнение транзакций t_1, t_2 называется *конкурентным*, если $\tau(t_1) \cap \tau(t_2) \neq \varnothing$, т. е. интервалы времени, в течение которого они выполнялись, имеют непустое пересечение. Протокол SI допускает конкурентное выполнение транзакций, только если пересечение множеств записываемых ими элементов данных пусто (другими словами, не существует элемента данных, который записывается каждой из двух транзакций). Формально это можно записать компактной формулой:

$$\big(\tau(t_1) \cap \tau(t_2) = \varnothing\big) \vee \big(\mathrm{WS}(t_1) \cap \mathrm{WS}(t_2) = \varnothing\big).$$

В этом разделе мы не рассматриваем вопрос о том, какими средствами СУБД может гарантировать выполнение этого условия. Если условие нарушается, т. е. пересекаются как интервалы времени, в которые выполняются транзакции, так и множества записываемых значений, то одна из транзакций обрывается.

Заметим, что в реализациях (в том числе и в PostgreSQL) могут использоваться другие правила. Это может быть необходимо, в частности, для реализации ослабленных уровней изоляции SQL. Так, режим Read Committed никогда не

приводит к обрывам в PostgreSQL. Строго говоря, такие варианты нельзя считать реализациями SI.

Легко видеть, что правила SI гарантируют невозможность грязного чтения: транзакция может читать результаты работы других транзакций только после их фиксации.

Существуют сериализуемые по конфликтам расписания, которые не допускаются протоколом SI. Например, в следующем расписании интервалы времени выполнения и записываемые множества непусты, но расписание сериализуемо по конфликтам:

$$r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ c_2\ r_1(y)\ w_1(y)\ c_1.$$

В то же время протокол SI не гарантирует сериализуемость даже по конечному состоянию. Например, следующее расписание:

$$r_1(x)\ r_2(y)\ w_1(y)\ w_2(x)\ c_1\ c_2 \tag{13.3}$$

допускается SI, т. к. записываются разные элементы данных, но не сериализуемо по конечному состоянию, что доказывается вычислением семантик Эрбрана для этого расписания и для двух вариантов последовательного выполнения этих транзакций. В приведенном расписании конечные значения *x* и *y* зависят только от начальных состояний, а при последовательном выполнении, например $t_1\ t_2$, значение *x* будет зависеть от значения, записанного первой транзакцией в *y*. Это расписание является примером аномалии несогласованной записи.

Можно доказать [1; 29], что кроме аномалии несогласованной записи SI допускает аномалию только читающей транзакции. Для того чтобы исключить эти аномалии, вводится вариант *сериализуемого протокола SI* (serializable snapshot isolation, SSI) [45]. Формальное описание этого протокола основано на понятии *зависимости* между транзакциями:

- зависимость WR возникает, если t_1 записывает некоторый объект, который читает t_2: $t_1 \xrightarrow{WR} t_2$;
- зависимость WW возникает, если t_1 записывает некоторый объект, который t_2 замещает: $t_1 \xrightarrow{WW} t_2$;
- антизависимость RW возникает, если t_1 записывает некоторый объект, а t_2 читает предыдущее состояние этого объекта: $t_2 \xrightarrow{RW} t_1$.

Зависимости WW не могут возникать при выполнении условий SI, потому что такая зависимость означала бы, что множества записываемых элементов данных и интервалы времени выполнения транзакций пересекаются. Проверка этого запрета может выполняться по-разному в разных реализациях. Мы вернемся к этому вопросу при обсуждении протоколов управления транзакциями в разделе 13.2.

Подчеркнем, что в определении зависимостей важны значения элементов данных. Зависимость имеет место в том случае, если читается записанное значение или заменяется прочитанное. Это отличает зависимости от конфликтов, которые имеют место всегда, когда операции разных транзакций обрабатывают один и тот же элемент данных (и по крайней мере одна из них — операция записи), независимо от значений этого элемента данных.

Например, в расписании

$$r_1(x)\ c_1\ w_2(x)\ c_2\ w_3(x)\ c_3$$

имеются конфликты между всеми парами транзакций, однако зависимость есть только между первой и второй транзакциями.

Как и в случае сериализуемости по конфликтам, строится граф сериализуемости, вершинами которого являются транзакции, входящие в расписание, а дуги определяются перечисленными зависимостями. Заметим, что в этом графе могут появиться транзакции, которые не конкурируют друг с другом непосредственно. Легко доказать, что расписание, допускаемое протоколом SI, сериализуемо тогда и только тогда, когда построенный таким образом граф не имеет контуров.

Для расписания, иллюстрирующего аномалию несогласованной записи 13.3, граф сериализуемости имеет вид, показанный на рис. 13.1.5.

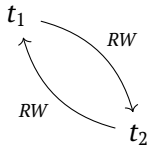


Рис. 13.1.5. Граф сериализуемости для несогласованной записи

Аномалия только читающей транзакции иллюстрируется расписанием:

$$r_2(x)\ r_2(y)\ r_1(y)\ w_1(y)\ c_1\ r_3(x)\ r_3(y)\ c_3\ w_2(x)\ c_2.$$

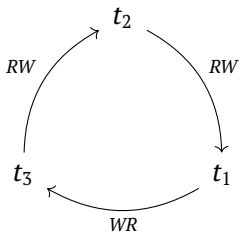


Рис. 13.1.6. Граф сериализуемости для аномалии только читающей транзакции

В этом расписании, граф сериализуемости которого представлен на рис. 13.1.6, имеются следующие зависимости:

- $t_1 \xrightarrow{WR} t_3$ по y;
- $t_2 \xrightarrow{RW} t_1$ по x;
- $t_3 \xrightarrow{RW} t_2$ по x.

Для расписаний, удовлетворяющих требованиям SI, можно доказать, что любой контур в графе сериализуемости имеет вид, показанный либо на рис. 13.1.5, либо на рис. 13.1.6 (причем транзакция t_3 — только читающая). Этот факт используется в реализации сериализуемого протокола SI для PostgreSQL с целью ускорения поиска контуров: если подграфы такого вида отсутствуют в графе сериализуемости, то в нем не может быть контуров.

13.1.3. Расписания с множественными версиями данных

В системах, одновременно выполняющих тысячи коротких транзакций, точное соблюдение последовательности транзакций, поступающих на вход, чаще всего не имеет большого значения. Более того, как показывает пример 13.2, логический порядок транзакций (в котором они сериализуются) может отличаться от фактической последовательности их выполнения. С другой стороны, одной из причин обрыва транзакции может быть попытка чтения данных, которые уже изменены другой, логически более поздней транзакцией.

Для того чтобы предотвратить обрыв такой опоздавшей транзакции, можно попытаться выполнить ее, используя немного устаревшие значения необходимых ей элементов данных. Это, конечно, не гарантирует возможность успешного завершения и фиксации, но потенциально дает возможность сократить

долю транзакций, которые обрываются из-за того, что не могут быть сериализованы.

Использование множественных версий данных для управления транзакциями не связано с каким-либо конкретным методом управления транзакциями и может рассматриваться в сочетании с различными критериями корректности.

Подчеркнем, что в рассматриваемом нами контексте каждая транзакция видит только одну согласованную версию базы данных. Множественные версии создаются только на сервере и недоступны за пределами СУБД. Существуют модели баз данных, позволяющие организовать обработку разных значений одного и того же элемента данных в одном запросе (и, следовательно, в одной транзакции). Такие модели называются *темпоральными*, однако эти модели не связаны с многоверсионностью для поддержки транзакций.

Для того чтобы формализовать многоверсионность, необходимо пересмотреть определения историй и расписаний. После этого можно пересмотреть определения критериев корректности и методы проверки того, что расписание относится к определенному классу. Конечно, наиболее интересны для нас расписания, сериализуемые по конфликтам.

Прежде всего необходимо различать версии одного элемента данных. Любая версия любого элемента данных записывается некоторой транзакцией. Напомним, что в нашей простой теоретической модели транзакция может записать элемент данных только один раз. Поэтому версию элемента можно идентифицировать номером той транзакции, которая ее записала: $w_i(x_i)$, т. е. операция записи, выполняемая транзакцией t_i, всегда записывает версию элемента данных с тем же номером. Для указания того, какую версию читает операция чтения другой транзакции, будем использовать запись $r_j(x_i)$. Однажды записанная версия элемента данных никогда не изменяется другой транзакцией, любая операция записи создает новую версию. Пока мы не предполагаем, что версии одного элемента данных как-либо упорядочены. Операция чтения, конечно, может читать только такую версию, запись которой предшествует этой операции чтения: $w_i(x_i) < r_j(x_i)$.

Будем считать, что начальные значения всех элементов данных имеют номер версии 0. Рассмотрим многоверсионное расписание:

$$r_1(x_0)\ r_2(x_0)\ w_2(x_2)\ r_2(y_0)\ w_2(y_2)\ r_1(y_2).$$

В этом расписании каждая операция чтения считывает последнюю записанную до нее версию элемента данных. Такие расписания называются *моноверсионными* (поскольку фактически множественные версии в таких расписаниях

не используются). Это расписание некорректно, т. к. оно содержит аномалию несогласованного чтения. Можно, однако, сделать это расписание корректным, если использовать многоверсионность. Для этого достаточно изменить только последнюю операцию чтения:

$$r_1(x_0)\ r_2(x_0)\ w_2(x_2)\ r_2(y_0)\ w_2(y_2)\ r_1(y_0).$$

Расписание стало корректным, потому что все значения, обрабатываемые операциями этого расписания, оказываются такими же, как при серийном выполнении $t_1\ t_2$. Только читающая транзакция t_1 оказалась логически выполненной раньше, чем t_2, хотя фактически завершилась позже. Однако, если первая транзакция выполнит операцию записи, как в следующем расписании:

$$r_1(x_0)\ r_2(x_0)\ w_2(x_2)\ r_2(y_0)\ w_2(y_2)\ r_1(y_0)\ w_1(y_1),$$

то расписание снова станет некорректным из-за появления в нем аномалии потерянного обновления. Устранить эту аномалию в данном расписании без перестановки операций (только за счет множественных версий) невозможно.

Упорядочение операций над одним элементом данных в многоверсионных расписаниях не требуется, т. е. операции чтения разных версий одного элемента данных упорядочивать не обязательно. Однако требуется, чтобы операция чтения следовала за операцией, которая записала читаемую версию: $w_i(x_i) < r_j(x_i)$. Это единственный вид конфликтов, которые встречаются в многоверсионных расписаниях.

Многоверсионное расписание можно использовать для вычисления семантик Эрбрана и на их основе определить критерии корректности. Для того чтобы критерий корректности был практически полезным, необходимо чтобы корректное расписание было эквивалентно (в смысле какого-либо отношения эквивалентности) серийному *моно*версионному расписанию.

В многоверсионном расписании конфликтами считаются только конфликты между операциями над одной версией, т. е. между созданием версии и последующими операциями чтения этой версии. Этих конфликтов слишком мало для получения корректных результатов, поэтому граф сериализуемости дополняется дугами, связывающими операции записи различных версий одного элемента. Таким образом, вводится упорядочение версий для каждого элемента.

Неформально можно сказать, что упорядочение версий каждого элемента задает некоторое упорядочение транзакций (в той последовательности, в которой они записывали этот элемент данных). Для того чтобы многоверсионное

расписание могло быть эквивалентно *моно*версионному, необходимо чтобы эти упорядочения транзакций совпадали для всех элементов данных.

Можно формально доказать, что отсутствие контуров в расширенном графе сериализуемости необходимо и достаточно, для того чтобы многоверсионное расписание было сериализуемо по конфликтам. При отсутствии контуров будет обеспечено и одинаковое упорядочение операций обновления для разных элементов, упомянутое выше.

13.1.4. Восстановимость

Понятие восстановимости кратко обсуждалось в разделе 6.3 первой части книги. Здесь это и связанные с ним понятия рассматриваются более детально.

Семантическая корректность, рассмотренная в предыдущем разделе, является далеко не единственным требованием к управлению транзакциями. Не менее важно обеспечить согласованность при наличии разного рода сбоев, приводящих к обрывам транзакций или отказу сервера базы данных.

Для того чтобы реализовать принцип атомарности при обрывах транзакций, т. е. отменить результаты выполнения операций оборванной транзакции, вводится операция обращения записи $w_i^{-1}(x)$. По определению эта операция восстанавливает значение элемента данных x в то состояние, в котором элемент был непосредственно перед выполнением прямой операции $w_i(x)$.

Расписание, в котором операция записи непосредственно предшествует операции ее обращения, эквивалентно расписанию, из которого данная пара операций исключена. Для того чтобы корректно выполнить обрыв транзакции, необходимо обратить все операции записи, выполненные этой транзакцией, в обратном порядке, и затем зафиксировать эту транзакцию. Любые операции чтения при этом можно игнорировать. Такая реализация операции обрыва называется *откатом* (rollback). Зачастую операции обрыва и отката не различают. Так, в языке SQL обрыв транзакции выполняется оператором `ROLLBACK`.

Наличие операций отката в расписании приводит к необходимости некоторого пересмотра критериев корректности, в частности понятия сериализуемости по конфликтам. Детальный анализ можно найти в книге [62].

Проиллюстрируем, как откат транзакций записывается в расписании, небольшим примером.

Рассмотрим следующее расписание:

$$r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)\ r_1(z)\ a_1.$$

В нем операция обрыва a_1 заменяется на обратные операции для всех операций записи этой транзакции и последующую фиксацию c_1:

$$r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)\ r_1(z)\ w_1^{-1}(y)\ w_1^{-1}(x)\ c_1.$$

Заметим, что алгоритм выполнения обратных операций может быть различным в разных СУБД. В системе PostgreSQL новые значения никогда не записываются на место старых, поэтому возврат к предыдущему значению достаточно прост и не требует фактической записи этого значения элемента данных.

Использование операции отката для реализации обрывов накладывает ограничения на допустимость расписаний.

Рассмотрим пример:

$$w_1(x)\ w_2(x)\ c_2\ a_1.$$

Это расписание некорректно, потому что операция $w_1^{-1}(x)$, реализующая обрыв транзакции t_1, восстановит значение x по состоянию на начало этой транзакции, а результаты транзакции t_2, которая уже была зафиксирована, будут потеряны, что нарушает свойство долговечности.

Заметим, что все другие варианты завершения транзакций в этом расписании корректны:

$$w_1(x)\ w_2(x)\ c_1\ a_2,$$
$$w_1(x)\ w_2(x)\ a_2\ a_1,$$
$$w_1(x)\ w_2(x)\ c_1\ c_2.$$

Расписание называется *восстановимым*, если для любой пары операций $w_i(x) \rightarrow w_j(x)$ либо фиксация транзакции t_i предшествует фиксации t_j, либо t_j обрывается. Класс восстановимых расписаний обозначается RC (recoverable).

Таким образом, если в расписании $w_1(x)\ w_2(x)$ транзакция t_2 пытается выполнить фиксацию, то эта операция должна быть отложена до тех пор, пока не зафиксируется транзакция t_1. Если при этом t_1 обрывается, то перед ее обрывом выполняется обрыв t_2 (несмотря на то что t_2 была готова зафиксироваться). Такой обрыв транзакции t_2 называется *каскадным*.

Свойство восстановимости расписаний никак не соотносится с сериализуемостью: восстановимое расписание может быть не сериализуемо даже по конечному состоянию, а сериализуемое по конфликтам расписание может не быть восстановимым.

Чтобы предотвратить каскадные обрывы, необходимо отложить выполнение операции $w_j(x)$ до завершения транзакции t_i. Расписания, удовлетворяющие этому условию, называются *бескаскадными*, однако этот класс расписаний не имеет практического значения. Класс бескаскадных расписаний обозначается ACA. Можно доказать, что ACA $\subset$ RC и это включение — строгое. Класс ACA также не соотносится с сериализуемостью (докажите!).

Еще более сильное условие предотвращает аномалию грязного чтения и состоит в том, что выполнение любой операции транзакции t_j над элементом данных, записанным транзакцией t_i, откладывается до завершения t_i. Расписания, удовлетворяющие этому условию, называются *строгими* (strict), этот класс расписаний принято обозначать ST. Можно доказать, что ST $\subset$ ACA. То, что любое ST-расписание бескаскадно, непосредственно следует из определений, а для доказательства того, что классы не совпадают, нужно предъявить бескаскадное расписание, которое не входит в класс ST. Читатель может построить такое расписание в качестве упражнения.

Наконец, расписания, в которых для любой операции транзакции t_i любая конфликтующая операция транзакции t_j над тем же элементом данных откладывается до завершения t_i, называются *точными* (rigorous). Класс точных расписаний обозначается RG.

Можно доказать, что RG $\subset$ ST $\cap$ CSR, т. е. любое расписание из класса RG будет строгим и сериализуемым. Расписания из класса RG, следовательно, обладают очень хорошими свойствами как с точки зрения корректности, так и с точки зрения восстановимости. Однако этот класс оказывается очень узким и слишком сильно ограничивает возможности конкурентного выполнения транзакций, поэтому он имеет только теоретическое значение.

Поскольку в расписаниях из класса SI транзакции могут читать только зафиксированные значения, можно сказать, что SI $\subset$ ST и, следовательно, расписания из этого класса бескаскадны и восстановимы. Конечно, это достигается потому, что транзакции, которые, возможно, пришлось бы обрывать каскадно, обязательно имеют непустое пересечение множества записи с множеством записи текущей транзакции и поэтому, в соответствии с правилами SI, обрываются безусловно (а не только при необходимости каскадного обрыва).

13.1.5. Дополнительные свойства классов расписаний

Практически полезные классы расписаний должны обладать еще некоторыми свойствами.

Класс расписаний называется *префиксно замкнутым*, если любой префикс расписания из этого класса также принадлежит этому классу. Другими словами, если в транзакциях, входящих в состав корректного расписания (в смысле рассматриваемого класса расписаний), оставить только операции, которые попали в префикс, то полученное таким образом расписание тоже будет корректным в том же смысле. Свойство префиксной замкнутости обеспечивает сохранение корректности (в смысле этого класса расписаний) при отказах системы: при неполном выполнении расписания база данных остается в корректном состоянии или может быть в него приведена.

Классы расписаний, сериализуемых по видимому состоянию VSR и по конфликтам CSR, а также класс SI являются префиксно замкнутыми, а класс расписаний, сериализуемых по конечному состоянию FSR, и класс восстановимых расписаний RC — не являются.

Класс расписаний называется *монотонным*, если исключение из расписания любой транзакции сохраняет принадлежность расписания к этому классу. Это свойство позволяет исключать из расписания оборванные транзакции.

Все классы сериализуемых расписаний и класс SI являются монотонными.

Заканчивая раздел, заметим, что класс расписаний SI, хотя и не гарантирует сериализуемость, однако обладает дополнительными полезными свойствами, в том числе восстановимостью, префиксной замкнутостью и монотонностью. Всеми этими свойствами обладает, конечно, и класс сериализуемых расписаний SSI. По-видимому, сочетание этих свойств и привело к тому, что именно этот класс чаще всего реализуется в высокопроизводительных промышленных системах, в том числе в PostgreSQL.

13.2. Диспетчеры и протоколы

Диспетчером транзакций будем называть программу (обычно входящую в состав ядра СУБД), которая отвечает за корректность выполнения транзакций. На вход диспетчера поступает упорядоченный (возможно, частично) поток запросов на выполнение операций над базой данных, а на выходе должен получаться

поток, выполнение которого удовлетворяет каким-либо критериям корректности (не обязательно рассмотренным в предыдущих разделах этой главы).

Для того чтобы добиться корректности, диспетчер может:

- передавать поступившую операцию на выполнение немедленно;
- задерживать выполнение операции, пока не будут удовлетворены некоторые условия, например завершение другой транзакции;
- запрещать выполнение операций, что, в силу принципа атомарности, влечет за собой обрыв транзакции, которая запросила выполнение этой операции.

Диспетчер обеспечивает выполнение некоторого набора правил для каждой транзакции. Обычно подобные совокупности правил, обязательных для независимых взаимодействующих участников, называются *протоколами* (в отличие от алгоритмов, предписывающих определенные действия). Примерами протоколов, не связанных непосредственно с управлением транзакциями, могут служить сетевые протоколы (такие, как TCP/IP или HTTP), а также дипломатические протоколы.

Протоколы управления транзакциями определяются так, чтобы гарантировать соблюдение требований к корректности расписаний при условии соблюдения правил протокола всеми транзакциями. Таким образом, диспетчер обеспечивает выполнение правил протокола каждой транзакцией, что в итоге приводит к соблюдению требований корректности расписания в целом.

13.2.1. Требования и критерии оценки

Кроме корректности, диспетчер транзакций должен обеспечить высокую производительность системы.

Рассмотрим диспетчер, который запускает выполнение операций той транзакции, операции которой начали поступать раньше остальных, а операции всех других транзакций ставит в очередь. После завершения транзакции такой диспетчер выбирает из очереди операцию, поступившую раньше других, и таким образом выбирает следующую транзакцию для выполнения. Этот диспетчер обеспечит получение на выходе серийного расписания, все критерии корректности будут выполнены, однако производительность системы будет крайне низкой из-за длительного ожидания в очереди.

Основной метрикой, определяющей качество диспетчера транзакций, является пропускная способность системы, определяемая как количество транзакций, которые система может выполнить за единицу времени. Конечно, пропускная способность зависит от возможностей оборудования, от алгоритмов выполнения операций в СУБД, от сложности этих операций и от размеров транзакций. Однако, кроме всех этих факторов, пропускная способность зависит и от протокола, реализуемого диспетчером: так, описанный в предыдущем абзаце диспетчер, по существу, блокирует возможность параллельного или псевдопараллельного выполнения транзакций, даже если оборудование и алгоритмы СУБД это позволяют.

Для практического измерения влияния свойств протоколов на пропускную способность используются эталонные тесты (benchmarks), в которых все остальные параметры, влияющие на пропускную способность, зафиксированы. Обычно эталонный тест предписывает определенную структуру и размеры базы данных, типы и размеры транзакций, поток которых поступает в систему при выполнении эталонного теста.

Другой важной метрикой, характеризующей качество протокола, является доля транзакций, которые завершаются обрывом из-за невозможности включить их в расписание, не нарушая правил протокола. Такие обрывы нежелательны, потому что, во-первых, они снижают фактическую пропускную способность системы, и во-вторых, на выполнение оборванных транзакций затрачиваются вычислительные ресурсы, а результаты выполнения при этом уничтожаются. Конечно, доля обрывов также зависит от многих параметров, поэтому и ее измеряют на эталонных тестах.

Обе характеристики протоколов (пропускная способность и доля обрывов) зависят от того, как соотносятся по размеру активная часть базы данных (то множество элементов данных, которые обрабатываются операциями транзакций) и транзакции: чем меньше база данных и чем длиннее транзакции, тем чаще одновременное выполнение таких транзакций оказывается невозможным.

В реальных базах данных активность всех элементов данных не может быть одинаковой, однако наличие элементов, которые очень часто обновляются очень многими транзакциями (такие элементы называются горячими точками, hot spots) может существенно ухудшить фактические значения характеристик системы. Принято считать, что наличие таких горячих точек указывает на неудачное проектирование базы данных.

Протоколы, предотвращающие появление нежелательных (скажем, несериализуемых) расписаний, называются *консервативными*. Такие протоколы тем или

иным способом задерживают выполнение некоторых транзакций и тем самым снижают пропускную способность системы. *Неконсервативные* протоколы позволяют увеличить степень параллелизма при выполнении транзакций, но могут увеличивать долю обрывов из-за невозможности удовлетворить требования протокола. Заметим, что консервативность не предотвращает полностью необходимость протокольных обрывов.

13.2.2. Блокировки

Большинство диспетчеров, реализованных в промышленных системах управления базами данных, для управления операциями транзакций использует *блокировки* (locks), называемые также *замка́ми*.

Каждая блокировка связана с некоторым объектом базы данных (иногда с набором объектов) и с операцией, которая должна быть выполнена над этим объектом. В простейшей модели базы данных, которая используется в этой главе, объектом является элемент данных, а операцией — чтение или запись.

Если для некоторой операции используется блокировка, то она устанавливается диспетчером до выполнения этой операции (не обязательно непосредственно перед операцией) и должна быть снята после завершения операции и не позже, чем во время завершения транзакции. Другими словами, все блокировки, установленные для транзакции, снимаются при ее завершении, если они не были сняты раньше.

Блокировки могут быть *совместимыми* или *несовместимыми* друг с другом. Допускается одновременная установка совместимых блокировок разными транзакциями, а попытка установить блокировку, несовместимую с уже установленной, обычно приводит к тому, что транзакция переводится в состояние ожидания (ставится в очередь) и выводится из ожидания, когда все ранее установленные несовместимые блокировки снимаются. Некоторые протоколы могут обрывать транзакции при попытке установить несовместимую блокировку.

Механизмы, близкие по назначению к блокировкам, используются не только в СУБД. Подчеркнем важное отличие блокировок от таких примитивов синхронизации параллельных процессов, как семафоры и мьютексы (mutex): последние связаны с критическими участками программного кода, а не с объектами данных. Конечно, для реализации механизма блокировок должны использоваться критические участки кода, защищаемые механизмами синхронизации операционной системы, но такая синхронизация необходима только на время

выполнения операций установки и снятия блокировок, а не на все время, когда блокировка установлена.

Как корректность, так и производительность протоколов, использующих блокировки, зависит от того, какие блокировки считаются совместимыми. Для того чтобы обеспечить, скажем, сериализуемость по конфликтам, необходимо (но не достаточно) сделать несовместимыми блокировки на операции, находящиеся в конфликте. Напомним, что конфликтом называется пара операций разных транзакций, работающих с одним объектом данных, и таких, что по крайней мере одна из них записывает новое значение этого объекта.

Влияние диспетчера транзакций на пропускную способность системы зависит от того, насколько часто блокировки оказываются несовместимыми, и от *гранулярности* (относительного размера) объектов, на которые блокировки устанавливаются. Например, таблиц в базе данных значительно меньше, чем строк таблиц, поэтому таблицы обладают более крупной гранулярностью, чем строки, а значения отдельных атрибутов — более мелкой, чем строки.

Это утверждение можно проиллюстрировать простым примером. Диспетчер, который использует всего одну блокировку, связанную со всей базой данных в качестве объекта данных и несовместимую с ней самой, будет вырабатывать серийное расписание, если эта блокировка устанавливается для разных транзакций. В этом случае причина низкой пропускной способности в том, что вся база данных рассматривается как один объект. Однако и противоположная ситуация, когда блокировки устанавливаются на слишком мелкие объекты, может приводить к снижению производительности из-за слишком больших затрат на управление блокировками.

Далее будем предполагать, что несовместимыми являются только блокировки, устанавливаемые для операций, находящихся в конфликте (или, что эквивалентно, не коммутирующих операций). В соответствии с определением конфликта блокировка для операции записи несовместима с любой другой блокировкой того же элемента данных. Блокировки, несовместимые с любой другой блокировкой того же элемента данных, называются *монопольными* или *исключающими*.

В отличие от монопольных блокировок (на запись) блокировки на чтение не препятствуют установке других блокировок того же элемента данных на чтение, потому что операции чтения никогда не находятся в конфликте. Иногда блокировки, допускающие установку некоторых видов блокировок того же элемента данных другими транзакциями, называются *разделяемыми*.

В отличие от понятий «блокировка для чтения» и «блокировка на запись», термины «монопольная блокировка» и «разделяемая блокировка» не предполагают какую-либо семантику выполняемых операций, а определяют только свойства блокировок по отношению к другим блокировкам.

13.2.3. Двухфазные протоколы, использующие блокировки

Наиболее широко известным протоколом управления транзакциями, использующим блокировки, является *двухфазный протокол блокирования* (two-phase locking, 2PL). В соответствии с этим протоколом для любой транзакции могут устанавливаться любые блокировки в любом порядке, однако после того, как хотя бы одна из установленных ранее блокировок снята, установка новых блокировок для этой транзакции запрещена. Таким образом, работа транзакции состоит из двух фаз: на первой фазе блокировки устанавливаются, на второй — только снимаются.

Заметим, что любые протоколы, основанные на блокировках, управляют только относительным порядком выполнения операций, с которыми связаны эти блокировки, но никак не зависят ни от типа блокировок (монопольные или нет), ни от типа объектов, ни даже от класса системы (это совсем не обязательно должна быть СУБД). Однако для того чтобы протокол был полезен для управления транзакциями, необходимо накладывать некоторые ограничения на используемые блокировки, различные для разных протоколов. Для семейства двухфазных протоколов предполагается, что совместимость блокировок определяется наличием конфликтов между операциями, для которых устанавливаются блокировки.

Двухфазный протокол хорошо подходит для СУБД, поскольку приложение может в ходе выполнения транзакций определять, какие еще элементы данных понадобятся для ее завершения. В этом смысле 2PL стал очень большим улучшением по сравнению с протоколами, которые были известны до его опубликования в 1976 г. Так, в некоторых операционных системах все необходимые блокировки устанавливались до начала выполнения задания и удерживались до его окончания, что, во-первых, приводило к существенному ограничению конкурентности, и во-вторых, список всех необходимых ресурсов требовалось предъявлять до начала выполнения.

Класс расписаний, которые могут быть получены в результате применения некоторого протокола управления транзакциями P, обозначается $\mathrm{Gen}(P)$. Так, класс расписаний, порождаемый протоколом 2PL, обозначается $\mathrm{Gen}(2\mathrm{PL})$.

Расписания, получаемые на выходе двухфазного протокола, являются сериализуемыми по конфликтам: $\mathrm{Gen}(2\mathrm{PL}) \subset \mathrm{CSR}$.

Для доказательства этого утверждения достаточно показать, что граф сериализуемости любого расписания $s \in \mathrm{Gen}(2\mathrm{PL})$ не содержит контуров. Пусть $b(t)$ и $e(t)$ обозначают соответственно операции установки последней блокировки и снятия первой блокировки для транзакции t. По определению двухфазного протокола $b(t) < e(t)$ для любой транзакции t. Пусть $l_i(x)$ и $u_i(x)$ обозначают операции установки и снятия блокировки на элемент данных x для транзакции t_i и пусть в расписании имеется конфликт по элементу данных x между транзакциями t_1 и t_2. Тогда

$$b(t_1) < e(t_1) < u_1(x) < l_2(x) < b(t_2) < e(t_2).$$

Следовательно, для любого пути t_1, t_2, …, t_n в графе сериализуемости имеет место

$$b(t_1) < e(t_1) < b(t_2) < e(t_2) < \ldots < b(t_n) < e(t_n),$$

и поэтому путь не может быть контуром.

Протокол 2PL, как следует из доказанного утверждения, гарантирует сериализуемость, но не обеспечивает восстановимость расписаний. Действительно, восстановимость зависит от порядка фиксации и обрывов, а не от упорядочения операций чтения и записи, а завершение транзакций никак не регулируется блокировками.

Вариант, который называется *строгим двухфазным протоколом* (strict 2PL, S2PL), отличается тем, что блокировки, установленные для операций записи, не снимаются до завершения транзакции.

Строгий двухфазный протокол генерирует расписания, которые сериализуемы по конфликтам и являются строгими: $\mathrm{Gen}(\mathrm{S2PL}) \subset \mathrm{CSR} \cap \mathrm{ST}$. Доказательство оставляется читателю в качестве упражнения.

Таким образом, на выходе S2PL получаются строгие, а следовательно, бескаскадные и восстановимые сериализуемые расписания.

Наконец, *точный двухфазный протокол* (rigorous 2PL, strong strict 2PL, SS2PL) требует, чтобы все блокировки удерживались до завершения транзакции.

Класс расписаний, получаемых на выходе точного двухфазного протокола, совпадает с классом точных расписаний: $\mathrm{Gen}(\mathrm{SS2PL}) = \mathrm{RG}$. Доказательство также оставляется читателю.

Протокол SS2PL полностью предотвращает феномен грязного чтения, однако получающийся класс расписаний оказывается слишком узким и поэтому такой протокол не позволяет добиться высокой пропускной способности системы. По этой причине протокол SS2PL следует рассматривать как красивое теоретическое решение, очень редко (если вообще когда-либо) применяемое на практике. Более точно, этот протокол был реализован во многих системах (до появления SI), однако приложения, работающие с такими СУБД, использовали, как правило, ослабленные критерии согласованности, отказываясь от сериализуемости. Возможно, именно ограниченность этого протокола привела к включению ослабленных уровней изоляции в стандарт SQL.

Заметим, что для доказательства сериализуемости по конфликтам важно, что правила установки блокировок связаны с выявлением конфликтов. В частности, необходимо устанавливать блокировки как на чтение, так и на запись.

Все варианты двухфазного протокола 2PL характеризуются низкой частотой обрывов транзакций, однако ограничивают пропускную способность, поэтому в PostgreSQL такие протоколы не применяются.

13.2.4. Тупики

В системах, в которых используются блокировки (или другие средства синхронизации доступа к общим ресурсам, например семафоры), могут возникать ситуации *тупика*. Предположим, что для всех операций устанавливаются блокировки (как это требуется в двухфазных протоколах) и рассмотрим следующее расписание:

$$w_1(x)\ w_2(y)\ r_1(y)\ r_2(x).$$

В этом расписании транзакция t_1 ожидает, когда транзакция t_2 снимет блокировку на элемент данных y, при этом транзакция t_2 ожидает снятия блокировки на x, которая никогда не будет снята, поскольку эта транзакция тоже находится в состоянии ожидания.

Важно отметить, что для возникновения тупика существенна несовместимость блокировок, а не тип выполняемых операций. В PostgreSQL блокировки на чтение обычно не устанавливаются, поэтому ситуация тупика возникает, если в приведенном выше расписании все операции будут операциями записи.

В некоторых изданиях ситуация тупика называется *взаимной блокировкой*, иногда использовался термин *смертельное объятие*, однако термин «тупик» пред-

почтительнее, потому что слово «блокировка» (без уточнения «взаимная») используется в другом смысле.

В отличие от систем других классов (например, операционных систем) в базах данных ситуация тупика не является катастрофической, потому что для ее устранения достаточно оборвать некоторые из оказавшихся в тупике транзакции, а операция обрыва в силу принципа атомарности не может привести к утрате согласованности или к разрушению данных.

В связи с тупиками рассматриваются и решаются следующие задачи:

- обнаружение тупиков;
- разрешение тупиков;
- предотвращение тупиков.

Точный алгоритм для обнаружения тупиков основан на понятии *графа ожиданий*. Вершины этого графа представляют активные транзакции, а направленные дуги соединяют ожидающие транзакции с теми транзакциями, которые удерживают блокировки на ожидаемый элемент данных. Ситуация тупика имеет место тогда и только тогда, когда граф ожиданий содержит контур.

Этот алгоритм, однако, трудно реализуем в распределенных системах, поскольку информация о состояниях ожиданий может утратить актуальность за время, необходимое для сбора этой информации по сети, что может привести к обнаружению ложных тупиков.

После того как тупик обнаружен, его необходимо разрешить. Для этого достаточно оборвать одну из транзакций, оказавшихся на контуре. Существуют различные эвристические стратегии выбора жертвы: случайный выбор, с позднейшим временем начала и т. п.

Во многих СУБД в качестве альтернативы для поиска тупиков используется ограничение времени, в течение которого транзакция может находиться в состоянии ожидания.

Все упомянутые варианты двухфазного протокола блокирования несвободны от тупиков, однако существует несколько модификаций, в которых возникновение тупиков предотвращается, например:

- обрыв вместо ожидания — транзакция, которая пытается установить несовместимый замок, обрывается;

- обрыв по приоритету — все транзакции запускаются с низким приоритетом, при попытке установить несовместимую блокировку обрывается транзакция с меньшим приоритетом и ее приоритет увеличивается при повторном запуске;
- ожидание по приоритету — если несовместимый замок пытается установить транзакция с более высоким приоритетом, то низкоприоритетная обрывается, а если наоборот — то низкоприоритетная ожидает;
- по времени старта — ожидать может только транзакция, стартовавшая позже транзакции, удерживающей блокировку.

Все варианты протоколов с предотвращением тупиков приводят к существенному увеличению доли оборванных транзакций, поэтому их целесообразно использовать только в тех случаях, когда состояние ожидания нежелательно. Например, вариант двухфазного протокола блокирования с обрывами вместо ожиданий применяется в системах транзакционной оперативной памяти.

13.2.5. Другие протоколы на основе блокирования

Не все протоколы, использующие блокировки, являются двухфазными.

В учебниках по СУБД обычно рассматривается протокол WTL (write-only tree locking), предполагающий, что база данных структурирована в виде иерархического дерева. Работа этого протокола основана на том, что структура дерева накладывает ограничения на пути навигации в данных: допускаются только переходы от предков к потомкам. Этот протокол свободен от тупиков и мог бы применяться, например, для индексов. Однако для индексных структур известны более эффективные протоколы блокирования.

Простая модель базы данных, предусматривающая только операции чтения и записи, недостаточна для поддержки высокоуровневых языков запросов. Дело в том, что при ассоциативном доступе к данным (выборке данных по условию, например, как в операторах SQL, содержащих предложение WHERE) множество элементов данных (строк таблиц), обрабатываемых оператором, становится известно только во время выполнения оператора, что исключает возможность предварительной блокировки этих данных.

Элегантное решение (предложенное одновременно с двухфазным протоколом блокирования в той же статье [59] и дополняющее этот протокол) состоит в применении *предикатных блокировок*, которые блокируют все записи, удовлетво-

ряющие заданному условию. Однако применение предикатных блокировок ограничивается тем, что задача проверки совместимости таких блокировок является в общем случае алгоритмически неразрешимой, а эффективные полиномиальные алгоритмы известны только для очень узкого класса предикатов (конъюнкций простых условий).

Поэтому на практике применяются *мультигранулярные протоколы блокирования*, предусматривающие блокировки на объекты базы данных разного уровня (например, таблицы и строки таблиц) и накладывающие дополнительные ограничения на последовательность установки и снятия таких блокировок.

Применение блокировок на всю таблицу, естественно, отрицательно влияет на возможность конкурентного выполнения транзакций, даже если такие блокировки устанавливаются лишь на короткий промежуток времени. Это служит дополнительным основанием для отказа от сериализуемости. Однако отказ от использования предикатных (или мультигранулярных) блокировок может приводить к возникновению новых видов аномалий, которые появляются при повторном выполнении операций поиска.

- Могут появиться новые строки, добавленные другими зафиксированными транзакциями и удовлетворяющие условиям поиска. Такие строки называются *фантомами*.
- Строки, удовлетворяющие условиям поиска, могут быть изменены или удалены другой зафиксированной транзакцией и поэтому не будут найдены при повторном выполнении поиска. Такая аномалия называется *неповторяемым чтением*.

13.2.6. Протокол на основе меток времени

Критерии корректности расписаний не предполагают применение блокировок. В этом подразделе представлен *протокол на основе меток времени* (timestamp, TS), обеспечивающий сериализуемость расписаний с помощью альтернативной системы правил.

В рамках этого протокола каждая транзакция t_i в начале своей работы (т. е. при регистрации транзакции в диспетчере) получает метку времени τ_i. Точность этих меток должна быть достаточной, для того чтобы все метки времени были уникальными.

Кроме этого, каждому элементу данных x присваивается две метки времени: метка транзакции, записавшей последнее значение этого элемента $\tau_w(x)$ и метка последней транзакции, прочитавшей это значение $\tau_r(x)$. В реализации эти метки нужны только для элементов данных, которые обрабатываются активными транзакциями.

Протокол TS предусматривает следующие правила выполнения транзакций:

TS-R Операция $r_i(x)$ допустима, если $\tau_w(x) < \tau_i$, т. е. последняя запись элемента данных x была выполнена транзакцией, стартовавшей раньше, чем читающая транзакция. Если это условие не выполнено, то операция чтения отвергается и транзакция обрывается.

TS-W Операция записи $w_i(x)$ допустима, если $\tau_r(x) < \tau_i$, т. е. последнее чтение элемента данных было выполнено транзакцией, стартовавшей раньше, чем транзакция, записывающая новое значение. Если это условие нарушено, то операция отвергается и транзакция должна быть оборвана.

Протокол TS генерирует расписания, сериализуемые с сохранением порядка транзакций: $\text{Gen}(\text{TS}) \subset \text{OCSR}$.

Для доказательства заметим, что правила протокола TS гарантируют, что направление конфликтов совпадает с их направлением в эквивалентном серийном расписании, в котором транзакции упорядочиваются по возрастанию их меток времени.

При реализации протокола TS необходимо гарантировать, что все операции над каждым элементом данных выполняются строго последовательно. Это может быть необходимо, если оборудование и операционная система допускают параллельное выполнение. (В протоколах, основанных на блокировках, нет необходимости специально заботиться об этом, потому что для этих протоколов необходимая синхронизация обеспечивается блокировками попутно с сериализуемостью.)

Для того чтобы выполнение операций над каждым элементом данных было последовательным, можно использовать блокировки, устанавливаемые на время выполнения одной операции. В некоторых системах блокировки, устанавливаемые на короткое время для предотвращения одновременной модификации разными процессами, называются *задвижками* (latch), в других этот термин применяется в более узком смысле (для конкретных типов объектов). Такие блокировки устанавливаются только на один элемент данных, поэтому они могут устанавливаться локально даже в распределенных системах, не могут привести к ситуации тупика и никак не связаны с обеспечением сериализуемости

(или другого уровня изоляции). В книге [62] показано, каким образом понятие задвижки может быть обобщено с помощью механизма многоуровневых транзакций.

Базовая версия протокола TS, рассмотренная в этом подразделе, редко (если вообще) использовалась в реальных СУБД, скорее всего, потому, что доля обрывов оказывается выше, чем для двухфазных протоколов блокирования, в особенности в тех случаях, когда транзакции используют относительно большую часть базы данных (т. е. либо активная часть базы данных невелика, либо транзакции читают и записывают много элементов данных).

В последние годы, однако, появилось несколько экспериментальных систем, которые используют варианты протокола TS в распределенных масштабируемых вычислительных системах. Применение этих вариантов протокола позволило повысить пропускную способность на несколько порядков по сравнению с протоколами, использующими блокировки.

13.2.7. Реализации протокола SI

Любая реализация протокола SI должна обеспечить выполнение следующих двух требований:

1) возможность чтения значений данных, которые были зафиксированы на момент начала транзакции;
2) невозможность обновления одних и тех же данных конкурентно выполняемыми транзакциями.

Для реализации этих требований обычно используются блокировки, устанавливаемые в соответствии со следующими правилами:

SI1 Используются только блокировки для операций записи.

SI2 Транзакция, пытающаяся установить блокировку, несовместимую с блокировкой, установленной другой транзакцией, обрывается. Это правило иногда формулируется следующим образом: побеждает первый записавший (first writer wins).

SI3 Изменения, сделанные транзакцией, становятся доступными для других транзакций во время ее фиксации (хранятся до ее завершения в отдельных областях памяти, недоступных другим транзакциям, которые в это время могут читать ранее зафиксированные копии элементов данных).

Таким образом, использование блокировок в этом варианте протокола SI не связано с понятием конфликта, а только предотвращает одновременную модификацию элемента данных разными транзакциями. Напомним, что, в соответствии с определением класса расписаний SI, множества элементов данных, обновляемых одновременно выполняемыми транзакциями, не могут пересекаться.

Может показаться, что правило SI3 автоматически превращает реализацию протокола SI в многоверсионную. Действительно, это правило очень легко реализуется в многоверсионном варианте, в частности таким способом протокол SI реализован в системе PostgreSQL. Отличие состоит в том, что новая копия данных, создаваемая транзакцией в соответствии с правилом SI3, недоступна для других транзакций до фиксации, а в момент фиксации замещает предыдущую зафиксированную версию, делая ее недоступной. В многоверсионном варианте протокола разные транзакции могут одновременно читать различные версии одного элемента данных.

Как показано выше в разделе 13.1.2, выполнение приведенных требований не гарантирует сериализуемость, однако, очевидно, предотвращает грязное чтение. Простые реализации протокола SI допускают появление аномалии неповторяющегося чтения, поскольку значения элементов данных могут измениться после фиксации другой транзакции.

Наконец, заметим, что требования абстрактного протокола SI предполагают проверку интервалов времени, в течение которых выполняются транзакции, поэтому проверку этих условий легко реализовать на основе меток времени (подобно протоколу TS) без применения блокировок.

13.2.8. Многоверсионные протоколы

Большинство протоколов управления транзакциями, в том числе все рассмотренные в этой главе, имеют варианты, работающие с множественными версиями элементов данных.

Во всех случаях применение множественных версий сокращает долю транзакций, обрываемых по протокольным причинам: множественные версии дают возможность нормально зафиксировать транзакции, которые читают данные после того, как эти элементы данных были изменены логически более поздней транзакцией (т. е. более поздней в порядке сериализации). Конечно, попытка записи таких элементов данных все равно приводит к обрыву транзакции.

Множественные версии особенно полезны в тех случаях, когда значительную часть нагрузки составляют только читающие транзакции, потому что такие транзакции всегда могут быть зафиксированы. Это утверждение, казалось бы, опровергается аномалией только читающей транзакции, однако эта аномалия никак не связана с многоверсионностью и может возникнуть, наряду с другими аномалиями, только при использовании ослабленных критериев корректности (уровней изоляции).

Наиболее просто множественные версии встраиваются в протокол TS: для каждой операции чтения выбирается последняя версия элемента данных, метка записи которой предшествует метке читающей транзакции. При этом для каждого элемента данных может потребоваться неограниченно большое количество версий, однако нет необходимости хранить версии, записанные раньше, чем последняя версия, записанная непосредственно до начала самой старой активной транзакции.

Если эта версия является последней для элемента данных (т. е. не была записана активной транзакцией) и если она не была прочитана ни одной из активных транзакций, то ее метки времени можно не хранить. Нет необходимости в хранении меток последнего чтения для всех версий, кроме последней записанной, т. к. попытка изменения любой версии, кроме последней, приводит к обрыву транзакции.

Проверка сериализуемости в многоверсионном варианте протокола TS использует не только метки времени записи, но и метки времени последнего чтения: версия, прочитанная более поздней транзакцией, не может быть изменена более ранней, даже если это последняя записанная версия.

В многоверсионном протоколе SI необходимо иметь возможность определять, какая именно версия каждого элемента данных должна быть доступна транзакции (входит в ее снимок состояния базы данных), и предотвращать доступ транзакции к другим версиям. По определению это последняя версия, записанная транзакцией, зафиксированной на момент старта транзакции, для которой строится снимок. Соответственно, реализация может использовать либо метки времени, связываемые с каждой версией и указывающие время фиксации транзакции, которая эту версию записала, либо связывать с версией идентификатор записавшей транзакции и при чтении проверять список транзакций, которые были активны на момент старта читающей. В системе PostgreSQL применяется именно такой протокол и вторая из названных стратегий.

Поскольку в PostgreSQL любое обновление записывается на новое место, поддержка множественных версий основана просто на использовании устаревших

версий. Обычная версия протокола, предусматривающая обрыв транзакций, применяется для реализации уровня изоляции Repeatable Read, а для уровня изоляции Read Committed необходимости в обрывах транзакций нет, и они никогда не выполняются.

Несколько сложнее построить многоверсионные протоколы двухфазного блокирования, потому что порядок сериализации может не совпадать с порядком, в котором транзакции выполнялись. Простую реализацию допускает двухверсионный двухфазный протокол, в котором для любого элемента данных необходимо хранить не более двух версий: последнюю зафиксированную и версию, обновляемую транзакцией, установившей блокировку на обновление. Такой протокол, однако, сохраняет большую часть недостатков одноверсионного 2PL.

13.2.9. Блокировки или метки времени?

Основное назначение блокировок состоит в том, чтобы организовать выполнение операций разных транзакций в таком порядке, который обеспечивает выполнение некоторых условий корректности. Ту же функцию выполняют и метки времени, поэтому можно сказать, что блокировки и упорядочивание операций на основе меток времени в каком-то смысле взаимозаменяемы.

Отличие состоит в том, что при невозможности немедленного выполнения операций блокировки приводят к ожиданиям, а применение меток времени — к обрывам транзакций.

Большинство систем управления базами данных, применяемых в промышленности, использует блокировки для реализации протокола управления транзакциями. По всей видимости, это вызвано тем, что потери, связанные с обрывами транзакций, считаются более существенными, чем снижение пропускной способности вследствие ожиданий. Однако относительная важность этих метрик зависит от характеристик оборудования, размеров базы данных, размера транзакций и от требований к системе.

13.3. Ослабленные критерии корректности: уровни изоляции в SQL

В связи с тем, что ранние протоколы управления транзакциями (в частности, протокол S2PL) ограничивают возможности параллельного выполнения тран-

закций, сложилась практика применения ослабленных условий корректности. Некоторые из таких ослабленных условий формализованы в стандарте SQL-92, в котором были определены уровни изоляции, кратко рассмотренные в главе 6.

В отличие от формальной теории транзакций ослабленные критерии корректности определяются не в терминах состояний базы данных и возвращаемых значений (выраженных, например, семантиками Эрбрана), а в терминах запретов на определенные аномалии конкурентного выполнения. Этот способ определения страдает серьезным концептуальным недостатком: можно запретить все или какие-либо из известных аномалий, однако нельзя гарантировать, что в будущем не будут обнаружены новые. Так, аномалия только читающей транзакции была открыта спустя полтора десятилетия после появления уровней изоляции в стандарте SQL.

В соответствии со стандартом SQL реализация каждого уровня изоляции должна гарантировать отсутствие указанных аномалий, однако не обязательно должна допускать все остальные. В действительности реализация протоколов, которые обеспечивали бы в точности то, что требуется стандартом, практически невозможна, поэтому все СУБД фактически реализуют более строгие ограничения, чем буквально требуется стандартом. Подробный анализ уровней изоляции, определенных в стандарте, содержится в статье [1]. Дополнительная сложность теоретического анализа ослабленных уровней изоляции вызвана тем, что для отдельных транзакций, выполняющихся в системе, можно задавать разные уровни изоляции. Анализ смешанных уровней изоляции можно найти в [36].

Наименее ограничительный уровень Read Uncommitted вводился, для того чтобы обеспечить наиболее высокую производительность. Поскольку в системе PostgreSQL используется многоверсионный протокол управления транзакциями, доступ к незафиксированным версиям не дает никаких преимуществ по производительности. Поэтому в PostgreSQL этот уровень реализуется точно так же, как Read Committed, хотя указание Read Uncommitted допускается синтаксисом SQL. Подобные решения приняты разработчиками и других СУБД.

Уровень изоляции Read Committed принято рассматривать как разумный компромисс между требованиями согласованности и производительности. Этот уровень обычно устанавливается в PostgreSQL по умолчанию. Уровень Repeatable Read гарантирует идентичность результатов повторного чтения данных. Наконец, уровень Serializable требует сериализуемость по конфликтам.

Указание уровня изоляции возможно в PostgreSQL с помощью установки параметра конфигурации default_transaction_isolation для всего кластера баз данных

или для отдельных баз данных (с помощью команды ALTER DATABASE). Также уровень изоляции можно установить для сеанса (соединения с базой данных) командой SET SESSION CHARACTERISTICS или для отдельной транзакции командой SET TRANSACTION.

Следовательно, протокол управления транзакциями должен обеспечивать одновременное соблюдение требований изоляции разного уровня для разных транзакций, выполняемых в системе. В системе PostgreSQL для реализации всех уровней изоляции применяются блокировки на запись и многоверсионность.

Для транзакций, выполняемых с уровнем изоляции Read Committed, для операций чтения используются значения, зафиксированные другими транзакциями не позже, чем начало операции чтения. Для операций записи устанавливаются блокировки, которые снимаются при завершении транзакций. При невозможности установить блокировку транзакция переводится в состояние ожидания.

Для транзакций, выполняемых в режиме Repeatable Read, операции чтения возвращают значения, записанные до начала этой транзакции, и, кроме этого, проверяются условия протокола SI: если интервалы времени выполнения транзакций пересекаются и они обновляют одни и те же данные, то только одна из них может быть зафиксирована. Выполнение этого правила обеспечивается применением блокировок на запись. Так же как и для Read Committed, успешно установленные блокировки удерживаются до завершения транзакции. Если установка блокировки невозможна, то операция откладывается до завершения транзакции, препятствующей установке блокировки. Если препятствующая транзакция фиксируется, то ожидающая транзакция обрывается.

Наконец, если транзакция выполняется с уровнем изоляции Serializable, то в дополнение к тому, что делается для уровня Repeatable Read, выполняется проверка зависимостей WR и RW. Как отмечено выше в разделе 13.1.2, в случае если расписание получено протоколом SI, граф сериализуемости может содержать только такие контуры, в которые входят пары или тройки транзакций, связанных зависимостями специального вида. Отсутствие в графе сериализуемости этих конфигураций, следовательно, является достаточным условием сериализуемости. Поиск таких конфигураций выполняется значительно эффективнее, чем поиск контуров.

Заметим, что для реализации абстрактного протокола SI ожидание не требуется: можно было бы просто обрывать транзакцию при невозможности установки

блокировки. Однако применяемый в PostgreSQL протокол дает шанс транзакциям в том случае, если препятствующая транзакция завершается обрывом по другим причинам.

Заметим также, что протокол PostgreSQL несвободен от тупиков. Проверка на наличие тупиков повторяется через интервал времени, устанавливаемый параметром сервера deadlock_timeout.

13.4. Итоги главы

В этой главе рассмотрены теоретические критерии согласованности и протоколы, которые могут обеспечить их выполнение. Итоги более чем десятилетнего развития теории корректного выполнения транзакций собраны в книге [10], которую принято считать эталонным представлением этой теории. Развернутое формальное представление теории транзакций, дополненное новыми (на момент написания) идеями и подходами, содержится в [62]. Двухфазный протокол блокирования предложен в [59], там же введено понятие блокировки на предикат.

Анализ протокола SI содержится в [47], реализация сериализуемого варианта SI в системе PostgreSQL представлена исследовательскому сообществу в [51].

13.5. Упражнения

Упражнение 13.1. Запишите семантики Эрбрана для всех расписаний, встречающихся в этой главе.

Упражнение 13.2. Постройте расписание, которое может быть выполнено при использовании протокола TS, но не может быть порождено протоколом 2PL: Gen(TS) \ Gen(2PL).

Упражнение 13.3. Постройте расписание из Gen(2PL) \ Gen(TS).

Упражнение 13.4. Постройте расписание из Gen(SSI) \ Gen(2PL).

Упражнение 13.5. Докажите, что реализации Read Committed и Serializable в системе PostgreSQL различаются, выполняя две транзакции в разных сеансах psql.

Упражнение 13.6. Постройте несериализуемое расписание из Gen(SI) и покажите, что оно не допускается в системе PostgreSQL, если выбран уровень изоляции Serializable.

Упражнение 13.7. Покажите, что в режиме Repeatable Read допускаются аномалии несогласованной записи и только читающей транзакции.

Упражнение 13.8. Получите ситуацию тупика.

Упражнение 13.9. Докажите утверждение, сформулированное в разделе 13.2.3: $\text{Gen}(\text{S2PL}) \subset \text{CSR} \cap \text{ST}$.

Глава 14

Надежность баз данных

14.1. Восстановление после отказов

Одна из основных функций СУБД — обеспечение сохранности данных, при этом данные должны оставаться в корректном состоянии. Попробуем классифицировать все многообразие программных и аппаратных средств (приложений, операционных систем, вычислительных систем, информационных сетей, электропитания и т. д.) и причин, которые могут привести к потере данных (ошибки при разработке и эксплуатации, стихийные и техногенные бедствия, действия злоумышленников и пр.).

С точки зрения поддержки сохранности данных отказы систем можно разделить на следующие категории:

Отказы транзакций и приложений. По каким-либо причинам приложение не может завершить транзакцию, например пропала связь с сервером или невозможно выполнить требования протокола. В этом случае транзакция обрывается и выполняется ее откат с помощью операций обращения (w^{-1}), как описано в разделе 13.1.4. Напомним, что операция w^{-1} является чисто логической, ее реализации в разных системах могут отличаться. В частности, в PostgreSQL выполнение отката не требует записи данных, потому что новые значения всегда записываются на новое место, а старые, следовательно, сохраняются.

Отказы сервера. Сервер базы данных или вся вычислительная система утратили возможность выполнять запросы приложений, но содержимое базы данных, размещенное на энергонезависимых носителях (вращающихся дисках, SSD и т. п.), сохранилось. В этом случае выполняется рестарт сервера базы данных, возможно, после рестарта операционной системы. Во время рестарта сервера необходимо привести базу данных в согласованное состояние, в котором все изменения, сделанные зафиксированными транзакциями, сохранятся, а незавершенные к моменту отказа сервера

транзакции будут оборваны и для сделанных ими изменений будет выполнен откат.

Разрушение носителей. Для того чтобы предотвратить потерю данных при разрушении носителей, необходимо регулярно создавать резервные копии. Восстановление после разрушения носителя сводится к восстановлению базы данных с резервной копии, возможно, с последующим восстановлением согласованности, как при отказе системы.

Меры по обеспечению надежности, необходимые для каждой базы данных, зависят от требования к прикладной системе. Чем выше уровень требований, тем более сложные (и дорогостоящие) необходимы решения. Любая конфигурация системы управления базами данных обеспечивает корректность откатов транзакций и восстановление после отказов сервера, однако выбор мер защиты от разрушения носителя может варьироваться в очень широком диапазоне.

14.2. Отказы сервера баз данных

14.2.1. Журнал транзакций

Для того чтобы обеспечить обработку отказов транзакций и отказов сервера, система управления базами данных регистрирует все изменения, выполняемые транзакциями, в *журнале транзакций*.

Записи заносятся в журнал строго последовательно, т. е. новые записи всегда добавляются в конец журнала. Каждая запись снабжается уникальным идентификатором LSN (log sequence number), который однозначно ее идентифицирует. В системе PostgreSQL для этого используется тип `pg_lsn`, представляющий собой 8-байтовое число, указывающее смещение записи от начала журнала. В других системах может использоваться другой формат LSN, например старшие 4 байта могут указывать номер файла журнала, а младшие — смещение в этом файле. Важным требованием к LSN, которое необходимо для корректной работы алгоритма восстановления и поэтому соблюдается во всех системах, является строгая упорядоченность: записи журнала, созданные позже, должны иметь большие значения LSN, чтобы, сравнивая номера LSN двух записей, всегда можно было установить, какая из них появилась раньше.

Переключение на новый файл журнала происходит, когда размер журнала превышает определенный порог, при создании резервной копии базы данных или по другим причинам, в том числе по указанию администратора базы данных.

Как и для любых других типов файлов, несколько последних записей журнала могут находиться только в оперативной памяти, пока не произойдет их перенос на энергонезависимый носитель информации. Далее будем условно называть такой носитель диском. Заметим, что при нормальной работе системы записи журнала не используются, поэтому нет необходимости сохранять в оперативной памяти записи, которые уже перенесены (*вытолкнуты*) на диск.

Ведение журнала транзакций подчиняется следующим правилам *опережающей записи* (write-ahead logging, WAL):

WAL1 записи, регистрирующие любые изменения в базе данных, должны быть занесены в журнал и вытолкнуты на диск, до того как сами эти изменения (в базе данных) попадут на диск;

WAL2 информация о фиксации транзакций в журнале должна быть вытолкнута на диск раньше, чем завершится операция фиксации, и раньше, чем приложение получит информацию о том, что фиксация выполнена успешно.

В журнал могут заноситься записи нескольких разных типов, в том числе:

- BEGIN отмечает начало транзакции (в PostgreSQL такая запись не применяется);
- COMMIT регистрирует фиксацию транзакции;
- ROLLBACK отмечает откат транзакции (такая запись необходима в системе PostgreSQL, но может быть ненужной в других системах, в которых применяются явные операции отката w^{-1});
- UNDO содержит информацию о том, как выполнить откат операции модификации базы данных (не применяется в PostgreSQL);
- REDO содержит информацию, достаточную для повторного выполнения операции модификации данных;
- CHECKPOINT содержит дополнительную информацию для восстановления базы данных при рестарте.

Существует еще несколько типов записей журнала, например в некоторых СУБД в журнале отмечается создание резервных копий базы данных (для обеспечения восстановления носителя данных). В системе PostgreSQL в журнале регистрируется создание файлов и каталогов, а также другие вспомогательные действия.

Записи о модификации данных могут быть *логическими*, связанными с выполняемыми операциями SQL (INSERT, UPDATE, DELETE), или *физическими*, отражающими состояние измененных страниц базы данных. В любом случае для каждой операции модификации могут быть созданы две записи: UNDO и REDO. Это дает возможность как устранить результаты выполнения операции, так и выполнить ее повторно.

Существует несколько стратегий записи в журнал, гарантирующих возможность восстановления согласованного состояния базы данных. Эти стратегии обеспечивают выполнение требований атомарности и долговечности транзакций: завершенные транзакции не могут быть потеряны, а оборванные (или не завершенные до отказа) транзакции не должны оставлять изменений в базе данных. Для выполнения этих требований, необходимо чтобы информация об изменениях, которые еще не записаны на диск базы данных, обязательно сохранялась в журнале и при этом попадала на энергонезависимый носитель данных.

Выбор стратегии ведения журнала определяется следующими свойствами:

FORCE / NO FORCE. Свойство FORCE означает, что все изменения, выполненные транзакцией, заносятся в базу данных и выталкиваются на диск, до того как завершается выполнение операции COMMIT. В этом случае нет необходимости в записях REDO.

Свойство NO FORCE, наоборот, предполагает, что записи REDO выталкиваются из журнала на диск до завершения фиксации, но не требуют записи изменений в базу данных.

STEAL / NO STEAL. Свойство STEAL означает, что страницы (блоки) базы данных, содержащие изменения незафиксированных транзакций, могут выталкиваться на диск. В этом случае необходимы журнальные записи UNDO, которые должны выталкиваться на диск раньше, чем страницы базы данных, содержащие эти изменения.

Свойство NO STEAL запрещает выталкивание на диск страниц, содержащих изменения незавершенных транзакций. В этому случае записи UNDO

не требуются, потому что на диск попадают только изменения зафиксированных транзакций.

Комбинация этих свойств дает четыре возможных стратегии ведения журнала.

FORCE + NO STEAL. Эта стратегия требует, чтобы все изменения записывались в базу данных в момент фиксации. Для ее реализации используется метод *теневых страниц*, состоящий в том, что измененные транзакцией страницы записываются на новое место, а при фиксации происходит обновление только одной страницы, содержащей указатели на актуальные версии данных. Для восстановления не требуются записи ни REDO, ни UNDO, поэтому при восстановлении после системных отказов журнал не нужен. Эта стратегия оказывается менее эффективной при нормальной работе (т. е. создает большую дополнительную нагрузку, чем остальные) и в высокопроизводительных системах не применяется.

FORCE + STEAL. Поскольку все изменения должны быть записаны на диск базы данных до фиксации, могут возникать задержки при фиксации транзакций, выполнивших относительно большое количество изменений. Для этой стратегии не требуются записи REDO.

NO FORCE + NO STEAL. Изменения записываются на диск базы данных только после фиксации транзакций. При этом некоторые страницы могут слишком долго оставаться в оперативной памяти (например, страницы, часто обновляемые различными транзакциями). В этом случае записи UNDO не требуются.

NO FORCE + STEAL. Эта стратегия дает возможности для полностью асинхронной записи изменений базы данных на диск, никак не связанной с фиксацией транзакций. Обычно именно эта стратегия используется высокопроизводительными системами, в том числе PostgreSQL.

Занесение в журнал двух записей на каждое изменение может создавать значительную дополнительную нагрузку на сервер, однако обычно эта нагрузка не ведет к снижению производительности, потому что последовательная запись в журнал почти для всех носителей данных выполняется быстрее, чем изменение в произвольном порядке (для вращающихся дисков она может быть быстрее на два порядка). Кроме этого, при наличии журнала запись изменений на диски базы данных выполняется отдельным фоновым процессом, т. е. не замедляет выполнение транзакций. Благодаря опережающей записи в журнал информация об изменениях не будет потеряна, даже если возникнет необходимость в рестарте сервера баз данных.

14.2.2. Рестарт сервера

После рестарта сервера база данных может оказаться в несогласованном состоянии. Во-первых, активные транзакции, которые не успели зафиксироваться, до того как произошел отказ системы, должны быть оборваны, и поэтому необходимо выполнить откат тех изменений, которые уже были занесены в базу данных. Во-вторых, изменения транзакций, которые были зафиксированы, могли не попасть в базу данных (остаться только в оперативной памяти и в журнале). Для того чтобы привести базу данных в согласованное состояние, при рестарте запускается алгоритм восстановления.

Известно несколько различных алгоритмов восстановления. Кратко опишем алгоритм «Redo history», выполнение которого включает две фазы.

Анализ и повторное выполнение. На фазе анализа выполняется анализ журнала и повторно выполняются изменения операций всех транзакций, которые еще не были занесены в базу данных. Для этого выполняется просмотр журнала в прямом направлении (от начала к концу) и выполняются следующие действия:

- При обнаружении записи BEGIN транзакция записывается в список активных транзакций. Если записи BEGIN не используются, то началом транзакции является первая операция, помеченная идентификатором транзакции, который не встречался ранее.
- Появление записи COMMIT приводит к исключению транзакции из списка активных.
- Записи REDO используются для повторения операций, если соответствующие изменения еще не внесены в базу данных. Для того чтобы определить, какие изменения занесены на страницу, на каждой странице имеется поле PSN (page sequence number), содержащее LSN последней записи журнала, изменения которой уже есть на странице. Если LSN > PSN, то изменения заносятся на страницу и изменяется ее PSN.
- Если изменение относится к активной транзакции, то изменяемый объект заносится в список объектов, измененных этой транзакцией. Этот список объектов не обязателен, но его наличие позволяет ускорить выполнение фазы отката (например, загрузить все необходимые для отката страницы в память перед выполнением этой фазы).

После завершения просмотра журнала все изменения, выполненные до отказа системы и зарегистрированные в журнале, будут занесены в базу данных.

Откат. На фазе отката выполняется просмотр журнала в обратном направлении (от конца к началу), и для всех операций из списка активных транзакций, полученного на первой фазе, выполняется операция отката с помощью записи UNDO. Операция отката регистрируется в журнале, т. е. для нее заносятся записи REDO или UNDO (если такие записи необходимы для восстановления в соответствии со стратегией ведения журнала, принятой в системе).

Несмотря на то что в системе PostgreSQL не используются записи UNDO, этот шаг все равно выполним, поскольку необходимые значения имеются в базе данных.

При обнаружении записи BEGIN для активной транзакции эта транзакция исключается из списка активных и выполняется ее фиксация, т. е. в журнал заносится запись COMMIT, и журнал выталкивается на диск.

Работа алгоритма заканчивается, когда список активных транзакций становится пустым.

Конечно, фаза отката не нужна в системах, использующих стратегию NO STEAL, потому что в таких системах результаты выполнения транзакций не могут попасть в устойчивую память до фиксации транзакции.

Одним из требований к алгоритмам восстановления после рестарта является возможность многократного выполнения (в случае отказа системы во время процедуры восстановления). Для того чтобы выполнить это требование, необходимо гарантировать, что повторное выполнение операции REDO не разрушает содержимое страницы.

Существует два вида записей REDO:

- Запись содержит полный образ страницы.
  В этом случае повторное выполнение операции записи дает такой же результат, как однократное.
- Запись содержит только информацию об изменениях.
  Однократность применения таких записей обеспечивается проверкой значения поля PSN на странице.

Основным критерием качества алгоритмов восстановления является время, необходимое для возобновления нормальной работы СУБД.

Для того чтобы сократить время недоступности, можно использовать список страниц, обновленных незавершенными активными транзакциями, построенный на первой фазе работы алгоритма восстановления. После окончания первой фазы доступ к страницам из этого списка (или к модифицированным элементам данных) блокируется. Это дает возможность начать обработку новых транзакций сразу после окончания первой фазы. По мере выполнения отката незавершенных транзакций блокировки снимаются, открывая доступ к восстановленным состояниям элементов данных.

14.2.3. Контрольные точки

Механизм *контрольных точек* предназначен для сокращения объема журнала, который просматривается при восстановлении. Журнальная запись контрольной точки CHECKPOINT содержит список активных транзакций и список страниц, состояние которых в оперативной памяти отличается от состояния на постоянном носителе. После выталкивания записи о контрольной точки фоновый процесс записи копирует все изменения из этого списка на диск. При этом нормальная работа системы продолжается. Если при этом страницы, включенные в список, будут изменены новыми транзакциями, эти изменения попадут на диск. Изменения на страницах, не включенных в список, будут учтены в следующей контрольной точке.

Включение списка активных транзакций, вообще говоря, не обязательно, но позволяет упростить работу алгоритма восстановления на фазе анализа: список, содержащийся в записи журнала, используется в качестве начального значения для списка активных транзакций.

Когда копирование страниц, включенных в контрольную точку, заканчивается, в журнал заносится запись о завершении контрольной точки. После этого (сразу или через некоторое время) может быть создана новая контрольная точка и работа системы продолжается. Наличие записей контрольной точки позволяет при рестарте сервера на фазе повторного выполнения начать просмотр журнала не с самого начала, а с предпоследней контрольной точки. При этом начальное состояние списков, которые строятся на первой фазе, считывается из записи о контрольной точке.

Заметим, что запись начала контрольной точки избыточна и нужна только для упрощения процесса анализа при восстановлении. В системе PostgreSQL начало контрольных точек в журнале не отмечается.

В системе PostgreSQL первая после контрольной точки запись REDO, относящаяся к любой странице, содержит полный образ этой страницы (при установленном параметре конфигурации full_page_writes). Это дает возможность корректно восстанавливать содержимое страниц, которые могли быть повреждены при отказе системы. Такие повреждения возможны, потому что в реальности операция записи страницы на диск не атомарна.

Необходимо обратить внимание на то, что запись изменений, внесенных транзакциями, выполняется абсолютно асинхронно и никак не связана с фиксацией транзакций. Вследствие этого можно считать, что операции изменения записей, ранее прочитанных приложением и, скорее всего, оставшихся в кеше, не требуют времени больше, чем необходимо для выполнения этих изменений в оперативной памяти, в отличие от операций чтения, которые могут ожидать считывания необходимых данных с диска. Поведение, которое при этом наблюдают приложения, может показаться парадоксальным: операции записи выполняются на порядки быстрее, чем операции чтения. Конечно, операции записи требуют определенных вычислительных ресурсов и дают свой вклад в общую нагрузку системы, однако эти операции выполняются асинхронно и, как правило, не оказывают существенного влияния на обработку запросов.

14.3. Разрушение носителя

Если по каким-либо причинам файлы базы данных или журналы не могут быть прочитаны, то восстановление после рестарта невозможно. Такая ситуация рассматривается как разрушение носителя (не имеет значения, разрушен ли сам носитель или только логическая структура данных).

Для того чтобы предотвратить потерю данных в случае разрушения носителя, необходимо периодическое создание резервных копий. Существует большое разнообразие методов создания резервных копий, различающихся по сложности, полноте восстановления и по стоимости.

Простые методы создания копий обеспечивают восстановление базы данных в состояние, которое было на момент создания резервной копии. При этом все изменения, выполненные после создания копии до отказа, будут потеряны.

Более сложные методы предполагают (в дополнение к копированию самой базы данных) хранение журнала транзакций. Для приведения системы в рабочее состояние после разрушения носителя необходимо восстановить базу данных с последней резервной копии и затем выполнить рестарт системы с использованием всех файлов журнала, записанных после создания этой резервной копии (а не с контрольной точки, как при отказе сервера).

Заметим, что для реализации этого процесса необходимы все файлы журнала вплоть до момента отказа с разрушением носителя, в том числе записанные после создания резервной копии. Для того чтобы такие файлы журнала были доступны, в системах, где необходима высокая надежность, при нормальной работе системы записывается несколько идентичных копий журнала на разные носители. При этом процесс восстановления может занимать значительное время.

Наиболее сложные и дорогостоящие методы предполагают поддержку дополнительных серверов (*реплик*) баз данных, готовых или почти готовых к работе. Такие схемы обеспечивают быстрое восстановление при любых отказах, но создают значительную дополнительную нагрузку во время нормальной работы системы.

Основными метриками, характеризующими качество процедур восстановления, являются следующие:

Время восстановления. Измеряется от начала процедуры восстановления до начала обработки новых транзакций. Время восстановления может варьироваться от нескольких часов для простых методов до долей секунды для сложных и непосредственно влияет на характеристику доступности системы.

Выживаемость. Обозначает количество разрушений носителя, которое приводит к потере данных. Эта характеристика не обязательно совпадает с количеством копий, т. к. процедура восстановления может включать создание новой резервной копии, что делает ее более дорогостоящей и более продолжительной, но обеспечивает более высокую выживаемость.

Задержка. Время, необходимое для распространения изменений по резервным копиям, обеспечивающим выживаемость. Для простых методов, не использующих журнал, задержка равна интервалу между созданиями резервных копий. В высоконадежных системах задержка измеряется долями секунды.

Методы создания резервных копий и восстановления с них выбираются в зависимости от требований к системе. Важно учитывать, что сложность и стоимость реализации и сопровождения очень существенно зависит от этих требований.

14.3.1. Экспорт и импорт

Наиболее простой способ создания резервных копий — экспорт логической структуры базы данных в какой-либо внешний формат. В системе PostgreSQL такой экспорт можно выполнить с помощью утилиты pg_dump. К достоинствам этого метода можно отнести простоту, возможность частичного восстановления базы данных, а также возможность восстановления в другой конфигурации сервера или даже в другой СУБД (в последних двух случаях решаются другие задачи, а не задача восстановления после разрушения носителя).

Размещение данных после восстановления экспортированной базы данных, скорее всего, не будет совпадать с размещением в исходной БД, поэтому восстановление актуального состояния по журналу в этом случае невозможно. Это означает, что база данных восстанавливается в то состояние, в котором она была в момент начала копирования, и, следовательно, задержка может достигать значений, равных интервалу времени между созданиями резервных копий.

Во многих случаях такие значения задержки допустимы — например, для баз данных, используемых только для разработки или тестирования приложений, когда система не находится в производственной эксплуатации или приложение является тиражируемым продуктом. В подобных случаях нагрузка на сервер базы данных относительно невелика, вероятность разрушения носителя достаточно мала, и ценность данных и особенно изменений, сделанных после создания копии, незначительна. Применение более сложных стратегий резервирования и восстановления имеет смысл только для отработки и проверки самих процедур резервирования и восстановления.

14.3.2. Копирование с восстановлением по журналам

Если допустимые значения задержки не должны превышать долей секунды, то методы на основе экспорта и импорта оказываются непригодными. Традиционная стратегия восстановления состоит в том, что в качестве резервной копии создается точный образ базы данных, зачастую в формате, непригодном для непосредственного запуска сервера баз данных (возможно, в сжатом виде).

Кроме этого, необходимо сохранять все данные, записываемые при нормальной работе системы в журнал транзакций.

В системе PostgreSQL для записи журнала используется несколько файлов, называемых *сегментами* журнала. При переключении на новый сегмент самый старый из ранее заполненных сегментов уничтожается, как только будет записана контрольная точка, после которой данные из этого сегмента станут ненужными для восстановления после отказа системы.

Для того чтобы использовать резервные копии, необходимо выполнить процедуру архивирования сегментов журнала, до того как они будут удалены. В системе PostgreSQL это делается с помощью параметров конфигурации и задания команды операционной системы, которая будет выполнять архивирование (это может быть просто копирование на другой носитель).

Для того чтобы восстановление было возможно, необходимо периодически создавать резервную копию базы данных, а в промежутках между созданиями копии архивировать все сегменты журнала, порождаемые в результате нормальной работы системы.

Существует два метода создания резервной базы данных:

- Применение программ создания резервных копий, входящих в состав СУБД. Для создания такой копии не требуется останавливать нормальную работу сервера базы данных.
- Копирование файлов базы данных (включая журнал транзакций) средствами операционной системы. Такое копирование может в некоторых системах работать быстрее, но для некоторых СУБД может требоваться остановка сервера баз данных, что, конечно, влияет на доступность системы.

В системе PostgreSQL такие копии кластера баз данных создаются с помощью утилиты pg_basebackup. Утилита работает как обычный клиент сервера баз данных (в некоторых случаях она устанавливает не одно, а два соединения с сервером). Поэтому во время создания копий нормальная работа сервера баз данных не останавливается, хотя копирование создает дополнительную нагрузку на сервер.

Вместо утилиты pg_basebackup можно использовать вызовы системных функций (например, через оператор SELECT). Это дает возможность более гибкого управления деталями создания резервной копии.

Если копия кластера баз данных записывается в архивный файл, то для использования такой копии необходимо восстановить ее на носителе и затем выполнить процедуру рестарта сервера. В результате база данных будет восстановлена в состояние, в котором она была во время создания резервной копии. Далее необходимо повторно выполнить изменения, зарегистрированные в сегментах журнала, накопленных после создания этой копии, как архивированных, так и текущих.

Периодичность создания резервной копии зависит от того, сколько времени допустимо потратить на восстановление. Более частое создание резервных копий сокращает количество сегментов журнала, которые необходимо применить, но увеличивает нагрузку на сервер при нормальной работе системы.

Для обеспечения большей выживаемости можно создавать несколько копий параллельно, однако более эффективным методом считается копирование резервной копии средствами операционной системы после того, как она создана, что может повысить эффективность использования оборудования.

Поскольку процедура восстановления предусматривает повторное внесение изменений по журналу в том порядке, в котором эти изменения выполнялись, возможно восстановление не только последнего согласованного состояния баз данных до момента отказа, но и состояния на любой предшествующий момент времени (после завершения записи резервной копии). Такая возможность полезна в том случае, когда необходимость восстановления вызвана, например, некорректностью работы приложений или ошибками персонала.

В системе PostgreSQL предусмотрено завершение процесса восстановления по одному из следующих критериев:

- достигнуто согласованное состояние базы данных;
- достигнут явно указанный момент времени;
- выполнена транзакция с указанным идентификатором;
- достигнута заранее созданная именованная точка восстановления.

Создание резервной копии сопровождается выполнением контрольной точки и переключением файлов (сегментов) журнала. Для корректного восстановления базы данных из резервной копии необходима информация о двух позициях в журнале:

1) перед началом резервного копирования — с этой позиции журнал просматривается при восстановлении;

2) после окончания резервного копирования — достижение этой позиции гарантирует восстановление того состояния базы данных, в котором она была на момент завершения создания резервной копии.

В некоторых системах указанные позиции отмечаются специальными записями непосредственно в журнале. В системе PostgreSQL они записываются в отдельный файл, также включаемый в состав резервной копии.

14.3.3. Резервные серверы баз данных

Недостатком стратегии копирования и восстановления на основе копий в последовательных файлах является большое время восстановления (несколько часов для больших баз данных), а достоинством — относительно небольшое количество ресурсов, необходимых для хранения резервных копий в сжатом виде. Очевидно, что большое время восстановления отрицательно влияет на характеристику доступности, которая считается крайне важной для некоторых классов приложений.

Для того чтобы исключить время копирования всей базы данных, резервные копии создаются в обычном формате базы данных и на этой копии запускается резервный сервер баз данных. При этом возможны различные варианты распространения на резервные серверы изменений, сделанных на основном сервере. Создание и ведение таких серверов является одним из применений репликации баз данных, детально рассматриваемой в главе 21.

Применение резервных серверов позволяет сократить время восстановления до десятков или даже единиц секунд, что обеспечивает очень высокие значения характеристики доступности, однако чем меньше требуемое время восстановления, тем большая дополнительная нагрузка ложится на систему при ее нормальной работе.

Для применения любой схемы защиты от разрушений носителя с запасным сервером требуется не менее чем вдвое большая конфигурация оборудования (как минимум два сервера, каждый из которых обладает производительностью, достаточной для выполнения запросов прикладной системы в нормальном режиме работы). Имеется возможность несколько уменьшить избыточность оборудования, используя предусмотренную в СУБД PostgreSQL и других высокопроизводительных системах возможность выполнения на запасном сервере запросов на чтение (но не на модификацию) данных. Запасные серверы часто используются для извлечения данных с целью загрузки в хранилище данных

(data warehouse) или для получения аналитических отчетов непосредственно из базы данных. В том и другом случае обычно небольшое отставание состояния базы данных от актуального не имеет значения.

14.4. Итоги главы

В этой главе рассмотрены методы и алгоритмы обеспечения надежности хранения данных. Результативность этих методов выбирается в зависимости от требований к информационной системе, использующей базу данных. Более сложные (и одновременно более дорогостоящие в эксплуатации) методы позволяют получить очень высокие значения характеристик надежности и доступности.

14.5. Упражнения

Упражнение 14.1. Создайте копию демонстрационной базы данных утилитой pg_dump, выполните несколько обновляющих транзакций. Уничтожьте базу данных и восстановите ее содержимое, используя резервную копию. Объясните результаты.

Упражнение 14.2. Включите архивирование журнала базы данных и убедитесь в том, что оно работает.

Упражнение 14.3. Создайте резервную копию кластера баз данных утилитой pg_basebackup в виде tar-файла. Выполните несколько обновляющих транзакций, создайте точку восстановления и выполните еще несколько транзакций. Уничтожьте кластер и восстановите его с резервной копии по состоянию на точку восстановления.

Упражнение 14.4. Создайте резервную копию кластера баз данных, разместив ее в другой файловой системе. Запустите новый сервер баз данных, работающий с этой копией.

Глава 15

Дополнительные возможности SQL

15.1. Дополнительные средства SQL

15.1.1. Общие табличные выражения

Одним из наиболее важных требований к высокоуровневым языкам программирования является возможность выделения логически автономных частей алгоритмов в виде процедур или функций. Аналогом такого структурирования в декларативном языке запросов SQL можно считать подзапросы. Однако в ранних версиях SQL общие части кода могли быть либо выделены в отдельный объект — представление (что не всегда целесообразно), либо код общих выражений дублировался всюду, где они использовались. Начиная с версии стандарта SQL:1999, введено понятие *общего табличного выражения* (common table expression, CTE). Такое выражение представляет собой именованный запрос, который определяется предложением WITH и может использоваться в запросе в любом месте, где может находиться отношение или табличное выражение. Например, запрос

```
demo=# SELECT count(*)
FROM flights f
    JOIN aircrafts a ON f.aircraft_code = a.aircraft_code
WHERE a.range > 7000;
```

можно переписать с использованием CTE в следующем виде:

```
demo=# WITH long_range AS (
    SELECT * FROM aircrafts WHERE range > 7000
)
SELECT count(*)
FROM flights f
    JOIN long_range a ON f.aircraft_code = a.aircraft_code;
```

Конечно, для таких несложных запросов использование CTE не имеет особого смысла. Тем не менее в подзапросе long_range определено понятие, не выделенное в схеме демонстрационной базы данных: самолеты, способные совершать

дальние перелеты. В более сложных случаях выделение подзапросов может существенно улучшить читаемость всего запроса.

В соответствии со стандартом SQL запросы, содержащие CTE, должны выполняться, как если бы каждое CTE было вычислено один раз. В системе PostgreSQL это было реализовано буквально: все CTE выполняются как отдельные запросы, результат *материализуется* (записывается) во временную память и затем используется при выполнении всего запроса, содержащего CTE. Заметим, что в подзапросе CTE могут встречаться операторы обновления (UPDATE, INSERT, DELETE), содержащие предложение RETURNING, а также определенные пользователем функции, обладающие побочным эффектом. Подобные CTE действительно должны выполняться один раз, иначе результаты будут некорректными.

В новых версиях PostgreSQL способ обработки CTE другой. В тех случаях, когда это возможно, CTE не вычисляется отдельно, а подставляется вместо своего имени везде, где он используется в основном запросе. Такую подстановку нельзя делать, если в CTE выполняется модификация базы данных или встречаются функции, которые могут делать такую модификацию. Кроме этого, в запросе можно явным образом указывать, требуется ли материализация CTE. Такое указание может быть необходимо, для того чтобы сохранить планы выполнения запросов, работающих в уже существующих системах.

В оставшейся части раздела обсуждается поведение CTE при использовании материализации.

Материализация, безусловно, гарантирует корректность выполнения любого запроса, однако может приводить к потере эффективности. Для простого запроса, приведенного выше, планы выполнения с CTE и без него отличаются не очень существенно. План выполнения запроса для варианта без CTE выглядит следующим образом:

```
demo=# EXPLAIN (costs off)
SELECT count(*)
FROM flights f
    JOIN aircrafts a ON f.aircraft_code = a.aircraft_code
WHERE a.range > 7000;
                        QUERY PLAN
-----------------------------------------------------------
 Aggregate
   ->  Hash Join
         Hash Cond: (f.aircraft_code = ml.aircraft_code)
         ->  Seq Scan on flights f
         ->  Hash
               ->  Seq Scan on aircrafts_data ml
                     Filter: (range > 7000)
```

В варианте с использованием CTE подзапрос вычисляется отдельно, однако, по существу, выполняются те же операции (фильтрация и соединение на основе хеширования):

```
demo=# EXPLAIN (costs off)
WITH long_range AS (
    SELECT * FROM aircrafts WHERE range > 7000
)
SELECT count(*)
FROM flights f
    JOIN long_range a ON f.aircraft_code = a.aircraft_code;
                        QUERY PLAN
-----------------------------------------------------------
 Aggregate
   CTE long_range
     ->  Seq Scan on aircrafts_data ml
           Filter: (range > 7000)
   ->  Hash Join
         Hash Cond: (f.aircraft_code = a.aircraft_code)
         ->  Seq Scan on flights f
         ->  Hash
               ->  CTE Scan on long_range a
```

Неэффективность выполнения запроса появляется, в частности, в тех случаях, когда результат вычисления CTE имеет относительно большие размеры, но при его использовании в запросе выполняется фильтрация, исключающая значительную часть данных. Очевидно, что в таких случаях выполнение фильтрации до завершения вычисления подзапроса, определяющего CTE, может сократить время выполнения запроса.

Рассмотрим пример:

```
demo=# EXPLAIN (costs off)
WITH pass_info AS (
    SELECT t.book_ref,
           f.flight_no,
           f.departure_airport dep,
           f.arrival_airport arr,
           t.passenger_name,
           f.scheduled_departure
    FROM tickets t
        JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
        JOIN flights f ON f.flight_id = tf.flight_id
)
SELECT *
FROM pass_info
WHERE book_ref = 'A55664'
ORDER BY passenger_name,
         scheduled_departure;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Sort
   Sort Key: pass_info.passenger_name, pass_info.scheduled_departure
   CTE pass_info
     ->  Hash Join
           Hash Cond: (tf.flight_id = f.flight_id)
           ->  Hash Join
                 Hash Cond: (tf.ticket_no = t.ticket_no)
                 ->  Seq Scan on ticket_flights tf
                 ->  Hash
                       ->  Seq Scan on tickets t
           ->  Hash
                 ->  Seq Scan on flights f
   ->  CTE Scan on pass_info
         Filter: (book_ref = 'A55664'::bpchar)
```

В этом запросе в CTE вычисляется результат соединения таблиц относительно большого размера, поэтому общее время выполнения запроса оказывается большим. При фильтрации никакие индексы не используются.

После переноса в основной запрос выражения, определяющего CTE, получается план, использующий индексы:

```
demo=# EXPLAIN (costs off)
SELECT *
FROM (
    SELECT t.book_ref,
           f.flight_no,
           f.departure_airport dep,
           f.arrival_airport arr,
           t.passenger_name,
           f.scheduled_departure
    FROM tickets t
        JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
        JOIN flights f ON f.flight_id = tf.flight_id
) pass_info
WHERE book_ref = 'A55664'
ORDER BY passenger_name, scheduled_departure;
                               QUERY PLAN
-------------------------------------------------------------------------
 Sort
   Sort Key: t.passenger_name, f.scheduled_departure
   ->  Nested Loop
         ->  Nested Loop
               ->  Index Scan using tickets_book_ref_idx on tickets t
                     Index Cond: (book_ref = 'A55664'::bpchar)
               ->  Index Only Scan using ticket_flights_pkey on
                   ticket_flights tf
                     Index Cond: (ticket_no = t.ticket_no)
         ->  Index Scan using flights_pkey on flights f
               Index Cond: (flight_id = tf.flight_id)
```

Время выполнения в этом случае уменьшается на несколько порядков.

С другой стороны, CTE не только выполняются, но и оптимизируются отдельно от основного запроса, поэтому как основной запрос, так и подзапросы оказываются более простыми. Следовательно, возможны случаи, в которых оптимизатор будет применять точные алгоритмы для запросов, содержащих CTE, но будет вынужден применять приближенные алгоритмы для эквивалентного запроса, не содержащего CTE. В таких случаях материализация CTE может быть полезна (если, конечно, условия фильтрации в него включены).

Все эти особенности необходимо учитывать при проектировании запросов.

15.1.2. Рекурсивные запросы

Напомним, что функция или процедура, написанная на некотором языке программирования, называется *рекурсивной*, если во время выполнения она вызывает сама себя. Применительно к языкам запросов различные варианты рекурсии исследовались в 80-х и 90-х гг. в контексте дедуктивных баз данных. Особенностью таких СУБД является то, что наряду с хранимыми таблицами, содержание которых называется *экстенсионалом* базы данных, при вычислениях могут использоваться результаты выполнения (в том числе рекурсивного) запросов. Такие данные принято называть *интенсионалом* базы данных. Хорошее изложение теории рекурсивных запросов содержится в [61], алгоритмы вычисления рекурсивных запросов представлены, например, в [3].

Одним из важных результатов этих исследований стало доказательство того, что рекурсивные запросы не могут быть выражены в рамках реляционных языков. Мы не будем приводить здесь доказательство, однако поясним на простом примере. Допустим, что иерархия представлена в базе данных таблицей, содержащей два атрибута, значениями которых является некоторая вершина дерева и непосредственно подчиненная ей вершина следующего уровня. Таким образом, каждая строка этой таблицы представляет одно ребро в дереве, которое можно рассматривать как путь длины 1. Легко написать реляционное выражение, вычисляющее пути длины 2: это просто соединение таблицы с ней самой. Можно также построить выражения, вырабатывающие пути любой заданной длины, однако невозможно построить выражение реляционной алгебры, которое вырабатывало бы пути неограниченной длины.

С другой стороны, операция соединения этой таблицы с запросом, вырабатывающим пути длины не больше чем n, дает в результате все пути длины не более

чем $n + 1$. Поэтому рекурсивный запрос может справиться с задачей перечисления всех путей в дереве, начинающихся от корня или от любой другой вершины вниз по иерархии.

Существует другая интерпретация задачи о нахождении путей в дереве. Исходную таблицу можно рассматривать как бинарное отношение. Это отношение не обладает свойством транзитивности (из того, что некоторые пары (a, b) и (b, c) принадлежат отношению, не следует, что пара (a, c) тоже принадлежит этому отношению). Можно, однако, добавить все такие пары в отношение, и тогда свойство транзитивности будет иметь место. Минимальное (в смысле вложенности множеств) отношение, содержащее исходное отношение и обладающее свойством транзитивности, называется *транзитивным замыканием* исходного отношения.

Легко доказать, что для любого бинарного отношения транзитивное замыкание существует и единственно. Для доказательства существования заметим, что множество всех возможных пар транзитивно замкнуто. Единственность следует из того, что пересечение любых транзитивно замкнутых отношений также транзитивно замкнуто. Следовательно, если бы нашлись два минимальных транзитивно замкнутых отношения, содержащих исходное, то их пересечение также было бы транзитивно замкнуто, содержало бы исходное и содержалось бы в каждом из них, что несовместимо с их минимальностью.

Применение рекурсии наиболее общего вида приводит к тому, что результат вычислений становится зависимым не только от рекурсивных правил вывода, но и от вычислительного алгоритма [61]. Чтобы исключить это нежелательное явление, в языках запросов ограничивается применение рекурсивных правил. Неформально этот класс характеризуется тем, что применение правил к частично вычисленному результату может только добавлять в него новые элементы, но не удалять уже имеющиеся в нем.

Вычисление транзитивного замыкания, так же как и нахождение путей в дереве, нельзя выразить в реляционном исчислении. Однако замечательный факт состоит в том, что после добавления операции транзитивного замыкания к реляционной алгебре все рекурсивные запросы из указанного класса можно выразить в расширенной алгебре.

В языке SQL рекурсия выражается с помощью общих табличных выражений (CTE). В качестве иллюстрации покажем, как на языке SQL записать вычисление транзитивного замыкания. В реализации рекурсивных CTE в системе PostgreSQL требуется явно указывать ключевое слово `RECURSIVE`. В демонстрационной базе данных с помощью рекурсивных запросов можно находить

маршруты полетов с неограниченным количеством пересадок, однако такой запрос был бы слишком громоздким для иллюстрации. Поэтому создадим небольшую таблицу:

```
demo=# CREATE TABLE parent_child (parent integer, child integer);
CREATE TABLE
demo=# INSERT INTO parent_child
    VALUES (1, 2), (1, 3), (2, 4), (2, 5), (4, 6), (4, 7);
INSERT 0 6
```

Транзитивное замыкание вычисляется следующим запросом:

```
demo=# WITH RECURSIVE trans_closure (ancestor, descendant, level)
AS (
    SELECT pc.parent, pc.child, 1
    FROM parent_child pc
    UNION ALL
    SELECT pc.parent, ts.descendant, ts.level+1
    FROM parent_child pc
        JOIN trans_closure ts ON pc.child = ts.ancestor
)
SELECT *
FROM trans_closure;
 ancestor | descendant | level
----------+------------+-------
        1 |          2 |     1
        1 |          3 |     1
        2 |          4 |     1
        2 |          5 |     1
        4 |          6 |     1
        4 |          7 |     1
        1 |          4 |     2
        1 |          5 |     2
        2 |          6 |     2
        2 |          7 |     2
        1 |          6 |     3
        1 |          7 |     3
(12 rows)
```

Заметим, что в этом примере содержимое таблицы представляет собой дерево, и, по существу, приведенный запрос выполняет обход этого дерева. Если содержимое таблицы представляет более сложный граф, то результаты могут быть некорректными или вычисление может не завершиться. Ниже мы обсудим условия, при которых завершение рекурсивных запросов может быть гарантировано. Сейчас заметим только, что для правильной работы запроса, независимо от наполнения таблицы parent_child, необходимо исключить атрибут level (который не имеет смысла для произвольных графов) и исключить дубликаты, используя UNION вместо UNION ALL в тексте запроса.

Обычный алгоритм вычисления транзитивного замыкания отношения представляет собой тройной цикл, в котором проверяется условие транзитивности отношения и при его нарушении добавляется необходимая пара в отношение. Он имеет сложность N^3, где N — кардинальность множества, на котором задано отношение. Применение этого алгоритма в системах управления базами данных нецелесообразно, т. к. его вычислительная сложность слишком велика и при его работе необходим доступ к отношению в произвольном порядке, что для больших таблиц оказывается крайне неэффективным.

Однако ограничения на рекурсию, накладываемые стандартом SQL и системой PostgreSQL, позволяют вычислять транзитивное замыкание с помощью итеративного алгоритма «снизу вверх». Пусть R обозначает исходное отношение, U_i — приближение к транзитивному замыканию, полученному на итерации i. Представим наивный вариант итеративного алгоритма следующим образом:

1. Инициализация: $U_0 = R$.
2. Шаг итерации: $U_{k+1} = U_k \cup (U_k \bowtie R)$.
3. Условие завершения: $U_{k+1} = U_k$.

Этот алгоритм можно улучшить, если на каждом шаге вычислять соединение не с полным отношением U_i, а только с множеством строк, полученных на предыдущей итерации. Промежуточное отношение U_k представим в виде

$$U_k = U_{k-1} \cup (U_k \setminus U_{k-1}).$$

Такое представление корректно, поскольку на каждой итерации никакие ранее добавленные элементы не включаются: $U_{k-1} \subset U_k$. Применяя дистрибутивность операции соединения относительно объединения, получаем

$$U_k \bowtie R = ((U_k \setminus U_{k-1}) \bowtie R) \cup (U_{k-1} \bowtie R).$$

Подставляя это в выражение, вычисляемое на каждой итерации, получаем

$$U_{k+1} = U_k \cup (U_k \bowtie R) = U_k \cup (U_{k-1} \bowtie R) \cup ((U_k \setminus U_{k-1}) \bowtie R)$$

и, наконец, используя соотношение $U_{k-1} \bowtie R \subset U_k$, получаем

$$U_{k+1} = U_k \cup ((U_k \setminus U_{k-1}) \bowtie R).$$

Условием завершения, как и в исходном варианте алгоритма, является стабилизация U_k, т. е. $U_{k+1} = U_k$, однако технически проще проверять, что приращение оказалось пустым:

$$U_{k+1} \setminus U_k = \varnothing.$$

По существу, этот алгоритм вычисляет рекурсивный запрос итеративно. Реализация рекурсии в PostgreSQL представляет собой вариант этого алгоритма. Примерно такой же алгоритм используется и в других СУБД, поддерживающих рекурсию.

Алгоритмы вычисления рекурсивных запросов «снизу вверх» в реляционной модели данных завершаются, если домены всех атрибутов, включая вычисляемые значения, конечны. Для доказательства заметим, что размер промежуточного результата может только увеличиваться, и этот размер ограничен произведением кардинальностей (размеров) доменов всех атрибутов.

Важно подчеркнуть, что язык SQL не является в точном смысле реляционным, поэтому утверждения, справедливые для реляционных запросов, могут быть неверными для SQL. В частности, рекурсивный запрос, содержащий UNION ALL, может генерировать неограниченное количество идентичных строк, и поэтому ограничение сверху на размер результата оказывается неприменимым. Оно не работает также в тех случаях, когда запрос генерирует значения из бесконечного домена, не принадлежащие исходным отношениям. Например, несложно написать запрос, генерирующий все целые числа. Такой запрос, конечно, завершиться не может.

В противоположность рассмотренному алгоритму «снизу вверх», при вычислении «сверху вниз» обработка рекурсивных правил может зациклиться, не порождая никаких кортежей. Однако при наличии дополнительных условий фильтрации проверка этих условий может выполняться только после завершения рекурсии. По этой причине объем промежуточных результатов, вычисляемых при использовании этого метода, может оказаться чрезмерно большим. Известны более сложные алгоритмы, которые лишены этого недостатка, в частности алгоритм на основе «магических множеств», однако мы не будем их рассматривать. Подробный анализ алгоритмов выполнения рекурсивных запросов можно найти в [61].

15.1.3. Аналитические и оконные функции

Неформально, аналитической обработкой называют такой вид обработки данных, который вырабатывает относительно небольшое количество результатов, представляющих некоторые обобщенные свойства большой совокупности данных. Чаще всего эти обобщенные свойства являются статистическими. Простыми примерами статистических характеристик могут служить среднее значение, количество значений, наиболее часто встречающиеся значения и т. п.

В языке SQL имеется две категории средств, которые могут быть полезны для аналитической обработки: *группировка* (GROUP BY), кратко описанная в главе 4, и *оконные функции*. Напомним, что при группировке несколько строк отношения объединяются вместе, и на основе значений их атрибутов вырабатывается одна строка результата. Эта строка может содержать значения атрибутов, общих для всей группы (по которым производится группировка), и значения некоторых агрегатных функций (count, max, min, avg, а также агрегатных функций, определенных пользователем).

Оконные функции также вычисляют обобщенные характеристики, получаемые агрегированием значений нескольких строк, так или иначе связанных с текущей, но объединение этих строк в одну не выполняется. Следующая серия примеров иллюстрирует применение оконных функций, хотя, конечно, в реальных аналитических задачах необходимы более сложные вычисления.

В приведенном ниже запросе подсчитывается количество пассажиров и количество рейсов, вылетающих из одного аэропорта (SVO) в течение каждого часа в указанный день. Для этого используется группировка с помощью обычного предложения GROUP BY:

```
demo=# SELECT date_part ('hour', f.scheduled_departure) "hour",
              count(ticket_no) passengers_cnt,
              count(DISTINCT f.flight_id) flights_cnt
FROM flights f
    JOIN ticket_flights t ON f.flight_id = t.flight_id
WHERE f.departure_airport = 'SVO'
AND f.scheduled_departure >= '2017-08-02'::date
AND f.scheduled_departure <  '2017-08-03'::date
GROUP BY date_part ('hour', f.scheduled_departure);
 hour | passengers_cnt | flights_cnt
------+----------------+-------------
    9 |            484 |           5
   10 |            381 |           4
   11 |            540 |           7
   12 |            534 |           7
   13 |            157 |           4
   14 |            217 |           4
   16 |            273 |           4
   17 |            421 |           3
   18 |            237 |           3
   19 |             30 |           1
(10 rows)
```

Чтобы сократить размер кода, иллюстрирующего применение оконных функций, определим вспомогательное представление, в котором вычисляется количество рейсов и количество пассажиров за каждый час:

```
demo=# CREATE VIEW per_hour AS
SELECT date_part ('hour', f.scheduled_departure) "hour",
    count(ticket_no) passengers_cnt,
    count(DISTINCT f.flight_id) flights_cnt
FROM flights f
    JOIN ticket_flights t ON f.flight_id = t.flight_id
WHERE f.departure_airport = 'SVO'
AND f.scheduled_departure >= '2017-08-02'::date
AND f.scheduled_departure <  '2017-08-03'::date
GROUP BY date_part ('hour', f.scheduled_departure);
CREATE VIEW
```

Используя оконную функцию avg, можно в каждой строке вывести среднее значение количества пассажиров за час:

```
demo=# SELECT *,
    avg(passengers_cnt) OVER (ROWS BETWEEN UNBOUNDED PRECEDING
                                       AND UNBOUNDED FOLLOWING)
FROM per_hour;
 hour | passengers_cnt | flights_cnt |         avg
------+----------------+-------------+----------------------
    9 |            484 |           5 | 327.4000000000000000
   10 |            381 |           4 | 327.4000000000000000
   11 |            540 |           7 | 327.4000000000000000
   12 |            534 |           7 | 327.4000000000000000
   13 |            157 |           4 | 327.4000000000000000
   14 |            217 |           4 | 327.4000000000000000
   16 |            273 |           4 | 327.4000000000000000
   17 |            421 |           3 | 327.4000000000000000
   18 |            237 |           3 | 327.4000000000000000
   19 |             30 |           1 | 327.4000000000000000
(10 rows)
```

В этом примере среднее вычисляется по всем строкам представления, показывающего почасовую загруженность аэропорта. Другие возможности предложения OVER позволяют задать ограниченный интервал, определяемый количеством строк до и после текущей, задать разбиение строк на группы или их упорядочение. При этом как разбиение на группы, так и упорядочение в предложении OVER могут не совпадать с группировкой, определенной в GROUP BY и упорядочением, задаваемым предложением ORDER BY для всего оператора.

В качестве простой иллюстрации вычислим общее количество пассажиров с начала дня до текущего часа «нарастающим итогом»:

```
demo=# SELECT *,
    sum(passengers_cnt) OVER (ROWS BETWEEN UNBOUNDED PRECEDING
                                       AND CURRENT ROW)
FROM per_hour;
```

```
 hour | passengers_cnt | flights_cnt | sum
------+----------------+-------------+------
    9 |            484 |           5 |  484
   10 |            381 |           4 |  865
   11 |            540 |           7 | 1405
   12 |            534 |           7 | 1939
   13 |            157 |           4 | 2096
   14 |            217 |           4 | 2313
   16 |            273 |           4 | 2586
   17 |            421 |           3 | 3007
   18 |            237 |           3 | 3244
   19 |             30 |           1 | 3274
(10 rows)
```

Окно, по которому выполняется частичное агрегирование, определяется предложением BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. В этом примере окно включает только строки, предшествующие текущей в заданном упорядочении. Однако возможно и включение строк, следующих за текущей, а также указание ограничения на размер окна, например BETWEEN CURRENT ROW–2 AND CURRENT ROW+2 определяет окно размером 5 строк. Такие окна можно применять, например, для сглаживания значений во временны́х рядах.

В следующем примере используется разбиение строк таблицы flights по значениям атрибутов с помощью указания PARTITION BY:

```
demo=# SELECT f.flight_no,
    f.scheduled_departure::time AS dep_time,
    f.departure_airport AS departs,
    f.arrival_airport AS arrives,
    count(flight_id)
        OVER (PARTITION BY departure_airport, arrival_airport)
        AS flight_count
FROM flights f
WHERE f.departure_airport = 'KZN'
AND f.scheduled_departure >= '2017-08-02'::date
AND f.scheduled_departure <  '2017-08-03'::date
ORDER BY flight_count DESC,
         f.arrival_airport,
         f.scheduled_departure;
 flight_no | dep_time | departs | arrives | flight_count
-----------+----------+---------+---------+--------------
 PG0609    | 11:15:00 | KZN     | ROV     |            2
 PG0610    | 12:45:00 | KZN     | ROV     |            2
 PG0203    | 17:45:00 | KZN     | DME     |            1
 PG0498    | 10:15:00 | KZN     | IKT     |            1
 PG0039    | 17:25:00 | KZN     | KVX     |            1
 PG0508    | 09:45:00 | KZN     | LED     |            1
 PG0621    | 16:05:00 | KZN     | MQF     |            1
 PG0579    | 15:45:00 | KZN     | PKV     |            1
(8 rows)
```

Приведенный запрос выводит для каждого рейса время вылета, коды аэропортов отправления и назначения, а также количество рейсов, следующих по такому же маршруту. Поскольку в предложении WHERE задается выборка только данных за один день, при подсчете учитываются рейсы, выполняемые в тот же день. Конечно, можно добавить дату вылета в список значений, по которым делается разбиение. Упорядочение результата выбрано таким образом, чтобы сначала появлялись наиболее популярные направления, а внутри каждого направления рейсы упорядочены по времени вылета. Подчеркнем, что упорядочивание результата никак не связано с вычислением функции count по окну.

15.2. Избыточные структуры хранения

Структура хранения называется *избыточной*, если ее удаление из базы данных не приведет к потере информации. Другими словами, данные, содержащиеся в избыточной структуре, можно вычислить, используя данные из других структур. Очевидно, что любые результаты выполнения запросов, которые могут быть получены при использовании избыточных структур, могут быть получены и без их использования, поэтому избыточные структуры имеет смысл создавать и поддерживать только в тех случаях, когда они делают работу с базой данных более эффективной. Как правило, избыточные структуры обеспечивают существенное ускорение выполнения некоторых классов запросов, однако их ведение (например, поддержание в актуальном состоянии) создает дополнительную нагрузку на систему и может поэтому привести к некоторому снижению эффективности выполнения других функций.

Мы уже неоднократно упоминали два вида избыточных структур хранения: *материализованные представления* и *индексы*. Избыточность хранения возникает также при репликации, которая обсуждается в связи с распределенными системами баз данных в главе 21.

15.2.1. Материализованные представления

При рассмотрении представлений в разделе 4.3.10 подчеркивалось, что представления не хранятся в базе данных, а вместо этого данные, видимые через них, вычисляются заново при выполнении каждого запроса, ссылающегося на представления. В отличие от обычных представлений материализованные

представления хранятся в базе данных как таблицы, но, в отличие от таблиц, их содержимое определяется запросом, как для обычных представлений.

Следующий оператор SQL создает материализованное представление, идентичное представлению, которое использовалось в разделе 15.1.3 для иллюстрации оконных функций:

```
demo=# CREATE MATERIALIZED VIEW per_hour AS
SELECT date_part ('hour', f.scheduled_departure) "hour",
    count(ticket_no) passengers_cnt,
    count(DISTINCT f.flight_id) flights_cnt
FROM flights f
    JOIN ticket_flights t ON f.flight_id = t.flight_id
WHERE f.departure_airport = 'SVO'
AND f.scheduled_departure >= '2017-08-02'::date
AND f.scheduled_departure <  '2017-08-03'::date
GROUP BY date_part ('hour', f.scheduled_departure);
SELECT 10
```

Это материализованное представление можно использовать в запросе:

```
demo=# SELECT *,
    sum(passengers_cnt) OVER (ROWS BETWEEN UNBOUNDED PRECEDING
                                       AND CURRENT ROW)
FROM per_hour;
 hour | passengers_cnt | flights_cnt | sum
------+----------------+-------------+------
    9 |            484 |           5 |  484
   10 |            381 |           4 |  865
   11 |            540 |           7 | 1405
   12 |            534 |           7 | 1939
   13 |            157 |           4 | 2096
   14 |            217 |           4 | 2313
   16 |            273 |           4 | 2586
   17 |            421 |           3 | 3007
   18 |            237 |           3 | 3244
   19 |             30 |           1 | 3274
(10 rows)
```

Время построения материализованного представления соответствует времени выполнения запроса, определяющего это представление, а запрос, выбирающий данные из созданного материализованного представления, выполняется моментально.

Очевидно, что материализованные представления целесообразно применять в тех случаях, когда необходимо многократное повторение ресурсоемких вычислений (в нашем примере это соединение нескольких таблиц и агрегирование). При этом наибольшая результативность достигается, если результаты

вычислений оказываются значительно меньшими по объему, чем промежуточные результаты, вырабатываемые во время выполнения запроса, определяющего представление.

Как и для таблиц, для материализованных представлений можно строить индексы.

Данные, записанные в материализованное представление, не обновляются автоматически при обновлении данных в таблицах, на основе которых было заполнено материализованное представление. В различных СУБД предоставляются различные возможности для приведения данных в материализованных представлениях в актуальное состояние. В системе PostgreSQL для актуализации материализованных представлений предусмотрен оператор SQL REFRESH MATERIALIZED VIEW. Этот оператор полностью заменяет все данные, хранимые в материализованном представлении, на результат нового вычисления запроса, определяющего это представление. Обычно для выполнения оператора REFRESH устанавливаются блокировки, исключающие доступ к материализованному представлению на время актуализации, но при указании режима CONCURRENTLY доступ к материализованному представлению на чтение допускается.

Обычно оператор актуализации материализованных представлений запускается автоматически на основе расписания, реализуемого средствами операционной системы (например, cron). Очевидно, что актуализация может создавать значительную дополнительную нагрузку на систему, и в любом случае материализованное представление может содержать устаревшие данные. Поэтому применять материализованные представления можно, только если использование устаревших данных явно или неявно допускается функциональными требованиями к прикладной системе. Например, если отчеты не используют данные за текущие сутки, то материализованные представления, на основе которых получаются эти отчеты, можно актуализировать один раз в сутки в интервалы наименьшей загруженности базы данных.

В некоторых СУБД расписание актуализации можно задавать непосредственно при определении материализованного представления.

Время, необходимое для актуализации материализованного представления, можно существенно сократить, применяя пошаговое обновление (incremental refresh). При пошаговом обновлении заново вычисляются только те строки материализованного представления, которые зависят от изменений, произошедших в базовых таблицах после предыдущего обновления материализованного представления.

Чтобы выполнить пошаговое обновление, необходимо определить, какие строки материализованного представления зависят от изменений и какие данные из базовых таблиц необходимы для вычисления этих строк материализованного представления. Существуют запросы, для которых это сделать невозможно, поэтому во всех системах пошаговое обновление применимо только для материализованных представлений, удовлетворяющих некоторым ограничениям (различным для разных СУБД). В основной версии системы PostgreSQL пошаговое обновление пока не реализовано.

15.2.2. Индексы

Напомним, что индексом называется дополнительная (избыточная) структура хранения, обеспечивающая ускорение некоторых запросов и прозрачная (невидимая) для приложения. Прозрачность в этом контексте означает, что язык запросов SQL не предоставляет никаких средств для явного указания на то, что какой-либо индекс следует использовать для выполнения запроса. Вопрос о том, какие из имеющихся индексов использовать для выполнения конкретного запроса, решается оптимизатором (планировщиком). Не менее важно то, что результат выполнения запроса не зависит ни от наличия каких-либо индексов, ни от того, использовались ли они для выполнения. Изменяться может только количество необходимых вычислительных ресурсов и время выполнения запроса, но не его результат.

Кроме ускорения операций поиска, индексы необходимы для поддержки ограничений целостности. В частности, для проверки ограничения целостности UNIQUE при объявлении этого ограничения автоматически строится уникальный индекс по совокупности атрибутов, включенных в это ограничение.

Не следует, однако, смешивать или отождествлять ограничения целостности с уникальными индексами. Прежде всего ограничения целостности являются декоративными и описывают свойства данных, а индексы являются структурой хранения. Кроме этого, уникальные индексы, как и любые другие, могут быть необходимы для ускорения поиска, а не только для запрета повторяющихся значений. Если уникальный индекс является составным (т. е. построен по нескольким атрибутам отношения), то их порядок в индексе важен для определения того, для выполнения каких запросов он будет применяться, а для проверки уникальности значений порядок атрибутов несуществен. В некоторых случаях целесообразно создавать несколько индексов, отличающихся только порядком включенных атрибутов.

Для того чтобы обеспечить возможность выполнения запросов только по индексу без обращения к строкам таблицы, в индекс может быть включено большое количество атрибутов. Например, такой индекс может содержать все атрибуты первичного ключа и еще некоторые. Объявление ограничения целостности по такому набору атрибутов бессмысленно, хотя значения составного ключа в этом индексе будут, очевидно, уникальными, потому что уникальны значения первичного ключа. Система PostgreSQL позволяет создать уникальный индекс, дополненный (с помощью предложения INCLUDE) неключевыми атрибутами, и явно указать его при объявлении первичного или уникального ключа. Такие дополнительные атрибуты не учитываются в условии поиска, но могут быть использованы при доступе только по индексу.

Эффект, который можно получить от применения индексов, продемонстрирован в разделе 4.4, а в главе 11 рассмотрены причины, по которым применение индексов может давать очень значительное сокращение затрат на выполнение запросов: это позволяет заменить последовательный просмотр таблиц на быстрый поиск по индексу, возможно, с последующим точечным доступом к отдельным страницам. При этом количество просматриваемых данных может уменьшиться на несколько порядков (для таблиц большого размера). Невозможно указать точные границы, но обычно применение индексов для улучшения производительности целесообразно рассматривать для таблиц, занимающих десятки страниц или больше.

Создание и поддержание индексов в актуальном состоянии требует некоторых вычислительных ресурсов. В частности, при выполнении операций обновления (UPDATE, INSERT, DELETE) необходимо не только внести изменения в основное хранилище таблицы, но и соответствующим образом изменить все индексы, построенные для этой таблицы. Этот факт подчеркивается во многих руководствах по проектированию баз данных. Нет сомнений в том, что поддержка индексов создает дополнительную нагрузку на сервер базы данных, но:

- современные методы ведения журнала [6] дают возможность регистрировать изменения индексов значительно более эффективно, чем изменения в таблицах: достаточно записывать в журнал информацию об операциях логического уровня (вставки и удаления), но не записывать копии страниц индекса;
- в системе PostgreSQL применяется механизм обновления HOT (heap-only tuple update), исключающий создание новых индексных записей для изменяемых версий строк таблицы, в которых значения атрибутов, использованных для построения индекса, не изменялись;

- в современных вариантах нагрузок типа OLTP доминируют запросы на чтение, а доля операций обновления относительно невелика.

По указанным причинам влияние дополнительной нагрузки на сервер для ведения индексов не так значительно, как несколько десятилетий назад.

Негативное влияние индексов может оказаться существенным только при массовых обновлениях (например, при массовой загрузке данных). В некоторых случаях может быть целесообразно удалить индекс перед массовым обновлением и затем создать его заново. Чтобы предотвратить блокирование данных на время построения индекса, в команде CREATE INDEX необходимо указывать ключевое слово CONCURRENTLY. Однако этот прием, хотя и ускоряет массовое обновление, неизбежно приведет к существенному снижению производительности при чтении из-за временного отсутствия индекса. По-видимому, он неприменим для систем, в которых важна высокая доступность (например, работающих в режиме 24×7).

Более удачным решением задачи массовой загрузки может оказаться *секционирование*: вновь загружаемые данных размещаются в новой секции, для которой строится индекс, при этом как во время загрузки, так и во время построения индексов возможна нормальная работа с ранее загруженными данными.

Несмотря на то что в SQL нет средств для явного управления использованием индексов для конкретных запросов, в литературе и в документации по многим СУБД можно найти рекомендации, каким образом управлять использованием индексов неявно.

Часто рекомендуемый прием основан на том, что в индексе хранятся значения атрибута или комбинации атрибутов. Для того чтобы индекс был применим для выполнения запроса, необходимо, чтобы в условиях фильтрации (или в условиях соединения) были указаны операции сравнения значений этого атрибута, которые поддерживаются этим типом индекса. Например, для индексов на основе B-дерева такими операциями являются сравнения на равенство или неравенства (больше или меньше) значений. Если условие накладывается не на значение атрибута, а на зависящее от него выражение, то применение индекса по значениям атрибута становится невозможно.

Рассмотрим пример. Следующий запрос, как видно из приведенного результата выполнения, выводит небольшое (по сравнению с размерами таблицы tickets) количество строк, однако для атрибутов, по которым происходит фильтрация, нет индексов. Поэтому план выполнения запроса содержит операцию, выполняющую полный просмотр этой таблицы. Сканирование выполняется

параллельно, поскольку конфигурация компьютера это обеспечивает, и сервер баз данных PostgreSQL по умолчанию использует такие возможности, когда они есть. Это, однако, не оказывает влияния на выбор операции доступа к данным: в данном случае никакой другой алгоритм, кроме полного просмотра, использовать невозможно.

```
demo=# SELECT book_ref
FROM tickets
WHERE passenger_name ='ALLA ZOTOVA'
AND book_ref >= '3FB8EB';
 book_ref
----------
 3FB8EB
 40F255
 688445
(3 rows)
Time: 232,258 ms
```

```
demo=# EXPLAIN (costs off)
SELECT book_ref
FROM tickets
WHERE passenger_name ='ALLA ZOTOVA'
AND book_ref >= '3FB8EB';
                        QUERY PLAN
-----------------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Seq Scan on tickets
         Filter: ((book_ref >= '3FB8EB'::bpchar) AND
                  (passenger_name = 'ALLA ZOTOVA'::text))
```

Попробуем создать индекс на атрибут book_ref.

```
demo=# CREATE INDEX ON tickets(book_ref);
CREATE INDEX
Time: 1133,852 ms (00:01,134)
```

Создание занимает значительное время, однако этот индекс не используется, в чем можно убедиться, проверив план выполнения. Причина в том, что ожидаемое количество строк результата при выборке данных по условию на этот атрибут велико:

```
demo=# SELECT count(*)
FROM tickets
WHERE book_ref >= '3FB8EB';
 count
--------
 275823
(1 row)
```

Попробуем вместо индекса на book_ref создать индекс на passenger_name. Поскольку условие на этот атрибут является условием равенства, ожидаемое количество строк результата невелико и поэтому план выполнения запроса получается другим.

```
demo=# CREATE INDEX ON tickets(passenger_name);
CREATE INDEX
Time: 1263,743 ms (00:01,264)
demo=# SELECT book_ref
FROM tickets
WHERE passenger_name ='ALLA ZOTOVA'
AND book_ref >= '3FB8EB';
 book_ref
----------
 3FB8EB
 40F255
 688445
(3 rows)
Time: 0,938 ms
demo=# EXPLAIN (costs off)
SELECT book_ref
FROM tickets
WHERE passenger_name ='ALLA ZOTOVA'
AND book_ref >= '3FB8EB';
                          QUERY PLAN
---------------------------------------------------------------
 Bitmap Heap Scan on tickets
   Recheck Cond: (passenger_name = 'ALLA ZOTOVA'::text)
   Filter: (book_ref >= '3FB8EB'::bpchar)
   ->  Bitmap Index Scan on tickets_passenger_name_idx
         Index Cond: (passenger_name = 'ALLA ZOTOVA'::text)
```

Изменим запрос таким образом, чтобы условие на атрибут, по которому построен индекс, содержало бы выражение. Результат, конечно, получится тот же самый, но индекс не используется.

```
demo=# EXPLAIN (costs off)
SELECT book_ref
FROM tickets
WHERE upper(passenger_name) = 'ALLA ZOTOVA'
AND book_ref >= '3FB8EB';
                             QUERY PLAN
--------------------------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Seq Scan on tickets
         Filter: ((book_ref >= '3FB8EB'::bpchar) AND
                  (upper(passenger_name) = 'ALLA ZOTOVA'::text))
```

Наконец, посмотрим, как используется индекс на оба атрибута, упоминаемые в этом запросе.

```
demo=# CREATE INDEX ON tickets(passenger_name, book_ref);
CREATE INDEX
Time: 1274,102 ms (00:01,274)
demo=# SELECT book_ref
FROM tickets
WHERE passenger_name = 'ALLA ZOTOVA'
AND book_ref >= '3FB8EB';
 book_ref
----------
 3FB8EB
 40F255
 688445
(3 rows)
Time: 0,832 ms
demo=# EXPLAIN (costs off)
SELECT book_ref
FROM tickets
WHERE passenger_name = 'ALLA ZOTOVA'
AND book_ref >= '3FB8EB';
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on tickets
   Recheck Cond: ((passenger_name = 'ALLA ZOTOVA'::text) AND
                  (book_ref >= '3FB8EB'::bpchar))
   ->  Bitmap Index Scan on tickets_passenger_name_book_ref_idx
         Index Cond: ((passenger_name = 'ALLA ZOTOVA'::text) AND
                      (book_ref >= '3FB8EB'::bpchar))
```

Таблица tickets относительно невелика даже в самом большом варианте демонстрационной базы данных, поэтому различие во времени выполнения не очень значительно. Однако для таблиц большего размера различие может быть очень существенным. Важно не абсолютное время выполнения, а то, как оно будет расти с увеличением размеров таблицы.

Во многих случаях необходимо выполнять поиск не непосредственно по значениям атрибутов, а по некоторым выражениям, зависящим от одного или нескольких атрибутов. Например, при задании условий на значения текстовых атрибутов может быть необходимо игнорировать различие между прописными и строчными буквами. Такие условия легко записать, применяя функции upper или lower к атрибуту, для которого задается условие, однако при этом получится выражение, для поиска по которому нельзя использовать обычный индекс.

В системе PostgreSQL в подобных случаях можно построить индекс не по значениям атрибута, а по значениям выражения. Для того чтобы такой индекс

был использован при выполнении запроса, необходимо, чтобы выражение, на значение которого накладывается условие фильтрации в запросе, текстуально совпадало с выражением, заданным при создании индекса.

Рассмотрим пример. В демонстрационной базе атрибут passenger_name содержит имя и фамилию пассажира, разделенные пробелом. Для того чтобы выполнить поиск только по фамилии пассажира (без учета имени), можно применить встроенные функции substring и position, но тогда не будет использован индекс по значениям атрибута. Можно, однако, построить индекс только по фамилии, и такой индекс будет использован:

```
demo=# CREATE INDEX ON tickets(substring(
    passenger_name FROM position(' ' IN passenger_name) + 1
));
CREATE INDEX
demo=# EXPLAIN (costs off)
SELECT book_ref, passenger_name
FROM tickets
WHERE substring(
    passenger_name FROM position(' ' IN passenger_name) + 1
) = 'ZOTOVA'
AND book_ref >= '3FB8EB';
                        QUERY PLAN
-----------------------------------------------------------
 Bitmap Heap Scan on tickets
   Recheck Cond: ("substring"(...) = 'ZOTOVA'::text)
   Filter: (book_ref >= '3FB8EB'::bpchar)
   ->  Bitmap Index Scan on tickets_substring_idx
         Index Cond: ("substring"(...) = 'ZOTOVA'::text)
```

Однако если изменить выражение в запросе (переставлены слагаемые в выражении, задающем начало подстроки), то индекс использоваться не будет:

```
demo=# EXPLAIN (costs off)
SELECT book_ref, passenger_name
FROM tickets
WHERE substring(
    passenger_name FROM 1 + position(' ' IN passenger_name)
) = 'ZOTOVA'
AND book_ref >= '3FB8EB';
                       QUERY PLAN
-------------------------------------------------------
 Gather
   Workers Planned: 2
   ->  Parallel Seq Scan on tickets
         Filter: ((book_ref >= '3FB8EB'::bpchar) AND
                  ("substring"(...) = 'ZOTOVA'::text))
```

Мы уже отмечали, что целесообразность применения индексов зависит от селективности: чем меньше доля строк, которые требуется выбрать, по отношению к общему числу строк в таблице, тем полезнее применение индексов. Если распределения значений атрибута близко к равномерному, то количество различных значений атрибута велико (и, следовательно, каждое из значений встречается в небольшом количестве строк). Это во всяком случае справедливо для запросов типа OLTP. В реальности, однако, преобладают неравномерные распределения. В этом случае одно или небольшое количество значений встречается почти во всех строках, а все остальные значения встречаются редко. Частный случай такой ситуации возникает, когда некоторый атрибут имеет неопределенное значение почти для всех строк таблицы.

В подобных ситуациях оптимальный план выполнения запроса может зависеть от значений атрибута: для редко встречающихся значений целесообразно использовать индекс, а для частых — полный просмотр. В подобных ситуациях крайне полезна возможность создания *частичных* (условных) индексов, реализованная в системе PostgreSQL. При создании такого индекса задается предложение WHERE, определяющее, какие строки таблицы будут учтены. Для того чтобы такой индекс был использован в запросе, предложение WHERE этого запроса должно содержать условие, которое планировщик PostgreSQL сможет свести к условию, заданному при создании индекса.

В следующем примере создается индекс по аэропорту отправления таблицы flights демобазы для рейсов, вылетающих после 1 сентября 2017 г. В запросе требуется вывести информацию о рейсах, вылетающих из указанного аэропорта после 1 октября того же года. Условие в запросе сильнее, чем условие, по которому построен индекс, поэтому индекс используется:

```
demo=# CREATE INDEX ON flights(departure_airport)
    WHERE scheduled_departure > '2017-09-01';
CREATE INDEX

demo=# EXPLAIN (costs off)
SELECT departure_airport, scheduled_arrival
FROM flights
WHERE departure_airport = 'SVO'
AND scheduled_departure > '2017-10-01';
                            QUERY PLAN
------------------------------------------------------------------
 Bitmap Heap Scan on flights
   Recheck Cond: ((departure_airport = 'SVO'::bpchar) AND
                  (scheduled_departure > '2017-09-01'::timestamptz))
   Filter: (scheduled_departure > '2017-10-01'::timestamptz)
   ->  Bitmap Index Scan on flights_departure_airport_idx
         Index Cond: (departure_airport = 'SVO'::bpchar)
```

Однако если запросить рейсы, вылетающие после 1 августа, то индекс не будет использован:

```
demo=# EXPLAIN (costs off)
SELECT departure_airport, scheduled_arrival
FROM flights
WHERE departure_airport = 'SVO'
AND scheduled_departure > '2017-08-01';
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on flights
   Filter: ((scheduled_departure > '2017-08-01'::timestamptz) AND
            (departure_airport = 'SVO'::bpchar))
```

15.3. Итоги главы

В главе рассматриваются методы и алгоритмы, применяемые для реализации общих табличных выражений и рекурсии в языке SQL. Общие табличные выражения позволяют улучшить структурирование текста запроса и таким образом облегчить его понимание, но могут привести к потере вычислительной эффективности.

Рекурсивные запросы позволяют записать на языке SQL спецификации вычислений, которые невозможно сформулировать в рамках реляционных языков запросов. Ограничения на использование рекурсии в SQL гарантируют однозначность их интерпретации, тем самым обеспечивая независимость результата от алгоритма его вычисления. В реализациях SQL применяются итеративные методы вычисления рекурсивных запросов «снизу вверх».

Применение оконных функций SQL позволяет упростить некоторые виды агрегирующих запросов.

Обсуждается также применение избыточных структур хранения, к которым относятся материализованные представления и индексы. Материализованные представления позволяют сохранить результаты вычислений большого объема, которые часто используются в запросах. Они полезны в тех случаях, когда допустимо некоторое отставание результатов от актуальных данных, и требуют регулярной перестройки. Функциональные индексы дают возможность исключить полный просмотр для сложных условий, содержащих выражения и функции, а частичные (условные) индексы полезны для атрибутов с неравномерным распределением значений.

15.4. Упражнения

Упражнение 15.1. Напишите рекурсивный запрос, который строит маршруты с пересадками, прибывающие в Москву. Маршрут не должен возвращаться в ранее посещенные пункты пересадки. На каждой пересадке интервал времени между прибытием и отправлением следующего рейса должен быть не менее 1 часа и не более 24 часов. Общая продолжительность маршрута не должна превышать 48 часов.

Упражнение 15.2. Найдите маршруты с пересадками, прибывающие в Москву из аэропортов, не имеющих прямых рейсов в аэропорты Москвы.

Упражнение 15.3. Найдите маршруты с пересадками, прибывающие в Москву, стоимость которых ниже, чем стоимость прямых рейсов от начального до конечного пункта маршрута.

Упражнение 15.4. Постройте пример запроса, в котором применение общих табличных выражений приводит к значительному ухудшению времени выполнения.

Упражнение 15.5. Постройте функциональный индекс, используя приведение значения к верхнему регистру (функция upper), и приведите пример запроса, для которого такой индекс сокращает время выполнения.

Упражнение 15.6. Постройте частичный индекс, в котором исключаются строки, содержащие значение NULL, и приведите пример запроса, для которого этот индекс улучшает время выполнения.

Упражнение 15.7. Напишите запрос, подсчитывающий количество пассажиров на каждом рейсе и среднее количество пассажиров на рейсах, следующих в тот же день по тому же маршруту. Выведите только рейсы, следующие по маршрутам, по которым имеется несколько рейсов в один день.

Упражнение 15.8. Создайте материализованное представление, наиболее существенно сокращающее время выполнения запроса из упражнения 15.7.

Глава 16

Функции и процедуры в базе данных

16.1. Хранимые подпрограммы

Возможности расширения функциональности системы PostgreSQL в значительной мере основаны на механизме *хранимых подпрограмм* (routines), определяемых пользователями. В системе PostgreSQL имеются два вида таких подпрограмм: *функции* и *процедуры*. Основное различие между ними состоит в том, что функции возвращают результат. Для того чтобы функция была вызвана и выполнена, ее необходимо использовать в выражении в операторе SQL, например включить в список выражений, возвращаемых оператором SELECT. Процедуры не вырабатывают результат, а для вызова процедуры необходимо использовать оператор CALL. Есть и другие различия, которые обсуждаются далее в этом разделе.

Применения таких функций и процедур многообразны и включают как дополнение или изменение функций сервера баз данных, так и реализацию функциональности конкретных прикладных систем:

- реализация операций над пользовательскими типами данных;
- определение пользовательских агрегатов и оконных функций;
- определение триггеров;
- обеспечение разграничения доступа для пользователей приложений;
- реализация функций приложения, требующих интенсивной работы с базой данных.

Конечно, возможности применения хранимых подпрограмм не исчерпываются этим списком.

Хранимые процедуры и функции выполняются в рамках процессов сервера баз данных и поэтому могут выполняться более эффективно, чем код в программе-клиенте. В частности, нет необходимости в пересылке данных по сети. С другой

стороны, применение функций может существенно изменить поведение сервера баз данных (например, при массовом применении триггеров). Кроме этого, плохо продуманное использование функций, определенных пользователем, может фактически блокировать работу оптимизатора.

Высокая эффективность исполнителя запросов в реляционных СУБД в значительной мере основана на декларативности языка запросов SQL. Возможности оптимизации запросов наиболее полны в случае, если оператор SELECT не содержит подзапросы, изменяющие базы данных или вызовы функций, свойства которых неизвестны оптимизатору. Императивные (процедурные) языки предоставляют средства, неконтролируемое использование которых может как существенно ограничить возможности оптимизации, так и помешать эффективному выполнению.

Негативные последствия применения функций и процедур могут возникать по следующим причинам:

Раздельное выполнение подзапросов. Существенная часть работы оптимизатора запросов состоит в изменении порядка выполнения операций, указанных в запросе, например вложенные запросы могут преобразовываться в операцию соединения. Однако подзапрос, размещенный внутри функции, оптимизируется и выполняется отдельно от основного запроса, в котором использована функция.

Побочные эффекты функций. Возможности перестройки планов существенно сокращаются, если использованные в запросе функции не обладают некоторыми необходимыми свойствами, например изменяют состояние базы данных.

Недоступность оценок стоимости. Если при определении функции не указаны параметры, описывающие стоимость ее выполнения, оптимизатор не может правильно оценить эту стоимость. Во многих случаях стоимость существенно зависит от значений параметров вызова функции, поэтому даже явное указание коэффициентов в определении функции оказывается не очень полезным.

Для того чтобы уменьшить возможные отрицательные эффекты от использования хранимых процедур, в системе PostgreSQL предусмотрен ряд средств, позволяющих описать свойства функций или процедур и ограничить действие факторов, потенциально снижающих эффективность исполнителя.

Вызов функции всегда выполняется для некоторого запроса клиентского приложения, поэтому действие функции ограничено этим запросом, даже если

внутри функции выполняется несколько операторов SQL или вызываются другие функции. В теле функции нельзя использовать операторы управления транзакциями.

В отличие от функций процедуры вызываются оператором CALL и могут управлять транзакциями, если только оператор вызова сам не находится внутри транзакции.

В отличие от методов объектов в традиционных языках программирования функции (и процедуры) не могут сохранять никакие значения во внутренних переменных между разными вызовами одной и той же или различных функций. Все данные, которые передаются между вызовами, должны быть переданы через параметры функций или процедур, конфигурационные параметры сервера базы данных или записаны в базу данных. Существуют также расширения, позволяющие использовать глобальные переменные.

Зависимость результата выполнения функции от контекста описывается свойствами *изменчивости*.

- Функции с пометкой IMMUTABLE должны возвращать значения, зависящие только от их параметров, но не от содержимого базы данных или других значений (например, от времени), и не могут модифицировать базу данных. Если значения параметров такой функции известны на этапе планирования, она может быть вычислена планировщиком еще до выполнения запроса.
- Функции, помеченные как STABLE, обязаны возвращать одни и те же значения при совпадающих значениях параметров и состояниях базы данных, но не могут зависеть от других значений и не могут модифицировать базу данных. Обычно при определении STABLE подчеркивается, что одно и то же значение возвращается в пределах одного запроса. Если такая функция вызывается (с одинаковыми параметрами) несколько раз в одном запросе, оптимизатор может использовать результаты первого вызова вместо повторного выполнения функции.
- Функции, для которых какие-либо из этих условий не выполнены, помечаются как VOLATILE. Если в запросе присутствуют такие функции, оптимизатор может применять только трансформации плана, не изменяющие контекст (параметры и состояние базы данных), в котором выполняются эти функции.

Разницу в том, какие трансформации плана запроса, содержащего вызов функции, может применить оптимизатор, проиллюстрируем простым примером.

Создадим функцию без параметров, возвращающую ложное значение, и проверим планы выполнения запросов с условием фильтрации, включающем вызов этой функции.

Если функция создана с пометкой IMMUTABLE, фильтрация выполняется на этапе планирования и при выполнении запроса доступ к таблице не требуется:

```
demo=# CREATE FUNCTION never() RETURNS boolean
LANGUAGE plpgsql IMMUTABLE AS $$
BEGIN
    RETURN false;
END;
$$;
CREATE FUNCTION
demo=# EXPLAIN (costs off)
SELECT * FROM aircrafts WHERE never();
        QUERY PLAN
--------------------------
 Result
   One-Time Filter: false
```

При смене категории изменчивости на STABLE оптимизатор включает в план выполнения запроса доступ к таблице, но при фильтрации функция будет вызвана однократно:

```
demo=# ALTER FUNCTION never STABLE;
ALTER FUNCTION
demo=# EXPLAIN (costs off)
SELECT * FROM aircrafts WHERE never();
             QUERY PLAN
------------------------------------
 Result
   One-Time Filter: never()
   ->  Seq Scan on aircrafts_data ml
```

Наконец, при пометке VOLATILE оптимизатор вынужден запланировать вызов функции для каждой строки результата:

```
demo=# ALTER FUNCTION never VOLATILE;
ALTER FUNCTION
demo=# EXPLAIN (costs off)
SELECT * FROM aircrafts WHERE never();
          QUERY PLAN
------------------------------
 Seq Scan on aircrafts_data ml
   Filter: never()
```

Выбор плана выполнения запроса зависит от стоимости плана, которая оценивается встроенными в оптимизатор моделями стоимости. Однако оптимизатор не может оценить стоимость выполнения пользовательских функций. Чтобы улучшить результаты работы оптимизатора, целесообразно для функций, содержащих сложные вычисления, задавать оценки стоимости выполнения.

Значения параметров COST и ROWS указывают оптимизатору ожидаемую стоимость выполнения функции и (для функций, возвращающих несколько строк) ожидаемое количество строк. Конечно, эти параметры недостаточны для точной оценки стоимости выполнения, но, если эти параметры заданы, оптимизатор получает хотя бы некоторую информацию о стоимости. Однако значения этих параметров никак не связаны со значениями аргументов вызова функции, поэтому их полезность ограничена.

В системе PostgreSQL тело подпрограммы может быть записано на любом из языков программирования, известных серверу баз данных во время выполнения оператора, создающего функцию или процедуру.

В любой конфигурации системы PostgreSQL можно использовать подпрограммы, написанные на языках C и SQL, которые считаются внутренними языками системы. Все остальные языки, на которых могут быть запрограммированы хранимые функции, принято называть *процедурными*. Для каждого процедурного языка в базе данных создается обработчик (handler) этого языка. В состав СУБД PostgreSQL основной распространяемой версии входят языки PL/pgSQL, Python, завоевавший широкую популярность в последнее десятилетие, в особенности в среде специалистов по анализу данных и машинному обучению, а также Tcl и Perl, широко применявшиеся для манипулирования текстовыми строками. Кроме перечисленных языков, существуют расширения, позволяющие использовать и другие языки, в том числе R, Java, JavaScript и командный язык системы Unix.

Все функции и процедуры, независимо от того, на каком языке программирования они написаны, должны быть специфицированы оператором языка SQL CREATE FUNCTION или CREATE PROCEDURE. Поскольку спецификация содержит определение параметров, в качестве типов параметров можно использовать только типы языка SQL, в том числе любые пользовательские типы, определенные в базе данных во время выполнения оператора создания подпрограммы. Функции и процедуры зависят от типов аргументов, поэтому определения типов не могут быть удалены, до тех пор пока не будут удалены зависящие от них подпрограммы (возможно, в результате указания CASCADE в операторе DROP).

Операторы SQL CREATE FUNCTION или CREATE PROCEDURE содержат тело функции или указание файла, из которого оно может быть загружено. Тело функции, если оно включено в оператор CREATE, записывается в виде константы, заключенной в кавычки. Кавычки могут быть необходимы и в теле функции; чтобы избежать их удваивания, целесообразно использовать именованные кавычки вида $*имя*$. Закрывающей кавычкой для нее становится только кавычка с таким же именем.

Функции и процедуры идентифицируются совокупностью, включающей:

1) имя функции;
2) имя схемы, в которой размещена функция;
3) количество и типы параметров.

Иначе говоря, функции и процедуры, отличающиеся списком параметров, могут сосуществовать в одной схеме, что обеспечивает *полиморфизм*: при вызове функции выбирается та, которая имеет подходящие типы формальных параметров. Иногда этот вид полиморфизма называют *перегрузкой* (overloading). При этом тип возвращаемого значения не является идентифицирующим, т. е. функции, которые отличаются только типом возвращаемых значений, сосуществовать в одной схеме не могут. Для изменения типа возвращаемого значения необходимо сначала удалить старое определение функции (оператором DROP).

Другая форма полиморфизма обеспечивается аппаратом *псевдотипов*. Если тип параметра функции задан псевдотипом, то в качестве значения такого параметра можно подставлять значение любого типа, соответствующего этому псевдотипу. Например, если параметр функции имеет тип anyarray, то значением такого параметра может быть любой массив.

В программных системах, в которых применяется более одного языка программирования, как правило, требуется преобразование типов данных. В системе PostgreSQL функции и процедуры вызываются из языка SQL, поэтому требуется преобразование входных аргументов подпрограммы из типа данных SQL в тип данных языка программирования и обратное преобразование для выходных параметров и результатов. Исключение составляют подпрограммы, тело которых написано на SQL или на PL/pgSQL, потому что в этих языках применяется система типов SQL.

Совокупность из двух функций, выполняющих такие преобразования, называется *трансформацией*. В обработчиках процедурных языков предусмотрены трансформации. Для многих типов, в частности пользовательских, системные

трансформации используют текстовое представление значений. Имеется возможность создать (оператором CREATE TRANSFORM) альтернативную, более эффективную трансформацию и задать ее в определении подпрограммы.

Аргументы функций могут быть входными (IN), выходными (OUT), входными и выходными (INOUT), а также повторяемые (VARIADIC). Выходные аргументы функций включаются в возвращаемый результат. Если выходных аргументов несколько, они образуют кортеж, как в следующем примере:

```
demo=# CREATE FUNCTION get_row(a int, b OUT int, c OUT int)
LANGUAGE plpgsql AS $$
BEGIN
    b = a + 1;
    c = a + 2;
END;
$$;
CREATE FUNCTION
demo=# SELECT get_row(1);
 get_row
---------
 (2,3)
(1 row)
```

Фактические значения аргументов подпрограмм разделяются запятыми. Их можно задавать позиционно (первое значение соответствует первому аргументу, второе — второму и т. д.), а также в формате *имя_аргумента* => *значение*. В определении функции можно указывать значения аргументов, используемые по умолчанию. Если такие значения заданы, то соответствующие параметры можно не указывать при вызове подпрограммы. Важно обратить внимание на то, что применение значений по умолчанию одновременно с полиморфизмом может приводить к неожиданным результатам. В качестве значений можно записывать любые выражения SQL подходящего типа или приводимого к подходящему типу.

Значения, возвращаемые функциями PostgreSQL, могут быть скалярными типами данных, кортежами, массивами или отношениями. Примеры применения встроенных функций, возвращающих отношения, имеются в главе 8.

В процедурах не допускается указание выходных (OUT) аргументов, потому что процедуры не возвращают результат, однако допускаются аргументы INOUT, и, конечно, IN и VARIADIC.

Важная особенность процедур состоит в том, что, в отличие от функций, в них можно использовать операции управления транзакциями (такие как START TRANSACTION, COMMIT, ROLLBACK).

При определении подпрограмм можно задать новые значения параметров сервера базы данных, которые будут установлены на время выполнения этой подпрограммы. Конечно, таким способом можно задавать только те параметры, которые допускают динамическое изменение с помощью оператора SET. После выхода из подпрограммы восстанавливаются значения параметров сервера, которые эти параметры имели перед входом в подпрограмму. Если, однако, такого указания в определении подпрограммы нет, но в теле использован оператор SET, то установленные этим оператором значения сохранятся и после выхода из подпрограммы.

Параметр PARALLEL определяет для оптимизатора возможность использования параллельных планов для запросов, содержащих определяемую функцию:

- UNSAFE указывает, что функцию можно использовать только в последовательных планах;
- SAFE указывает на возможность использования параллельных планов без ограничений;
- RESTRICTED ограничивает использование функции только головной (последовательной) частью параллельных планов.

При спецификации функции можно указать, с какими привилегиями она будет выполняться. Обычно функция выполняется с привилегиями той роли, от имени которой она вызывается (SECURITY INVOKER), однако можно указать, что функция должна выполняться с привилегиями роли, от имени которой функция была определена (SECURITY DEFINER). Второй вариант дает возможность предоставить контролируемый (кодом функции) доступ пользователей к объектам, которые не должны быть им доступны непосредственно, и таким образом ограничить возможности манипулирования этими объектами только теми операциями, которые реализованы в этих функциях. Эти вопросы обсуждаются в главе 19.

16.2. Процедурный язык PL/pgSQL

Особое место PL/pgSQL среди многочисленных процедурных языков, поддерживаемых системой PostgreSQL, связано с тем, что этот язык очень хорошо интегрирован с SQL.

Непосредственным предшественником языка PL/pgSQL принято считать язык PL/SQL, применяемый в СУБД Oracle [49]. По своему синтаксису оба эти языка

относятся к семейству языков программирования, родоначальником которого можно считать Pascal, а другими представителями — Modula и Ada. Неформально, принадлежность к этому семейству выражается в том, что для выделения структурных единиц программного кода не используются фигурные скобки, как в языках программирования, входящих в семейство языка C.

Отметим, однако, что в языке PL/pgSQL отсутствует понятие *пакета* (package), унаследованное PL/SQL из языка Ada и позволяющее объединять несколько функций и процедур в один программный модуль с общими переменными.

В то же время PL/pgSQL (как и PL/SQL) содержит развитые средства интеграции с SQL и приспособлен для работы с объектами базы данных. В качестве предшественников этой части языка можно назвать группу языков «четвертого поколения» (4GL). Среди языков этой группы был широко известен язык Natural, реализованный в СУБД ADABAS [56].

Этот раздел кратко иллюстрирует основные конструкции языка PL/pgSQL. Цель этого раздела — облегчить ориентацию в документации. Изложение ни в какой мере не является исчерпывающим.

Некоторые из примеров, приведенных в этом разделе, полезны только для иллюстрации конструкций языка. Действия, выполняемые функциями в таких примерах, могут быть записаны проще и эффективнее без использования хранимых функций или как функции SQL (а не PL/pgSQL).

16.2.1. Структурные конструкции языка PL/pgSQL

Тело любой функции, написанной на языке PL/pgSQL, оформляется как *блок*, содержащий:

1) необязательный раздел описаний локальных переменных, используемых в этом блоке (начинается ключевым словом DECLARE);
2) раздел, содержащий выполнимые операторы (начинается ключевым словом BEGIN);
3) необязательный раздел, описывающий обработку исключительных ситуаций (начинается ключевым словом EXCEPTION).

Весь блок завершается ключевым словом END. Все операторы, в том числе блоки, завершаются точкой с запятой.

Пример, близкий к минимальному, может выглядеть следующим образом:

```
demo=# CREATE OR REPLACE FUNCTION hello(p text) RETURNS text
LANGUAGE plpgsql AS $$
DECLARE
    v text;
BEGIN
    v := 'Hello, ';
    RETURN v || p || '!';
END;
$$;
CREATE FUNCTION
demo=# SELECT hello('world');
     hello
---------------
 Hello, world!
(1 row)
```

Кроме структуры блока, этот пример показывает определение локальной переменной, оператор присваивания, вывод результата функции и ее использование в запросе. Локальные переменные могут иметь любые типы, определенные в SQL, в том числе любые типы, определенные пользователем.

В языке PL/pgSQL можно использовать вложенные блоки и составные операторы (блоки без раздела описаний). Вложенные блоки ограничивают области действия и области видимости локальных переменных.

Условный оператор записывается в одной из следующих форм:

```
IF условие THEN
    оператор; ...
ELSE
    оператор; ...
END IF;
```

```
IF условие THEN
    оператор; ...
END IF;
```

Оператор выбора альтернативных вариантов вычислений позволяет (так же как и выражение CASE в языке SQL) описывать ветвление на несколько альтернатив и может записываться в двух формах — с отдельными условиями на каждую альтернативу или с перечислением возможных значений выражения:

```
CASE
    WHEN условие-1 THEN
        оператор; ...
    WHEN условие-2 THEN
        оператор; ...
    ...
    ELSE
        оператор; ...
END CASE;
```

```
CASE выражение
    WHEN значение-1 THEN
        оператор; ...
    WHEN значение-2 THEN
        оператор; ...
    ...
    ELSE
        оператор; ...
END CASE;
```

Программистам, обычно работающим на языках семейства C (в том числе Java), следует обратить внимание на то, что и в условном операторе IF и в операторе выбора CASE заключать условие в скобки не нужно, а также на то, что составные операторы (после THEN и ELSE) не требуют никаких дополнительных скобок.

Подчеркнем, что оба рассмотренных оператора являются управляющими конструкциями и, в отличие от выражения CASE в языке SQL, не вырабатывают никакого значения. В то же время, конечно, ничто не мешает использовать выражение CASE (определенное в SQL) при конструировании выражений в подпрограммах.

В языке определено большое количество различных вариантов оператора цикла, среди которых есть как обычные для языков программирования, так и предназначенные для обработки множеств объектов базы данных, представленных запросами или курсорами.

В любом случае тело цикла обрамляется ключевыми словами LOOP и END LOOP, между которыми размещается составной оператор. Как и в других аналогичных конструкциях, дополнительные скобки не нужны.

```
LOOP
    оператор; ...
END LOOP;
```

Количество повторений цикла определяется заголовком. В языке предусмотрены следующие варианты:

- WHILE *условие*
  Заголовок WHILE вызывает повторение цикла до тех пор, пока *условие* вырабатывает истинное значение. Условие проверяется перед выполнением тела цикла.
- FOR *переменная* IN *начало* .. *конец* BY *шаг*
  Заголовок FOR задает цикл с переменной, пробегающей значения отрезка целочисленной арифметической прогрессии. В этом варианте цикла *переменная* не должна быть определена в разделе описаний переменных (она определяется заголовком).
- FOREACH *переменная* IN ARRAY *массив*
  Заголовок FOREACH задает выполнение цикла для каждого элемента массива, указанного в заголовке. Для многомерных массивов можно задать размерность, по которой выполняется цикл.
- Цикл без заголовка повторяется бесконечно.

В любом случае для выхода из цикла можно использовать оператор EXIT WHEN *условие*. Этот оператор является единственным способом выхода из цикла без заголовка.

Оператор CONTINUE прекращает текущую итерацию цикла и выполняет переход к началу следующей итерации.

Завершая краткий обзор управляющих конструкций языка PL/pgSQL, отметим, что не стоит искать в документации оператор безусловного перехода GO TO, потому что в языке такого оператора нет.

16.2.2. Работа с объектами базы данных

В качестве операторов в хранимых функциях и процедурах, написанных на языке PL/pgSQL, можно использовать операторы SQL. Если в выражениях, встречающихся в таком операторе, используются параметры функции или локальные переменные, то эти переменные подставляются как параметры подготовленного оператора SQL.

В действительности интерпретатор PL/pgSQL не вычисляет никакие выражения — вместо этого формируется неявный оператор SQL. Это, с одной стороны, гарантирует идентичность системы типов данных, с другой — приводит к появлению накладных расходов на формирование и обработку этих операторов.

Если оператор SQL возвращает одну строку, то значения, содержащиеся в этой строке, можно записать в переменные с помощью предложения INTO. Для оператора SELECT это предложение размещается сразу после списка выбираемых значений, для всех остальных операторов манипулирования данными (INSERT, UPDATE, DELETE с предложением RETURNING) — последним предложением, т. е. тоже после списка возвращаемых значений.

```
demo=# CREATE OR REPLACE FUNCTION air_city(a_code text) RETURNS text
LANGUAGE plpgsql AS $$
DECLARE
    v text;
BEGIN
    SELECT city
    INTO v
    FROM airports
    WHERE airport_code = a_code;
    RETURN v;
END;
$$;
CREATE FUNCTION
```

```
demo=# SELECT air_city('SVO');
 air_city
----------
 Москва
(1 row)
```

Важно подчеркнуть, что приведенный выше и в следующих примерах код иллюстрирует применение конструкций языка PL/pgSQL (в этом примере — предложения INTO), но не рекомендуемый стиль программирования. Функцию, возвращающую в точности такой же результат, как в этом примере, можно записать более компактно:

```
demo=# CREATE OR REPLACE FUNCTION air_city(a_code text) RETURNS text
LANGUAGE plpgsql AS $$
BEGIN
    RETURN (SELECT city
            FROM airports
            WHERE airport_code = a_code);
END;
$$;
```

Если же записать эту функцию на SQL (а не на PL/pgSQL), то код получится еще более компактным. Предложение INTO полезно в тех случаях, когда получаемые значения используются в других операторах в той же функции.

Если после ключевого слова INTO указано STRICT, то выполняемый оператор SQL должен возвращать ровно одну строку. Если оператор не возвращает ни одной строки или возвращает больше одной, возбуждается исключительная ситуация. Если STRICT не указано, то в переменные записываются значения из первой строки или значения NULL, если не возвращено ни одной строки.

Количество строк результата выполнения любого оператора SQL (не важно, с предложением INTO или без него) можно узнать с помощью оператора GET CURRENT DIAGNOSTICS. В частности, этот оператор можно использовать для получения значения ROW_COUNT — количества строк, возвращенных последним запросом. Кроме этого, в каждой функции предопределена локальная переменная FOUND типа boolean, которая указывает, была ли выбрана (хотя бы одна) строка последним оператором SQL, который может возвращать строки.

Результатом выполнения хранимой функции, как указано выше, может быть множество строк. Такие множества, полученные при вызове функций, можно использовать в предложении FROM наравне с таблицами и представлениями. Для задания имен столбцов такого множества можно использовать определение типа.

Наиболее элегантный способ обработки результатов операторов SQL, возвращающих несколько строк, состоит в использовании специальной формы заголовка цикла FOR. Следующий пример иллюстрирует применение такого цикла в функции для возвращения в качестве результата множества строк предварительно определенного составного типа. Конечно, цикл для обработки запроса можно применять и в том случае, если функция возвращает только одно значение или вовсе не возвращает значений.

```
demo=# CREATE TYPE airplane AS (code char(3), model text);
CREATE TYPE
demo=# CREATE OR REPLACE FUNCTION plane_set() RETURNS SETOF airplane
LANGUAGE plpgsql AS $$
DECLARE
    v record;
BEGIN
    FOR v IN SELECT * FROM aircrafts
    LOOP
        RETURN NEXT ROW(v.aircraft_code, v.model)::airplane;
    END LOOP;
END;
$$;
CREATE FUNCTION
demo=# SELECT * FROM plane_set();
 code |         model
------+---------------------
 773  | Боинг 777-300
 763  | Боинг 767-300
 SU9  | Сухой Суперджет-100
 320  | Аэробус A320-200
 321  | Аэробус A321-200
 319  | Аэробус A319-100
 733  | Боинг 737-300
 CN1  | Сессна 208 Караван
 CR2  | Бомбардье CRJ-200
(9 rows)
```

Как и в предыдущем примере, код иллюстрирует применение конструкций языка. Функцию можно записать более компактно, не используя оператор цикла и не записывая явно обработку каждой строки результата:

```
demo=# CREATE OR REPLACE FUNCTION plane_set() RETURNS SETOF airplane
LANGUAGE plpgsql AS $$
BEGIN
   RETURN QUERY
      SELECT ROW(aircraft_code, model)::airplane
      FROM aircrafts;
END;
$$;
```

Синтаксически допустимо такую функцию использовать и в списке значений SELECT, однако результат может быть не вполне очевидным, если в списке значений имеется более одной такой функции, и в разных версиях PostgreSQL результат может быть разным. Кроме этого, в этом случае будет отличаться и формат результата — выводится одна колонка типа record, что не всегда удобно для обработки:

```
demo=# SELECT plane_set();
          plane_set
----------------------------
 (773,"Боинг 777-300")
 (763,"Боинг 767-300")
 (SU9,"Сухой Суперджет-100")
 (320,"Аэробус A320-200")
 (321,"Аэробус A321-200")
 (319,"Аэробус A319-100")
 (733,"Боинг 737-300")
 (CN1,"Сессна 208 Караван")
 (CR2,"Бомбардье CRJ-200")
(9 rows)
```

Другой, более низкоуровневый способ обработки результатов запросов, возвращающих несколько строк, использует понятие *курсора*, представляющего собой объект базы данных, содержащий частично выполненный запрос.

В языке PL/pgSQL для манипулирования курсорами используются переменные, параметры и результаты функций типа refcursor. Для того чтобы начать работу с курсором, необходимо открыть его оператором OPEN для выполнения запроса. Открытый курсор можно возвращать как результат функции и принимать его в качестве параметра.

Для получения данных из курсора используется оператор FETCH, а для изменения текущей позиции в множестве строк, составляющих результат выполнения запроса, применяется оператор MOVE. Курсор можно явно закрыть оператором CLOSE, чтобы использовать его для выполнения другого запроса. Однако при завершении транзакции все открытые в ней курсоры закрываются автоматически.

Кроме последовательного считывания данных в прямом направлении, возможно движение и в обратном направлении, а также установка в позицию любой строки по ее номеру или перемещение на заданное количество строк от текущей. Заметим, что заголовок цикла FOR для чтения результатов запроса также создает неявный курсор. Такой курсор считывается всегда в прямом направлении, никакие другие операции с ним невозможны.

Следующая функция показывает наиболее простое использование курсора:

```
demo=# CREATE OR REPLACE FUNCTION plane_set_cur()
RETURNS SETOF airplane
LANGUAGE plpgsql AS $$
DECLARE
    cur refcursor;
    v record;
BEGIN
    OPEN cur FOR SELECT * FROM aircrafts;
    LOOP
        FETCH NEXT FROM cur INTO v;
        EXIT WHEN NOT FOUND;
        RETURN NEXT ROW(v.aircraft_code, v.model)::airplane;
    END LOOP;
    CLOSE cur;
END;
$$;
CREATE FUNCTION
```

```
demo=# SELECT * FROM plane_set_cur();
 code |         model
------+----------------------
 773  | Боинг 777-300
 763  | Боинг 767-300
 SU9  | Сухой Суперджет-100
 320  | Аэробус A320-200
 321  | Аэробус A321-200
 319  | Аэробус A319-100
 733  | Боинг 737-300
 CN1  | Сессна 208 Караван
 CR2  | Бомбардье CRJ-200
(9 rows)
```

Во многих языках программирования применение курсоров является единственным способом обработки результатов запросов.

16.2.3. Динамический SQL

Операторы SQL можно формировать динамически как значения текстовых выражений. Такие операторы могут принимать параметры, значения которых передаются из переменных PL/pgSQL. Позиции в запросе, в которые необходимо подставить значения параметров, обозначаются псевдопеременными $1, $2 и т. д. Для выполнения динамического SQL используется оператор EXECUTE. В нем предложение USING указывает переменные функции, которые надо подставить в SQL, а предложение INTO, как и в операторах SQL, указывает, куда поместить результаты.

```
demo=# CREATE OR REPLACE FUNCTION air_city_dyn(a_code text)
RETURNS text
LANGUAGE plpgsql AS $$
DECLARE
    v text;
    query_text text :=
        'SELECT city FROM airports WHERE airport_code = $1';
BEGIN
    EXECUTE query_text
    INTO v
    USING a_code;
    RETURN v;
END;
$$;
CREATE FUNCTION

demo=# SELECT air_city_dyn('SVO');
 air_city_dyn
--------------
 Москва
(1 row)
```

Динамический SQL можно также использовать в заголовках циклов и в курсорах. В приведенных выше примерах можно задавать запрос в виде выражений, вырабатывающих текстовые строки:

```
FOR v IN EXECUTE 'SELECT * FROM aircrafts'
```

или

```
OPEN cur FOR EXECUTE 'SELECT * FROM aircrafts';
```

Применение динамического SQL неизбежно приводит к появлению дополнительных накладных расходов на предварительную обработку (компиляцию), однако в системе PostgreSQL доля дополнительных расходов по сравнению с расходами на обработку статического SQL значительно ниже, чем в других СУБД. Дело в том, что оптимизация запросов выполняется в PostgreSQL относительно поздно, после того как становятся известны значения параметров запроса как для динамического, так и для статического SQL. Поэтому формирование запросов оказывается не только мощным, но и эффективным инструментом.

Значения переменных PL/pgSQL можно передавать, преобразуя их в строковые константы или с помощью механизма псевдопеременных $*n*. Последний способ предпочтительнее, т. к. предотвращает атаки путем внедрения SQL-кода, однако его можно применять только в тех случаях, когда в SQL необходимы значения атрибутов. Псевдопеременные нельзя использовать, например, для

передачи имен таблиц или выражений. В таких случаях необходимо конкатенировать текстовые значения и, соответственно, принимать дополнительные меры для защиты от возможных атак, используя специальные функции.

- Функция quote_ident представляет свой аргумент как правильно записанное имя идентификатора, при необходимости окружая значение двойными кавычками и удваивая встречающиеся внутри двойные кавычки. Эту функцию целесообразно использовать для подстановки имен таблиц, колонок и т. п.
- Функция quote_literal возвращает значение своего аргумента как правильно записанную строковую константу в одиночных кавычках.
- Функция quote_nullable работает как предыдущая, но если аргумент имеет значение NULL, то возвращается строка «NULL».

Применение этих функций гарантирует, что значения переменных не будут рассматриваться как часть SQL-кода. Попытка внедрения приведет к тому, что будет сформирован некорректный оператор SQL, выполнение которого вызовет ошибку, или включаемый код будет превращен в текстовую константу. В том и другом случае включаемый злоумышленником код не будет выполнен.

Альтернативный способ достижения тех же целей предоставляет функция format, похожая на функцию sprintf языка программирования C. Первый параметр этой функции задает формат, в который подставляются значения остальных параметров в соответствии с типом, указанным в формате. Следующий пример показывает, каким образом можно использовать типы «I» и «L», по своему действию эквивалентные функциям quote_ident и quote_literal:

```
demo=# SELECT format(
    'SELECT * FROM %I WHERE name = %L',
    'table name', 'a value'
);
                        format
-----------------------------------------------
 SELECT * FROM "table name" WHERE name = 'a value'
(1 row)
```

16.2.4. Обработка исключительных ситуаций

Если при выполнении функции возникает исключительная ситуация, препятствующая продолжению работы, обычно выполнение функции прекращается и сообщение об ошибке возвращается клиенту. Однако если в блоке PL/pgSQL

имеется раздел EXCEPTION, в котором предусмотрена обработка этой ситуации, то выполняются операторы этого обработчика, и затем выполнение блока может быть завершено нормально или аварийно в зависимости от обработчика ошибки.

При возникновении любой исключительной ситуации система PostgreSQL выполняет откат базы данных в то состояние, которое она имела в момент начала выполнения наименьшего блока, в котором предусмотрена обработка этой исключительной ситуации и внутри которого она возникла. Если такого блока нет, то выполняется откат транзакции, в которой возникла исключительная ситуация.

Тип ошибки задается предложением WHEN *условие* THEN, за которым следуют операторы, выполняющие обработку ошибки. В качестве условия может быть указано имя исключительной ситуации, условие на код ошибки или ключевое слово OTHERS для обработки всех ситуаций, не предусмотренных для предшествующих обработчиков в этом блоке. Следующий пример иллюстрирует обработку исключительной ситуации:

```
demo=# CREATE OR REPLACE FUNCTION air_city(a_code text) RETURNS text
LANGUAGE plpgsql AS $$
DECLARE
    v text;
BEGIN
    SELECT city
    INTO STRICT v
    FROM airports
    WHERE airport_code = a_code;
    RETURN v;
EXCEPTION
    WHEN no_data_found THEN
        RETURN '-- Invalid airport code --';
    WHEN OTHERS THEN
        RAISE;
END;
$$;
CREATE FUNCTION

demo=# SELECT air_city('XXX');
          air_city
----------------------------
 -- Invalid airport code --
(1 row)
```

В этом примере в предложении INTO добавлено ключевое слово STRICT, требующее, чтобы в результате выполнения запроса была найдена ровно одна строка. Поскольку в вызове функции указан неверный код аэропорта, возникает

исключительная ситуация, предусмотренная первым обработчиком, который возвращает константу и завершает функцию нормально. Конечно, такой возврат сообщения об ошибке вместо результата нельзя считать хорошим стилем программирования.

Второй обработчик возбуждается при возникновении любой другой ошибки и возбуждает ту же самую ошибку оператором RAISE. Такой обработчик не имеет никакого смысла, потому что он стирает информацию о том, где в действительности произошла ошибка (которая могла произойти в любой функции, вызываемой из блока, где находится обработчик ошибок).

При обработке исключительных ситуаций можно использовать предопределенные переменные SQLSTATE (код ошибки) и SQLERRM (сообщение об ошибке). Можно получить и более детальную информацию с помощью оператора GET STACKED DIAGNOSTICS. Оператор RAISE позволяет выводить информационные сообщения, предупреждения, а также возбуждать исключительные ситуации, в том числе определенные пользователем.

В документации PostgreSQL указано, что блоки, содержащие обработчики исключительных ситуаций, требуют для выполнения значительно больше вычислительных ресурсов (что связано с неявной установкой точки сохранения в начале таких блоков). Поэтому, как правило, целесообразно предотвращать возбуждение исключительных ситуаций, выполняя дополнительные проверки с помощью условных операторов. В нашем примере такой же результат можно получить, проверяя наличие строк в ответе на запрос вместо применения ключевого слова STRICT:

```
demo=# CREATE OR REPLACE FUNCTION air_city(a_code text) RETURNS text
LANGUAGE plpgsql AS $$
DECLARE
    v text;
BEGIN
    SELECT city
    INTO v
    FROM airports
    WHERE airport_code = a_code;
    IF NOT FOUND THEN
        RETURN '-- Invalid airport code --';
    END IF;
    RETURN v;
END;
$$;
```

При использовании обработчиков исключительных ситуаций необходимо учитывать, что при возникновении исключительной ситуации откат базы данных

выполняется в любом случае независимо от того, завершает обработчик функцию нормально или нет. Если бы приведенная выше функция air_city выполняла операции модификации базы данных, то в варианте с обработкой исключительной ситуации эти изменения были бы потеряны, а в варианте с проверкой значения переменной FOUND сохранились бы при любом значении аргумента, потому что исключительная ситуация no_data_found не возбуждается.

Можно сформулировать несколько рекомендаций по применению средств обработки исключительных ситуаций, которые имеет смысл учитывать при проектировании прикладной системы, использующей хранимые подпрограммы.

- При обработке исключительных ситуаций необходимо сохранять системную диагностику, для того чтобы облегчить выявление причин, вызвавших возбуждение такой ситуации.
- Обработка исключительных ситуаций необходима, для того чтобы скрыть от конечного пользователя системные сообщения об ошибках. Это можно делать как в хранимых подпрограммах, так и в клиентской части приложения, однако перехват на уровне сервера баз данных (в хранимой подпрограмме) дает больше возможностей для сбора диагностической информации. Конечно, такая информация должна анализироваться службой сопровождения, а не конечным пользователем.
- Целесообразно включать обработку исключительных ситуаций в подпрограммы, которые выполняются как часть одного тяжеловесного процесса, состоящего из нескольких относительно независимых частей. Например, обновление нескольких материализованных представлений может продолжаться, даже если при обновлении одного из них возникла ошибка.
- Целесообразно включать обработку исключительных ситуаций в тех случаях, когда вероятность возникновения исключительной ситуации относительно велика (например, при вводе данных с пользовательского терминала могут появляться дублирующие сообщения).

Обработчики исключительных ситуаций, в отличие от основного выполняемого кода, могут перехватывать ошибки, возникающие не только в блоке, в котором размещен обработчик, но и в любом динамически вложенном, в том числе в других функциях, выполнение которых инициировано из этого блока. Поэтому обычно нецелесообразно обрабатывать исключительные ситуации в подпрограммах, которые вызываются из подпрограмм, где такая обработка уже предусмотрена.

По-видимому, целесообразно обрабатывать исключительные ситуации на том уровне, где такая обработка может дать диагностику, которую легко интерпретировать в терминах приложения. Если же такая диагностика невозможно, то, вероятно, лучше вовсе не перехватывать ошибки.

16.3. Функции и процедуры на языке SQL

Тело функции или процедуры, написанной на языке SQL, представляет собой последовательность операторов SQL, разделенных точкой с запятой. Последний оператор должен сформировать результат функции. Если результат функции специфицирован не как таблица (TABLE или SETOF), то возвращается первая строка результата последнего оператора. Для процедур, конечно, результат не требуется.

Для простых подпрограмм запись тела на языке SQL обычно получается более компактной, чем на любом другом языке, доступном в системе PostgreSQL, включая язык PL/pgSQL. Приведем несколько примеров записи функций из раздела 16.2 на языке SQL.

Функция, возвращающая одно скалярное значение:

```
demo=# CREATE OR REPLACE FUNCTION hello(p text) RETURNS text
LANGUAGE sql AS $$
    SELECT 'Hello, ' || p || '!';
$$;
CREATE FUNCTION

demo=# select hello('world');
     hello
---------------
 Hello, world!
(1 row)
```

Функция, возвращающая множество строк составного типа:

```
demo=# CREATE OR REPLACE FUNCTION plane_set() RETURNS SETOF airplane
LANGUAGE sql AS $$
    SELECT ROW(aircraft_code, model)::airplane
    FROM aircrafts;
$$;
CREATE FUNCTION

demo=# SELECT * FROM plane_set();
```

```
 code |         model
------+---------------------
 773  | Боинг 777-300
 763  | Боинг 767-300
 SU9  | Сухой Суперджет-100
 320  | Аэробус A320-200
 321  | Аэробус A321-200
 319  | Аэробус A319-100
 733  | Боинг 737-300
 CN1  | Сессна 208 Караван
 CR2  | Бомбардье CRJ-200
(9 rows)
```

Если в спецификации функции на языке SQL нет указания SETOF (или TABLE), то функция возвращает первую строку результата. Использование этого факта, конечно, не может рассматриваться как рекомендуемая практика программирования.

```
demo=# CREATE OR REPLACE FUNCTION plane_row() RETURNS airplane
LANGUAGE sql AS $$
    SELECT ROW(aircraft_code, model)::airplane
    FROM aircrafts;
$$;
CREATE FUNCTION
```

```
demo=# SELECT * FROM plane_row();
 code |     model
------+---------------
 773  | Боинг 777-300
(1 row)
```

Применение функций, написанных на языке SQL, предоставляет возможность для вынесения запросов из кода приложения, а также может быть полезно как инструмент ограничения доступа, как описано в главе 19. Однако некоторые виды функций, например функции триггеров, на SQL писать нельзя.

Важная особенность функций на языке SQL состоит в том, что в ряде случаев оптимизатор может подставлять тела таких функции в запрос, в котором они вызываются. Для этого функция должна содержать единственный оператор SELECT и должна быть объявлена как SECURITY INVOKER. Кроме того, функция, возвращающая множество строк, должна быть объявлена как STABLE или IMMUTABLE, а скалярным функциям нельзя обращаться к таблицам и содержать предложения FROM, GROUP BY, ORDER BY и т. п. Точные правила включают и некоторые другие ограничения. Однако если все условия соблюдены, то выполняется совместная оптимизация запроса и тела функции, что обычно приводит к получению более эффективного плана выполнения запроса.

16.4. Итоги главы

В этой главе кратко представлены основные свойства и характеристики хранимых функций и процедур. Код таких подпрограмм может быть написан на любом из нескольких языков программирования, поддерживаемых для системы PostgreSQL. Более детально рассмотрены основные конструкции языка PL/pgSQL, в том числе основные управляющие структуры и средства взаимодействия с SQL.

16.5. Упражнения

Упражнение 16.1. Напишите на языке PL/pgSQL функцию, возвращающую все данные, относящиеся к одному бронированию, номер которого задан параметром.

Упражнение 16.2. Напишите на языке SQL (не на PL/pgSQL) функцию, эквивалентную функции из упражнения 16.1.

Упражнение 16.3. Напишите процедуру на языке PL/pgSQL, создающую новое бронирование в указанный день из заданного пункта отправления в заданный пункт назначения не более чем с двумя пересадками.

Упражнение 16.4. Напишите процедуру, добавляющую нового пассажира к указанному бронированию при условии, что на рейсе есть свободные места.

Упражнение 16.5. Исследуйте, как происходит откат базы данных при возникновении исключительной ситуации.

Для этого напишите функцию, которая выполняет некоторые изменения, затем выполняет другие изменения внутри блока, обрабатывающего исключительную ситуацию, а затем в некоторых случаях выполняет оператор, вызывающий возникновение исключительной ситуации внутри блока с обработчиком или вне его.

Проверьте состояние базы данных при нормальном выполнении, при нормальном выходе из обработки исключительной ситуации, при отсутствии обработки исключительной ситуации.

Упражнение 16.6. Повторите пример, приведенный в разделе 16.1, переписав функцию never на языке SQL, и объясните полученный результат.

Глава 17

Расширяемость PostgreSQL

Одной из главных отличительных особенностей системы PostgreSQL является ее расширяемость: возможность включения средств поддержки новых классов приложений, новых типов данных и т. п. Для таких расширений необходима возможность выполнения кода, написанного пользователем, на сервере баз данных. Многие системы обеспечивают эти возможности с помощью хранимых подпрограмм (процедур или функций). Механизм хранимых подпрограмм, имеющийся в системе PostgreSQL, обсуждается в главе 16.

Этого тем не менее недостаточно, для того чтобы система могла считаться по-настоящему расширяемой. Необходимо также, чтобы расширения можно было использовать наравне со встроенными средствами системы и без помех для работы внутренних механизмов системы (например, управления транзакциями и оптимизации запросов). Для выполнения этих требований нужны более развитые механизмы, чем просто возможность добавления хранимых процедур и функций.

17.1. Пользовательские агрегаты

Агрегаты вычисляют значение на основе выражений, полученных либо на основе данных из всех кортежей отношения, либо из кортежей, объединяемых в группы предложением GROUP BY. В системе PostgreSQL определены не только агрегаты, предусмотренные стандартом SQL (count, sum, avg, min, max), но и многие другие, а также имеется возможность определять новые агрегаты.

Определение агрегата включает несколько функций, из которых наиболее важны две:

- функция, накапливающая информацию по каждой входной записи,
- функция, формирующая окончательный результат.

Приведем пример определения агрегата, который строит список констант, полученных из значений некоторой колонки в таблице. Предполагается, что эта колонка имеет тип text или может быть преобразована к этому типу. Для преобразования значения атрибута в запись константы использована функция quote_literal. Элементы списка разделяются запятыми, а весь список заключается в круглые скобки.

```
demo=# CREATE OR REPLACE FUNCTION build_list_next(
    agg_state text,
    listitem text
) RETURNS text
LANGUAGE plpgsql AS $$
BEGIN
    RETURN agg_state || ',' || quote_literal(listitem);
END;
$$;
CREATE FUNCTION
demo=# CREATE OR REPLACE FUNCTION build_list_final(agg_state text)
RETURNS text
LANGUAGE plpgsql AS $$
BEGIN
    RETURN '(' || substr(agg_state, 2) || ')';
END;
$$;
CREATE FUNCTION
demo=# CREATE AGGREGATE build_list(list_item text) (
    STYPE = text,
    SFUNC = build_list_next,
    FINALFUNC = build_list_final,
    INITCOND = ''
);
CREATE AGGREGATE
```

Далее следует пример применения этого агрегата:

```
demo=# SELECT build_list(aircraft_code)
FROM aircrafts;
                        build_list
-----------------------------------------------------------
 ('773','763','SU9','320','321','319','733','CN1','CR2')
(1 row)
```

Следующий пример агрегата показывает, каким образом можно использовать псевдотипы anyelement и anyarray. В этом агрегате не нужна функция, возвращающая окончательный результат, потому что он формируется в переменной, содержащей состояние агрегата после обработки каждой записи. Первый параметр включается в массив результата, если второй параметр имеет истинное значение.

```
demo=# CREATE OR REPLACE FUNCTION array_agg_next(
    agg_sta anyarray,
    val anyelement,
    b boolean
) RETURNS anyarray
LANGUAGE plpgsql AS $$
BEGIN
    IF b THEN
        agg_sta := agg_sta || ARRAY[val];
    END IF;
    RETURN agg_sta;
END;
$$;
CREATE FUNCTION
demo=# CREATE AGGREGATE array_agg(anyelement, boolean) (
    STYPE = anyarray,
    SFUNC = array_agg_next
);
CREATE AGGREGATE
```

Следующий запрос группирует типы самолетов в два массива в зависимости от предельной дальности полета и размещает их в двух колонках одной строки результата:

```
demo=# SELECT
    array_agg(aircraft_code, range <  5500) AS short_distance,
    array_agg(aircraft_code, range >= 5500) AS Long_distance
FROM aircrafts;
  short_distance   |     long_distance
-------------------+-----------------------
 {SU9,733,CN1,CR2} | {773,763,320,321,319}
(1 row)
```

В этом примере использован полиморфизм: имя агрегата совпадает с именем встроенного агрегата array_agg, но, поскольку список параметров отличается, сервер может выбрать подходящую функцию.

Тот же самый агрегат можно использовать для обработки данных другого типа; в следующем примере строятся два массива числовых значений атрибута range:

```
demo=# SELECT
    array_agg(range, range < 5500) AS short_distance,
    array_agg(range, range >=5500) AS Long_distance
FROM aircrafts;
    short_distance     |        long_distance
-----------------------+----------------------------
 {3000,4200,1200,2700} | {11100,7900,5700,5600,6700}
(1 row)
```

Еще один запрос строит массив всех имеющихся дальностей полета:

```
demo=# SELECT array_agg(range) AS ranges
FROM aircrafts;
                      ranges
-------------------------------------------------
 {11100,7900,3000,5700,5600,6700,4200,1200,2700}
(1 row)
```

В этом запросе отсутствует второй аргумент агрегата, поэтому используется системная функция агрегирования. Таким способом, конечно, можно строить массив значений любого типа; в примере использован атрибут range только потому, что результат получается небольшого размера.

17.2. Типы данных, операторы и классы операторов

Определение типа данных включает спецификации того, какие значения относятся к этому типу. Для того чтобы говорить о том, что понятие типа данных примерно соответствует теоретическому понятию абстрактного типа, необходимо также определить, какие действия можно выполнять с экземплярами этого типа. Однако в системе PostgreSQL функции и операторы определяются отдельно от типов данных, при этом операторы и функции могут быть полиморфными и применяться к разным типам. Тем не менее для каждого типа данных операторы или функции должны быть определены, поскольку значения, с которым нельзя делать никакие действия, вряд ли имеют смысл.

Основой системы типов в СУБД PostgreSQL являются *базовые типы*, для которых формат хранения в базе данных и операции определяются вне языка SQL, обычно на языке программирования C. Значения базовых типов всегда рассматриваются как скалярные. Для того чтобы создать новый базовый тип, необходимо определить, как значения этого типа будут записываться в структурах хранения базы данных, что требует от разработчика детального знания внутренних структур хранения, принятых в PostgreSQL, в частности структуры блоков данных. Как минимум определение типа должно включать имена двух функций, выполняющих преобразование внутреннего формата хранения значения в строку и обратное преобразование строки, содержащей значение, во внутренний формат.

Заметим, что преобразование во внутренний формат может оказаться довольно сложным, т. к. для его выполнения может потребоваться синтаксический

анализ входной строки и диагностика возможных ошибок. Поскольку для регистрации функций в базе данных необходимо, чтобы типы аргументов уже существовали, определение типа выполняется в несколько этапов: сначала в базу данных заносится только имя типа данных, затем создаются необходимые функции (возможно, не только ввода и вывода значения), и после этого функции привязываются к определению типа:

```
CREATE TYPE new_base_type;
CREATE FUNCTION new_base_in(text) RETURNS new_base_type
    ...;
CREATE FUNCTION new_base_out(new_base_type) RETURNS text
    ...;
CREATE TYPE new_base_type (
    INPUT = new_base_in,
    OUTPUT = new_base_out
);
```

Для каждого базового типа автоматически определяется тип массива, состоящего из элементов этого базового типа.

Другие возможности для расширения системы типов реализуются в терминах языка SQL.

Типы доменов представляют собой базовые типы с дополнительными ограничениями, т. е. множество возможных значений домена — подмножество значений базового типа, удовлетворяющее ограничениям домена. Эти ограничения проверяются, когда значение базового типа преобразуется в значение домена, однако они не ограничивают применение операций и функций над доменом значениями базового типа, удовлетворяющими ограничениям. Например, если определить домены длины и веса над базовым типом вещественных чисел, то операция сложения веса с длиной все равно будет синтаксически допустима и будет выполнена, хотя результат вряд ли имеет смысл.

Составной тип представляет собой набор именованных атрибутов, каждый из которых должен иметь ранее определенный тип. Составной тип автоматически создается для каждой таблицы или представления и включает набор столбцов таблицы или представления. Можно также определить составной тип, не создавая таблицу или представления, с помощью оператора CREATE TYPE:

```
demo=# CREATE TYPE linear AS (
   value numeric,
   unit text
);
CREATE TYPE
```

```
demo=# CREATE TABLE measurements AS
    SELECT (a.value, b.unit)::linear length,
           (c.value, d.unit)::linear width
    FROM
        (VALUES (0.1), (1), (100))     a(value),
        (VALUES ('см'), ('м'), ('км')) b(unit),
        (VALUES (0.1), (1), (100))     c(value),
        (VALUES ('см'),( 'м'), ('км')) d(unit);
SELECT 81
```

Атрибуты составного типа могут иметь любой тип, известный в базе данных на момент создания типа. В частности, атрибуты могут быть массивами, что создает возможность для построения сложных иерархических типов данных.

После того как составной тип определен, его можно использовать в качестве параметра или результата при описании функций, а также в качестве типов значений атрибутов составных типов и таблиц. Можно сказать, что определение типа вместе с набором функций, обрабатывающих значения этого типа, представляет некоторый абстрактный тип данных, однако такое понятие в системе PostgreSQL специально не выделяется.

```
demo=# CREATE FUNCTION to_meters(a linear) RETURNS numeric
LANGUAGE sql AS $$
    SELECT a.value * CASE a.unit
                         WHEN 'см' THEN 0.01
                         WHEN 'м'  THEN 1
                         WHEN 'км' THEN 1000
                     END;
$$;
CREATE FUNCTION
```

Кроме функций, можно определять *операции*, которые бывают унарными (правыми и левыми) и бинарными. В действительности любое определение операции ссылается на функцию, которая реализует эту операцию, поэтому можно сказать, что операции дают более привлекательный способ записи выражений, использующих функции с одним или двумя аргументами.

```
demo=# CREATE FUNCTION linear_eq(a linear, b linear) RETURNS boolean
LANGUAGE plpgsql AS $$
BEGIN
    RETURN to_meters(a) = to_meters(b);
END;
$$;
CREATE FUNCTION
```

```
demo=# CREATE OPERATOR = (
    FUNCTION = linear_eq,
    LEFTARG = linear,
    RIGHTARG = linear
);
CREATE OPERATOR
```

```
demo=# SELECT *
FROM measurements
WHERE length = width
AND (length).unit != (width).unit;
  length   |  width
-----------+----------
 (0.1,км) | (100,м)
 (1,м)    | (100,см)
 (100,см) | (1,м)
 (100,м)  | (0.1,км)
(4 rows)
```

Кроме этого, для операций можно указывать дополнительные свойства, помогающие оптимизатору преобразовывать выражения и запросы, содержащие операции.

Для бинарных операций можно указывать свойство COMMUTATOR, значением которого является операция, вырабатывающая такой же результат, как исходная операция, после перестановки аргументов. Например, операции $+$ и $=$ являются коммутаторами сами для себя, а коммутатором для операции $>$ является операция $<$.

Для операций, результат которых является логическим значением, можно задавать операцию NEGATOR (вырабатывающую противоположное логическое значение), указывать селективность в операциях фильтрации и соединения, а также возможность их использования в алгоритмах соединения на основе хеширования и слияния.

Известно, что неточные оценки кардинальности оказываются одной из основных причин, приводящих к получению субоптимальных планов выполнения запроса. Указание оценок селективности дает возможность приблизить качество планов, содержащих операции, определенные пользователем, к качеству планов, содержащих только встроенные операции. В то же время необходимо заметить, что получение оценок высокого качества достаточно сложно, поэтому документация по системе PostgreSQL рекомендует указывать какую-либо из функций оценки, уже входящих в состав СУБД PostgreSQL.

17.3. Индексы

Высокая эффективность реляционных СУБД в значительной мере основана на использовании индексов. В системе PostgreSQL предусмотрены средства, позволяющие определить, какие индексы можно строить для пользовательских типов данных и как использовать эти индексы для проверки условий, встречающихся в запросах (в частности, в предложениях WHERE). Благодаря этим средствам поиск по значениям атрибутов, имеющих тип, определенный пользователем, может выполняться не менее эффективно, чем для встроенных типов данных.

В системе PostgreSQL возможность создания и использования индексов для типов данных, определенных пользователем, обеспечивается тем, что индексные структуры реализованы как обобщенные, пригодные для индексирования произвольных типов данных, удовлетворяющих требованиям выбранного типа индекса. Так, обобщенная реализация B-дерева требует, чтобы значения индексируемого типа данных были полностью упорядочены и чтобы для этого типа данных были определены бинарные предикаты $<$, $>$ и другие отношения неравенства.

Для других видов индексов требуется другой набор базовых операций и предикатов над типом данных. Например, для индексирования прямоугольников понадобятся предикаты для проверки вложенности и непустоты пересечения. Набор предикатов для одномерных числовых интервалов (который принято называть алгеброй Аллена [2; 4]) насчитывает 13 отношений. Конечно, для любого типа данных и для любого индекса должно быть определено отношение равенства.

В состав системы PostgreSQL включены следующие абстрактные (обобщенные) индексные структуры:

BTREE — одномерный упорядоченный индекс на основе B-деревьев;

HASH — одномерный индекс на основе хеширования;

GIST — пространственный индекс на основе сбалансированных деревьев;

SPGIST — многомерный индекс на основе несбалансированных деревьев;

GIN — инвертированный индекс, применяемый в системах текстового поиска;

BRIN — неплотный одномерный индекс для таблиц очень большого размера.

Более подробно эти структуры обсуждаются в разделе 11.2.

Для того чтобы создавать и использовать индексы одного из перечисленных видов для конкретного типа данных, необходимо привязать операторы, определенные для типа данных (например, операции сравнения) к абстрактной (обобщенной) индексной структуре. Такая привязка выполняется созданием *класса операторов* командой CREATE OPERATOR CLASS, которая относится к расширению SQL, специфическому для системы PostgreSQL. Какие именно операции необходимо задать для работы метода индексирования, зависит от типа индекса.

Так, для индекса на основе B-деревьев необходимо задать операции, реализующие предикаты сравнения $<$, $\leqslant$, $=$, $\geqslant$ и $>$. Это можно сделать по аналогии с созданным выше оператором равенства.

Также потребуется вспомогательная функция для сравнения значений:

```
demo=# CREATE FUNCTION linear_cmp(a linear, b linear) RETURNS integer
LANGUAGE plpgsql AS $$
BEGIN
    RETURN sign(to_meters(a) - to_meters(b))::integer;
END;
$$;
CREATE FUNCTION
```

После создания класса операторов значения будут упорядочиваться согласно определенным функциям сравнения (а не лексикографически, как по умолчанию упорядочиваются значения любых составных типов) и могут быть корректно проиндексированы:

```
demo=# CREATE OPERATOR CLASS linear_ops
DEFAULT FOR TYPE linear
USING btree AS
    OPERATOR 1 <,
    OPERATOR 2 <=,
    OPERATOR 3 =,
    OPERATOR 4 >=,
    OPERATOR 5 >,
    FUNCTION 1 linear_cmp(linear,linear);
CREATE OPERATOR
```

```
demo=# SELECT *
FROM measurements
ORDER BY length, width
OFFSET 5
LIMIT 8;
```

```
  length  |  width
----------+----------
 (0.1,см) | (0.1,км)
 (0.1,см) | (100,м)
 (0.1,см) | (1,км)
 (0.1,см) | (100,км)
 (1,см)   | (0.1,см)
 (1,см)   | (1,см)
 (1,см)   | (0.1,м)
 (1,см)   | (1,м)
(8 rows)

demo=# CREATE INDEX ON measurements(length);
CREATE INDEX

demo=# EXPLAIN (costs off)
SELECT *
FROM measurements
WHERE length = (1,'м')::linear;
                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using measurements_length_idx on measurements
   Index Cond: (length = '(1,м)'::linear)
(2 rows)
```

Типы данных могут быть семантически взаимосвязаны, как, например, числовые типы с разной точностью или с разным диапазоном возможных значений. Взаимосвязь типов в данном случае выражается в том, что значения разных типов можно проверять, например, на равенство или отношение порядка (больше или меньше). Для таких типов данных можно создать *семейство операторных классов*, в котором определяются операторные классы для каждого типа данных и операции сравнения значений из разных типов. В любой индексируемой колонке значения, конечно, все равно должны быть одного типа, но определение семейства операторных классов делает возможным использование индексов и в тех случаях, когда в условиях запроса значения из колонки сравниваются со значениями другого типа.

17.4. Другие инструменты расширения

В этом разделе кратко перечисляются средства расширения и модификации системы PostgreSQL, для применения которых требуется более глубокое знакомство с внутренней организацией системы. К таким средствам можно отнести:

- создание модулей расширений (extensions);
- подключение дополнительных процедурных языков;

- создание новых оберток для сторонних (внешних по отношению к базе данных PostgreSQL) данных;
- замена или модификация отдельных компонент системы (например, планировщика запросов);
- подключение альтернативных подсистем хранения данных (pluggable storage).

Этот список, конечно, не является исчерпывающим: система PostgreSQL развивается, появляются новые возможности и новые средства расширения.

17.4.1. Модули расширения

Предыдущие разделы показывают, что в большинстве случаев для расширения возможностей системы PostgreSQL необходимо создать несколько разнообразных объектов базы данных, например описания типов, набор функций и операторов. Для того чтобы скомпоновать все взаимосвязанные объекты и структуры, можно использовать понятие *расширения* (extension).

Для системы PostgreSQL каждое расширение описывается конфигурационным файлом, определяющим свойства этого расширения, и еще одним файлом, содержащим операторы SQL, создающие или модифицирующие объекты базы данных, входящие в расширение. Кроме этого, в состав расширения могут входить и другие файлы, например объектный код функций, написанных на процедурных языках, требующих компиляции.

После того как все файлы расширения размещены в специальном каталоге на сервере, где находится код СУБД, можно использовать операторы SQL CREATE EXTENSION, ALTER EXTENSION и DROP EXTENSION, для того чтобы подключить, изменить или удалить это расширение из базы данных.

Некоторые расширения входят в состав основной распространяемой версии системы PostgreSQL, но не подключаются автоматически при установке системы. Для подключения таких расширений достаточно выполнить команду CREATE EXTENSION, указав имя расширения в качестве аргумента.

Механизм расширений сам по себе не добавляет новые возможности, однако позволяет организовывать модули расширения, описывать действия, необходимые для смены версий расширения, обеспечивает согласованную обработку объектов, входящих в расширение, при создании резервных копий и выполнении других административных действий.

17.4.2. Обертки сторонних данных

Во многих современных приложениях необходимо совместно обрабатывать данные, размещенные в различных хранилищах, зачастую принадлежащих разным владельцам. Такие данные могут храниться под управлением различных систем хранения, основанным на разных моделях данных и реализующим различающиеся подмножества функций СУБД. Обработка таких данных обеспечивается созданием неоднородных распределенных систем, которые рассматриваются в главе 22.

В системе PostgreSQL возможность использования сторонних данных в запросах обеспечивается механизмом *оберток сторонних данных* (foreign data wrappers). Применение этого механизма также обсуждается в главе 22. Здесь мы кратко опишем, каким образом можно добавлять обертки для новых систем хранения данных, расширяя таким образом систему PostgreSQL. Подчеркнем, что речь здесь идет именно о новых типах систем и способах организации данных, а не о подключении новых наборов данных.

По существу, для того чтобы создать обертку для некоторого типа сторонних данных, необходимо написать несколько функций, определяющих для сервера PostgreSQL, каким образом эти данные обрабатывать. Возможности обработки сторонних данных зависят от того, какие из этих функций реализованы в обертке. Полный набор функций, которые могут быть включены в обертку, позволяет, в частности, выполнять следующие действия:

- полный просмотр;
- выполнение операций соединения на внешнем сервере;
- обновления сторонних данных;
- управление блокировками;
- взаимодействие с оптимизатором внешней системы;
- получение описания данных (импорт схемы).

Этот список не является полным. Конечно, не каждый тип внешнего сервера может обеспечить выполнение всех операций, которые можно определить в обертке. Например, файловая система обычно не может выполнять операции соединения. В подобных случаях обертка не сможет передавать операции на внешний сервер, поэтому в данном примере операция соединения должна быть выполнена локально после копирования данных с помощью операции сканирования.

17.4.3. Подключение новых процедурных языков

В системе PostgreSQL поддержка процедурных языков практически полностью отделена от ядра системы, поэтому добавление новых языков выполняется относительно несложно.

Для того чтобы выполнить подпрограмму, написанную на любом языке, кроме языка программирования C (на SQL или на любом процедурном языке), вызывается обработчик подпрограмм для языка, указанного в операторе CREATE FUNCTION, CREATE PROCEDURE или DO. Обработчик языка должен:

- найти нужную подпрограмму, используя справочную информацию из базы данных;
- если необходимо, найти скомпилированный модуль или передать текст подпрограммы в интерпретатор;
- обеспечить передачу аргументов подпрограммы, указанных в вызове (выраженных в терминах типов данных SQL), в формат, требуемый для подпрограммы (в типы данных языка программирования);
- обеспечить возврат результатов.

Для того чтобы подключить новый язык программирования, необходимо написать (на языке программирования C) обработчик для этого языка и, если необходимо, еще две функции для синтаксической проверки кода на новом языке и для использования этого языка в операторе DO.

После того как все необходимые модули готовы, для регистрации языка в базе данных PostgreSQL используется оператор CREATE LANGUAGE. Однако документация рекомендует упаковывать новый язык как расширение (extension), поэтому обычно команда CREATE LANGUAGE размещается внутри SQL-файла, входящего в состав расширения, а добавление языка в базу данных выполняется командой CREATE EXTENSION.

17.5. Итоги главы

В этой главе рассматриваются механизмы системы PostgreSQL, позволяющие расширить выразительные возможности языка SQL, определяя новые типы данных, функции и операции. Указание свойств функций и операций, а также средства создания индексов для типов данных, определенных пользователем,

позволяют оптимизатору обрабатывать запросы, содержащие новые функции и типы данных, не менее эффективно, чем запросы, использующие только встроенные средства SQL.

В системе PostgreSQL предусмотрены также средства для подключения модулей расширения функциональности, добавления новых процедурных языков программирования, для полной замены отдельных компонент (таких, как оптимизатор запросов) и другие возможности, позволяющие при необходимости использовать эту СУБД как платформу для построения специализированных решений.

17.6. Упражнения

Упражнение 17.1. Постройте описание типа для хранения значений температуры. Температура может быть выражена в кельвинах и градусах Цельсия, Фаренгейта и Реомюра. Значения температуры не должны быть ниже 0 по шкале Кельвина. Сравните реализации в виде домена и в виде нового типа данных.

Упражнение 17.2. Опишите типы данных, необходимые для представления бронирования в виде одного значения составного типа. Информация о бронировании содержит основные атрибуты самого бронирования, список пассажиров и список беспосадочных перелетов. Для включения в значение списков пассажиров и перелетов используйте массивы.

Глава 18

Полнотекстовый поиск

18.1. Модели информационного поиска

Необходимость поиска в больших коллекциях текстов возникла задолго до появления компьютеров, по-видимому, вместе с появлением публичных библиотек. Первые компьютерные системы информационного поиска появились примерно в то же время, что и базы данных. Эти два направления развивались параллельно, и только в последние десятилетия началось их взаимопроникновение.

В самом общем случае задача информационного поиска состоит в том, чтобы в большой коллекции документов найти те, содержание которых может удовлетворить информационную потребность пользователя. В этой главе мы будем рассматривать коллекции текстовых документов, однако задача поиска, конечно, рассматривается и для других типов документов (например, изображений).

Поскольку ни понятие «содержание документа», ни «информационная потребность» не определены точно, приведенная постановка задачи вряд ли может быть полезна для построения автоматизированной системы. Тем не менее это определение дает способ проверки качества работы информационно-поисковой системы: такая оценка может быть произведена пользователем, у которого возникла информационная потребность.

В автоматической системе информационная потребность выражается *поисковым запросом*, а документы представляются *поисковыми образами*, которые строятся на основе текста документа. Результатом работы системы становится набор документов, которые система сопоставила с поисковым запросом. Документ, содержание которого удовлетворяет информационную потребность, называется *релевантным*.

Поисковые образы не могут точно отражать содержание, а запросы — информационную потребность. Дополнительная неточность может вноситься методом сопоставления запросов и поисковых образов. Поэтому результат работы информационно-поисковой системы всегда неточен. В этом состоит основное

отличие поисковых запросов от обычных: последние выдают набор объектов, удовлетворяющих точно определенным критериям, а поисковые запросы выражают информационную потребность неточно.

Одна из простых идей состоит в том, чтобы использовать для поиска текстов встроенный предикат LIKE, определенный в стандарте SQL, или регулярные выражения. Эта идея, однако, очень плохо работает для текстов, поскольку таким образом очень трудно выразить особенности естественных языков.

Язык запросов и структура базы данных определяются моделью данных, а способы формирования поисковых запросов и поисковых образов документов — *моделью информационного поиска*. Важная особенность моделей информационного поиска состоит в том, что они, в отличие от моделей баз данных, не предполагают замкнутость мира. В английской терминологии это, соответственно, closed world assumption и open world assumption. Проиллюстрируем это различие примерами. В демонстрационной базе данных запрос

найти коды всех аэропортов, для которых город *не* Москва,

возможно, не очень полезен, но вполне осмыслен, и система может выдать вполне определенный ответ, потому что (неявно) предполагается, что никаких аэропортов, кроме перечисленных в таблице базы данных, не существует. В этом и выражается предположение о замкнутости мира.

В моделях информационного поиска, однако, предполагается, что множество всех возможных документов, вообще говоря, неизвестно. Например, можно рассматривать запрос

найти все документы, посвященные системам управления
базами данных, *но не* упоминающие MySQL,

однако запрос

найти все документы, *не* упоминающие MySQL

считается недопустимым. В реальности, конечно, любая система информационного поиска может найти только те документы, поисковые образы которых в ней зарегистрированы, поэтому фактически система работает как замкнутая.

Существует несколько моделей информационного поиска. Наиболее важной из них для этой главы является *булева модель*. В рамках любой модели тексты на естественном языке могут подвергаться предварительной обработке.

18.1.1. Предварительная обработка текста

Обработка текста на естественном языке начинается с разбиения на *слова* (tokens). Далее каждое слово приводится к некоторой канонической форме, зависящей от части речи (например, именительный падеж единственного числа для имен существительных). Оба этих процесса могут существенно зависеть от языка, на котором написан анализируемый документ.

Слова в поисковом запросе также приводятся к каноническим формам. Слова естественного языка, которые приводятся к одной канонической форме, считаются эквивалентными.

Например, слова из предыдущего предложения могут быть приведены к канонической форме следующим образом:

```
слово естественный язык который приводить к один канонический
форма считать эквивалентный.
```

Этот пример показывает, что приведение к канонической форме приводит к потере части информации, содержавшейся в тексте до обработки.

После приведения к канонической форме все или некоторые слова становятся *поисковыми термами*. Способ выбора термов зависит от модели информационного поиска и от конкретной системы. Далее выбранные термы используются для формирования поискового образа (документа или запроса).

Кроме этого, эквивалентными могут быть и различные канонические формы слов, если они выражают одно или близкие понятия. Особая сложность задачи приведения к канонической форме связана с тем, что эквивалентность может зависеть от предметной области, к которой относится документ. В ранних системах информационного поиска словари синонимов и эквивалентов составлялись вручную. В современных системах для этого широко применяются методы искусственного интеллекта.

18.1.2. Булева модель информационного поиска

В рамках булевой модели каждый документ представляется набором термов, в качестве которых обычно используются некоторые слова естественного языка, характеризующие этот документ. Различные способы получения термов и составления поискового образа обсуждаются ниже. Поисковый запрос в этой

модели представляется как логическое выражение, содержащее термы, связанные логическими операторами конъюнкции (AND), дизъюнкции (OR), исключения (BUT NOT) и скобками. Вместо оператора конъюнкции в таких языках часто используется запятая.

Документ считается соответствующим запросу, который представляет собой просто список термов, разделенных запятыми, если он содержит все термы, содержащиеся в запросе (или эквивалентные термы, если такая эквивалентность определена в системе). Дизъюнкции таких списков соответствует набор документов, поисковые образы которых содержат либо слова из одного, либо из другого списка, а операция исключения запретит включение документов, соответствующих ее второму аргументу.

В теоретико-множественной интерпретации логики конъюнкции (AND) соответствует пересечение множеств, дизъюнкции (OR) — объединение, а отрицанию (NOT) — дополнение. Операция BUT NOT интерпретируется как теоретико-множественная разность. Выбор операции исключения BUT NOT объясняется предположением об открытости мира.

Поисковые запросы в булевой модели могут содержать и более сложные условия, например указание предельного расстояния между термами в тексте документа. В некоторых системах можно указывать, что термы должны встречаться в одном предложении и т. п.

В ранних системах поиска текстовых документов в 70-е гг., когда ресурсы были крайне ограничены, для построения поискового образа использовались тексты заглавий, аннотаций или списки ключевых слов. В настоящее время для построения поисковых образов используются полные тексты документов, т. к. ничто другое не может представить содержание документа точнее, а вычислительные ресурсы такую обработку уже не ограничивают.

Структуры данных, обычно применяемые в поисковых системах для реализации булевой модели поиска, схематически представлены в упрощенном виде на рис. 18.1.1. Эти структуры являются конкретизацией инвертированных индексных структур, которые рассмотрены в разделе 11.2.3.

В результате предварительной обработки документ превращается в последовательность поисковых термов и представляется в этой структуре как список ссылок на словарь (т. е. каждый терм в документе заменяется на ссылку на этот терм в словаре, а все, что не рассматривается как поисковый терм, исключается из документа).

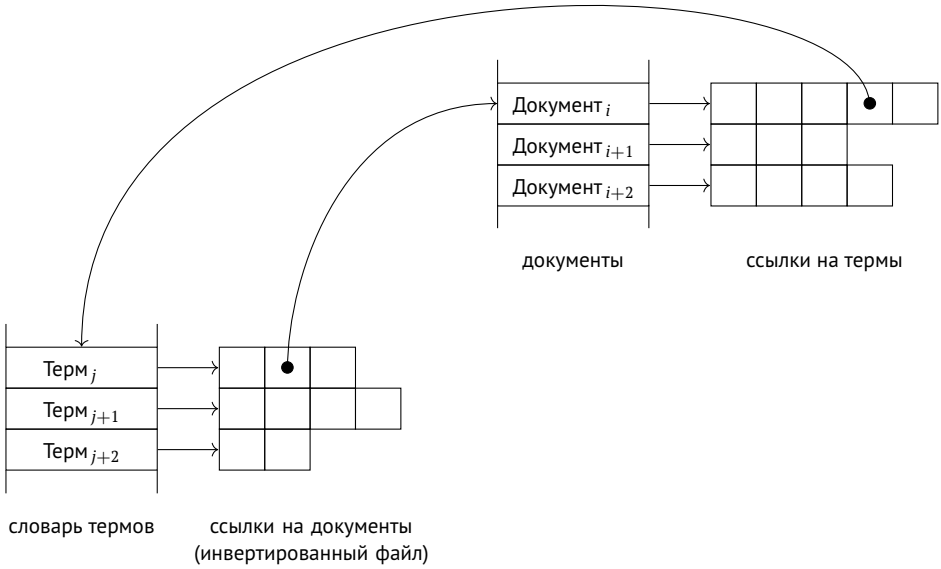


Рис. 18.1.1. Структуры данных для булевой модели поиска

Каждая запись словаря может содержать либо поисковый терм (например, слово в канонической форме), либо слово в исходной форме. В некоторых системах применяется двухуровневый словарь: на первом уровне хранятся слова в исходных формах вместе со ссылками на каноническую форму, которая хранится на втором уровне. Двухуровневая структура позволяет выполнять поиск как по исходным словам, так и по приведенным к канонической форме. Обычно для хранения словаря используется структура инвертированного файла.

Инвертированный файл содержит для каждого терма из словаря список ссылок на документы, содержащие этот терм. Более сложная конструкция предусматривает хранение количества вхождений для каждого терма и, возможно, номера позиций в документе, на которых находится этот терм. При этом ссылки на документы в каждом списке упорядочиваются.

Это позволяет находить список документов, соответствующих поисковому запросу, с помощью следующего алгоритма:

1) выбрать из словаря все термы, упомянутые в запросе;

2) получить список документов, соответствующих запросу, однократным просмотром пост-листов (алгоритм слияния).

Известны многочисленные уточнения и улучшения этого алгоритма.

Развитые поисковые системы, как правило, используют несколько более сложные структуры. Так, обычно в таких системах ведется список часто встречающихся слов. Включение таких слов в словарь не помогает решению задачи поиска, поскольку они встречаются почти во всех документах. С другой стороны, по этой же причине списки документов, содержащих такие слова, получаются очень длинными, и, следовательно, занимают много места в памяти и замедляют поиск. Тем не менее полное исключение часто встречающихся слов может быть нежелательно: известны примеры документов, заглавие которых не содержит никаких слов, кроме часто встречающихся. Например, запрос на поиск документов, озаглавленных «Что делать?», может оказаться безрезультатным, но, возможно, некоторые читатели найдут в своей памяти непустой ответ на подобный запрос.

Как правило, для улучшения качества поиска в системах хранятся списки синонимов. Такие списки применяются для расширения поисковых запросов.

Булева модель поиска допускает относительно простую и вычислительно эффективную реализацию (схема которой описана выше), однако обладает существенными недостатками. Основной недостаток этой модели состоит в том, что она не позволяет оценить, в какой мере содержание найденного документа (неточно выраженное поисковыми термами) удовлетворяет информационную потребность, неточно выраженную запросом. Другими словами, эта модель не дает возможности количественно оценить степень релевантности документа: документ либо включается в результат, либо не включается.

18.1.3. Векторные модели информационного поиска

Другой подход к решению задачи информационного поиска предполагает, что между документами и запросами определено некоторое количественно оцениваемое отношение схожести или близости. Формально это означает, что на множестве всех документов определено понятие *расстояния*. Документ тем более релевантен запросу, чем меньше расстояние между ними.

Поскольку документ можно представлять текстовой строкой, можно было бы рассматривать известные определения расстояния для текстовых строк, например редакционное расстояние (расстояние Левенштейна, вычисляемое как количество операций вставки, удаления и замены символов, необходимых для

превращения одной строки в другую). Такая метрика, однако, никак не учитывает особенности текстов на естественном языке, и поэтому ее применение обычно дает очень плохие результаты.

Более удачной оказывается идея строить поисковые образы документов как векторы в многомерном пространстве.

Простейший вариант *векторной модели информационного поиска* основан на гипотезе о том, что чем чаще некоторое слово встречается в документе, тем важнее это слово для представления содержания документа. Однако слова, которые часто встречаются во многих документах, имеют меньшее значение для представления содержания отдельных документов.

Поисковый образ документа в этой модели строится как вектор, размерность которого равна количеству различных слов (или термов) в коллекции. В качестве значения компоненты вектора используется вес слова в документе. В соответствии с гипотезой, сформулированной выше, вес должен быть тем больше, чем чаще встречается слово в документе, и тем меньше, чем чаще это слово встречается в коллекции. Отношение количества вхождений терма в документ к количеству документов, содержащих терм, обладает этими свойствами. Этот простейший метод вычисления веса принято обозначать TF-IDF (term frequency — inverse document frequency). Известно довольно много различных формул для вычисления веса термов, а также другие методы построения векторных поисковых образов документов. В современных системах применяются более сложные методы построения векторов, в том числе основанные на искусственных нейронных сетях.

Для того чтобы компенсировать различие документов по длине, векторы нормируют, т. е. умножают все компоненты на константу так, чтобы длина вектора стала равной 1. Близость (схожесть) документов оценивается в векторной модели расстоянием между векторами. Мы не будем обсуждать в этой книге выбор метода вычисления расстояния.

Точно такие же векторы строятся по запросам, которые в этом случае рассматриваются просто как наборы слов или тексты на естественном языке. Другими словами, в этой модели нет разницы между запросами и документами. В качестве запроса можно просто использовать документ, т. е. запрос имеет вид: *найти документы, похожие по содержанию на заданный*. Задача поиска при этом превращается в задачу поиска ближайших соседей в метрическом (или псевдометрическом) пространстве.

Математическое определение метрического пространства не используется далее в этой книге и приводится здесь только для любознательных читателей. *Метрикой* на некотором множестве M называется функция с вещественными числовыми значениями, определенная на множестве пар элементов из M, такая, что:

1. $d(a, a) = 0$;
2. $d(a, b) = d(b, a)$;
3. $d(a, b) > 0$;
4. $d(a, b) + d(b, c) \geqslant d(a, c)$.

Обычное евклидово расстояние в векторном пространстве является метрикой. Если последнее из условий (неравенство треугольника) не выполняется, то функция называется *псевдометрикой*.

Обратим внимание на то, что в векторной модели изменяется понятие релевантности: в отличие от булевой модели расстояние между запросом и документом позволяет оценить степень релевантности, при этом любой документ оказывается в какой-то мере релевантным запросу.

Часто вместо функции расстояния применяют другую функцию, которая называется *подобием* и принимает значения в интервале $(0, 1]$: равна 1 для совпадающих документов и убывает при увеличении расстояния между документами. Значение функции подобия можно интерпретировать как вероятность того, что документ релевантен запросу.

Несмотря на то что векторная модель никак не учитывает структуру документа, оказывается, что она может давать лучшие результаты, чем булева модель поиска.

Применение векторной модели требует значительно бóльших вычислительных ресурсов по сравнению с булевой моделью и осложняется тем, что поиск ближайших соседей в многомерном пространстве оказывается сложной задачей. Заметим, однако, что для вычисления весов может использоваться информация, содержащаяся в инвертированном списке, если список документов содержит счетчики вхождений терма в документ.

Модели векторного типа используются как для решения задач, связанных с обработкой текстов на естественном языке, так и для поиска других типов объектов (например, изображений).

18.2. Средства полнотекстового поиска в PostgreSQL

Средства поддержки полнотекстового поиска, входящие в состав системы PostgreSQL, предоставляют возможность полноценного поиска документов в рамках булевой модели, дополненную оценками близости на основе частот термов (как в векторной модели).

Коллекция документов, подлежащих индексированию для последующего полнотекстового поиска, может содержаться в качестве значений некоторого атрибута отношения текстового типа, храниться вне базы данных во внешних файлах или вычисляться как выражение, составленное из нескольких атрибутов. Однако в любом случае документ, подлежащий индексированию, должен быть представлен в виде текстовой строки и в таком виде передан для предварительной обработки.

Предварительная обработка документа включает выделение слов, выполняемое синтаксическим *анализатором* (parser), и преобразование слов или групп слов в термы (приведение к канонической форме). В процессе приведения к канонической форме могут использоваться многочисленные *словари*:

- словарь часто встречающихся слов, игнорируемых при индексировании;
- словарь синонимов;
- словарь устойчивых словосочетаний, которые отображаются в один терм;
- словарь, управляющий приведением слов к канонической форме;
- правила преобразования при предварительной обработки текста.

Предварительная обработка завершается построением внутреннего представления документа в виде упорядоченного списка термов. Предварительно обработанные документы записываются в виде значений типа данных tsvector, а внутреннее представление поискового запроса является значением типа данных tsquery.

В системе PostgreSQL предусмотрено большое количество функций для преобразования текстов в формы tsvector и tsquery, а для последнего также функции, превращающие значение типа tsquery в запрос булевой модели, возможно, содержащий операторы & (AND), | (OR), ! (NOT) и <-> для задания относительной позиции термов в документе.

Сопоставление текста с запросом выполняется бинарным оператором @@.

Результаты работы многих функций, связанных с полнотекстовым поиском, зависят от того, какие используются словари и какие программные модули применяются для анализа и приведения к канонической форме. Информация об этом записывается в виде *конфигурации*. В составе PostgreSQL имеются конфигурации для многих естественных языков, кроме этого, имеется возможность определять дополнительные конфигурации. Какая именно конфигурация будет использоваться, можно задать на нескольких уровнях от сервера баз данных до отдельного вызова функции, в которой используется конфигурация.

В составе системы PostgreSQL имеются средства, упрощающие создание и частичное изменение конфигураций, в том числе программные модули и шаблоны для формирования словарей.

Для того чтобы исключить полный просмотр коллекции документов при каждом поиске, необходимо использовать индексы. Для поиска в относительно больших документах наиболее подходящим типом индекса является GIN. Нет необходимости хранить документы в формате tsvector, поскольку предварительную обработку можно совместить с построением индекса, указывая в операторе CREATE INDEX выражение, вычисляющее tsvector. В этом случае, однако, необходимо явно указывать конфигурацию и индекс становится функциональным: в запросах придется также включать вычисление tsvector в точно такой же форме, как указано при построении индекса. Кроме этого, индекс содержит не всю информацию, которую можно получить из формата tsvector, поэтому при выполнении запросов будет происходить дополнительная обработка документов для построения этого представления. В частности, это необходимо для выполнения запросов, в которых указаны ограничения на порядок слов (термов) в документе.

Поэтому более практичный способ состоит в определении дополнительной колонки таблицы типа tsvector и хранении там обработанного документа. Для поддержания значения этой колонки актуальным целесообразно создать триггеры. Получаемая в результате структура данных близка к изображенной на рис. 18.1.1.

Оценка релевантности документа запросу выполняется с помощью функций ранжирования, которые используют информацию о количестве вхождений термов, встречающихся в запросе, в документ, а также относительные позиции термов в документе. Кроме этого, имеется возможность задать относительные веса термов или частей документа при его предварительной обработке. Эти веса, если они были назначены, также используются при вычислении ранга.

Важно заметить, что вычисление ранга учитывает только запрос и документ и выполняется после того, как релевантные документы уже выбраны. Ранг вычисляется только для документов, найденных с помощью оператора @@ и не может повлиять на то, какие документы будут найдены. При этом значения ранга могут отражать степень релевантности документов в пределах одного запроса, но эти значения несопоставимы для разных запросов. По этой причине ранг можно использовать только для упорядочивания результатов выполнения одного запроса.

Инвертированные индексы (GIN) сами по себе никак не связаны с обработкой текстов на естественном языке. Они могут быть полезны в тех случаях, когда значения атрибута представляют собой множества или списки, количество элементов которых велико или варьируется в широких пределах, для организации эффективного поиска по значениям элементов этих списков. Например, можно использовать такие индексы для поддержки запросов на включение множеств (представленных массивами).

18.3. Поддержка нечеткого поиска в PostgreSQL

В реальных задачах параметрами для поиска часто являются данные, введенные пользователями. Такие данные могут содержать орфографические ошибки, неточности (пользователи часто не знают точных терминов) и опечатки. В этом случае имеет смысл говорить о нечетком поиске, т. е. поиске по образцу с учетом допустимых различий. Алгоритмы нечеткого поиска используют метрики, позволяющие оценить степень схожести слов. Степень схожести может учитывать набор символов, из которых состоят слова, фонетику (поскольку пользователи могут не знать написание слова, но помнить его звучание), а также расстояние между клавишами на клавиатуре для определения вероятности опечаток.

Для поддержки нечеткого поиска система PostgreSQL предоставляет несколько расширений.

18.3.1. Триграммный поиск

Данный вид поиска помогает учитывать опечатки и ошибки автоматического распознавания. Алгоритм основан на том, что схожие слова должны обладать

общими подстроками, и чем больше общих подстрок эти слова имеют, тем более они схожи. В PostgreSQL используются триграммы, т. е. подстроки длины 3.

Чтобы воспользоваться триграммным поиском нужно активировать расширение pg_trgm, входящее в стандартную поставку PostgreSQL.

```
demo=# CREATE EXTENSION pg_trgm;
CREATE EXTENSION
```

Чтобы получить представление о триграммах, можно воспользоваться функцией, которая возвращает массив триграмм для данного слова:

```
demo=# SELECT show_trgm('airport');
                show_trgm
----------------------------------------
 {"  a","  ai",air,irp,ort,por,rpo,"rt "}
(1 row)
```

Функция similarity возвращает число от 0 до 1, которое показывает, насколько схожи параметры функции:

```
demo=# SELECT similarity('airport', 'ariport'),
              similarity('airport', 'seaport');
 similarity | similarity
------------+------------
 0.33333334 | 0.23076923
(1 row)
```

Для того чтобы выбрать значения, схожие с заданным параметром, используется оператор %:

```
demo=# SELECT city FROM airports WHERE city % 'Новый Новгород';
       city
-----------------
 Нижний Новгород
 Новый Уренгой
(2 rows)
```

Этот запрос возвращает все города, названия которых схожи со значением параметра. Слова, у которых схожесть не превышает некоторый порог (по умолчанию равный 0,3), отбрасываются. Установить новое значение порога можно при помощи конфигурационного параметра pg_trgm.similarity_threshold.

Для увеличения производительности запросов необходимо использовать индексы GiST или GIN.

Триграммный поиск может быть использован для любого языка.

18.3.2. Фонетический поиск

Если поиск проводится по именам собственным, у которых точное правописание неизвестно, имеет смысл применять фонетические алгоритмы. Для этого можно использовать расширение fuzzystrmatch.

```
demo=# CREATE EXTENSION fuzzystrmatch;
CREATE EXTENSION
```

Расширение предоставляет несколько алгоритмов.

Soundex каждому слову ставит в соответствие код, первый символ которого — первая буква слова, а остальные три символа выбираются в соответствии с таблицей кодировки, в которой буквам соответствуют цифры. Буквы, отсутствующие в таблице (например, гласные), отбрасываются. Если код получается больше четырех символов, то он обрезается; если меньше, то дополняется нулями.

Для получения кода используется функция soundex. Функция difference возвращает значение от 0 до 4, показывающее, сколько символов в коде совпадает:

```
demo=# SELECT soundex('Anne') anne,
    soundex('Ann') ann,
    difference('Anne', 'Ann');
 anne | ann  | difference
------+------+------------
 A500 | A500 |          4
(1 row)
```

Функция работает только для английского языка, а попытка применить ее для имен на любом другом языке не даст никаких полезных результатов.

В настоящее время существуют более точные алгоритмы, чем soundex.

Metaphone также сопоставляет строкам некоторый код. При этом используются более сложные правила преобразования, и результат получается точнее. Функция metaphone имеет дополнительный аргумент — максимальную длину получаемого кода:

```
demo=# SELECT metaphone('William Smith',10) william,
    metaphone('Wiliem Smyth',10) wiliam;
 william | wiliam
---------+--------
 WLMSM0  | WLMSM0
(1 row)
```

Так же как и Soundex, Metaphone работает только с английским языком.

Double Metaphone — более полезный алгоритм для неанглийских имен, т. к. он пытается учесть различные варианты произношения. Для каждого слова генерируются два кода, соответствующие основному и альтернативному вариантам произношения:

```
demo=# SELECT dmetaphone('william') william,
    dmetaphone_alt('william') william_alt,
    dmetaphone('wiliem') wiliem,
    dmetaphone_alt('wiliem') wiliem_alt;
 william | william_alt | wiliem | wiliem_alt
---------+-------------+--------+------------
 ALM     | FLM         | ALM    | FLM
(1 row)
```

18.4. Итоги главы

В системах полнотекстового информационного поиска используются специфические средства и алгоритмы, в которых практически нет необходимости при работе со структурированными типами данных, более характерными для традиционных применений СУБД.

Встроенные в систему PostgreSQL средства полнотекстового поиска дают возможность организовать высококачественный поиск по текстам документов, интегрированный в базу данных и легко сочетаемый с другими средствами СУБД. По сравнению с проектными решениями, основанными на копировании и синхронизации данных в специализированную систему поиска, реализация поиска средствами СУБД дает возможность всегда выполнять поиск по актуальному согласованному состоянию данных, обеспечивается разграничение доступа, имеется возможность сочетания поисковых запросов с обычными реляционными.

Не следует, однако, заменять поиск по структурированной части базы данных полнотекстовым поиском, поскольку это — надежный способ получения неповоротливых прикладных систем с ущербной функциональностью.

Булева модель информационного поиска описана в [55] и интенсивно исследовалась в более ранних работах, например в [34], а также была реализована в промышленных системах информационного поиска в 70-е гг. Активное развитие векторных моделей началось в 80-е гг. благодаря исследованиям, которые показали перспективность этой модели [53; 54], а также вследствие роста

доступных вычислительных мощностей, сделавшему эту модель практически применимой. В литературе описаны и применялись другие модели, например вероятностные [38].

18.5. Упражнения

Упражнение 18.1. Создайте в базе данных таблицу, содержащую 10 000 текстов на естественном языке (например, статей из википедии). Все документы должны быть на одном языке.

1. Сформируйте и запишите в файл 20 поисковых запросов и выполните их, выбирая тексты из созданной таблицы. Для каждого запроса запишите время выполнения.
2. Постройте функциональный GIN-индекс и повторите запросы; сравните время выполнения.
3. Добавьте колонку для значений типа tsvector, уничтожьте старый индекс и создайте новый GIN-индекс по этой колонке. Повторите выполнение запросов и сравните время выполнения.

Упражнение 18.2. Выполните то же самое, что и в предыдущем упражнении, но таблица должна содержать документы на разных языках.

Упражнение 18.3. Напишите выражение, которое построит текстовое представление для бронирования, выбирая информацию о времени отправления и прибытия, типе самолета, аэропортах и населенных пунктах отправления и прибытия. Постройте GIN-индекс для этого документа и обычные индексы для поиска по времени отправления и по аэропортам отправления и прибытия. Выполните поиск рейсов, вылетающих из Шереметьево в определенный день, с помощью текстового поиска и обычного SQL-запроса. Сравните время выполнения.

Глава 19

Безопасность данных

19.1. Безопасность и разграничение доступа

Вопросы безопасности информационных систем исключительно важны, однако подробное обсуждение всех аспектов этой темы далеко выходит за рамки этой книги. Как правило, надежная защита должна быть многоуровневой. Мы же рассмотрим вопросы безопасности только применительно к серверу баз данных.

Одно из основных правил любой системы защиты состоит в том, что клиенты базы данных (пользователи или приложения) должны иметь доступ только к тем функциям и ресурсам, которые нужны для решения их задач.

Прежде всего, подчеркнем, что недопустимо использование роли с правами суперпользователя для нормальной работы информационной системы. Такие роли могут использоваться только для выполнения операций по обслуживанию базы данных.

Далее следует определить, какие из приложений будут выполняться от имени одного пользователя, а какие — от имени разных пользователей. В системах с очень большим количеством потенциальных пользователей, например в системах с открытым доступом из интернета, все соединения могут устанавливаться от имени одной роли базы данных. Скажем, все покупатели в интернет-магазине могут обслуживаться одной ролью, но при этом, возможно, каждый менеджер будет использовать индивидуальную роль.

В этом случае, очевидно, ограничить доступ покупателя исключительно к его заказам можно только на уровне приложения, а ограничить доступ менеджеров к отдельным функциям, отношениям или строкам отношений можно на уровне базы данных. В то же время понадобится (скорее всего) определить специальную роль «менеджер», которая объединит привилегии, необходимые всем менеджерам. Право использования этой роли предоставляется всем ролям, предназначенным для персонального использования отдельными менеджерами. В более сложных системах количество подобных групповых ролей

может быть значительно больше. Отдельные роли могут понадобиться для аналитиков, а также для персонала, занимающегося обслуживанием базы данных.

Для разграничения доступа между ролями базы данных можно использовать:

- размещение данных и функций в различных схемах, при этом разные роли могут иметь различные права на доступ к различным схемам;
- различные привилегии на хранимые таблицы;
- представления, ограничивающие доступ к хранимым таблицам;
- разграничение доступа на уровне строк таблиц;
- пользовательские функции, хранимые в базе данных.

Применение пользовательских функций дает наиболее широкие возможности для контроля доступа. В некоторых организациях доступ приложений к таблицам запрещают полностью, предоставляя вместо этого хранимые процедуры, реализующие все операции работы с данными, необходимые приложению.

Хранимые функции в особенности полезны для выполнения операций, требующих дополнительных проверок. Для того чтобы предотвратить возможность обхода такой процедуры, ее можно создать в режиме SECURITY DEFINER. Это означает, что код процедуры будет выполняться с правами роли, от имени которой эта процедура была определена, а не той, от имени которой она выполняется.

19.2. Основные понятия и модели

Напомним, что защита данных от несанкционированного использования с самого начала рассматривалась как одна из основных функций систем управления базами данных. В главе 5 рассмотрены модели, на которых основаны средства защиты в системе PostgreSQL.

Основными функциями СУБД, связанными с безопасностью данных, являются:

- аутентификация — проверка пользователя или приложения, запускаемого от имени пользователя, на наличие права работы с базой данных;
- разграничение доступа — возможность пользователей обращаться только к тем объектам данных, на доступ к которым они имеют право (при этом разные пользователи могут иметь различные права).

Основными сущностями, используемыми в системе PostgreSQL для реализации функций безопасности, являются *роли* и *привилегии*. Другими словами, основной моделью безопасности, принятой в PostgreSQL, является, как и во многих других СУБД, модель управления доступом *на основе ролей* (role based access control, RBAC).

Альтернативная модель, позволяющая разграничивать доступ к данным *на основе значений атрибутов объектов* (attribute based access control, ABAC), частично поддерживается с помощью средств разграничения доступа на уровне строк таблиц, определяемых политиками доступа.

19.3. Особенности ролей в PostgreSQL

Понятие роли в системе PostgreSQL объединяет понятия *пользователя* и *группы пользователей* с одинаковыми правами доступа к объектам данных. Роли определяются на уровне кластера баз данных, идентифицируются своим именем и характеризуются набором *атрибутов* и *привилегий*.

Атрибуты определяются ключевыми словами, определенными в языке SQL для системы PostgreSQL. Значения атрибутов задаются в операторе CREATE ROLE и могут быть изменены оператором ALTER ROLE.

Многие пары атрибутов являются взаимоисключающими, и каждая роль имеет один из двух атрибутов пары независимо от того, был ли указан один из них при определении роли. Например, такую пару образуют атрибуты CREATEROLE и NOCREATEROLE, определяющие, может ли роль создавать новые роли.

Среди атрибутов имеются, в частности, следующие:

- SUPERUSER — наличие прав суперпользователя;
- CREATDB — право создания баз данных;
- CREATEROLE — право создания ролей;
- LOGIN — право создавать сеансы работы с базой данных.

Роли, имеющие право создавать сеансы, по существу выполняют функции пользователей кластера баз данных. Для создания, изменения и удаления пользователей можно использовать операторы CREATE USER, ALTER USER и DROP USER соответственно, однако созданные таким образом пользователи неотличимы от ролей, имеющих атрибут LOGIN.

Другие атрибуты могут задавать ограничения на использование роли: ограничивать количество одновременных сеансов (CONNECTION LIMIT), определять пароль (PASSWORD) или ограничивать его срок действия (VALID UNTIL).

Роли могут быть связаны отношением наследования.

19.4. Привилегии

Создание пользователя (роли с атрибутом LOGIN) само по себе не дает возможности выполнять какие-либо действия над объектами базы данных. Для того чтобы такие действия стали возможными от имени некоторой роли, необходимо предоставить этой роли соответствующие *привилегии*.

В качестве привилегий для роли можно назначать:

- привилегии на выполнение операций над объектами данных;
- другие роли, которым были назначены некоторые привилегии.

В отличие от атрибутов набор привилегий, которые могут быть назначены роли, не ограничен в том смысле, что любая роль может иметь привилегии для работы с любым количеством объектов базы данных. Добавление привилегий выполняется оператором GRANT, отзыв привилегий — оператором REVOKE.

Для того чтобы создавать объекты данных, роль должна иметь привилегию CREATE на создание объектов в соответствующей схеме (на логическом уровне) и в соответствующем табличном пространстве (на уровне хранения). Исключение составляют временные объекты, для создания которых требуется привилегия TEMPORARY для базы данных, и сами базы данных, право создавать которые внутри кластера задается не привилегией, а атрибутом роли.

Роль, от имени которой создавался объект данных, становится его *владельцем* (однако имеется возможность изменить владельца впоследствии). Владелец объекта имеет все привилегии, которые определены для этого типа объекта. Передачу привилегий другим ролям можно выполнять от имени:

- владельца объекта данных;
- роли с правами суперпользователя;
- роли, которая имеет соответствующую привилегию с правом передачи привилегии другим ролям.

Перечень привилегий, которые могут быть предоставлены для работы с некоторым объектом базы данных, зависит от типа этого объекта. Для каждого типа объекта список возможных привилегий зафиксирован в языке SQL и, как правило, определяется списком операторов SQL, которые можно выполнять для этого объекта. Например, список привилегий для отношений включает SELECT, INSERT, UPDATE, DELETE, TRUNCATE. Если объект базы данных является отношением, то в некоторых случаях можно указать список колонок, на которые распространяются предоставляемые привилегии.

19.5. Права доступа при выполнении хранимых функций

Привилегии, связанные с объектами данных, определяют возможность выполнения тех или иных операций на уровне операторов SQL. Так, привилегия на изменение данных разрешает выполнение оператора UPDATE, при этом становятся возможными любые обновления. Точно так же право чтения данных никак не ограничивает вид запросов SELECT, которые могут быть выполнены от имени роли, имеющей такую привилегию.

Правила, устанавливаемые для выполнения функций, несколько сложнее.

Прежде всего для того чтобы выполнить функцию от имени роли, необходимо, чтобы эта роль имела право выполнять эту функцию, т. е. привилегию EXECUTE. Кроме этого, необходимо, чтобы во время выполнения тела функции имелись права на доступ к тем объектам базы данных, которые используются в этой функции. Обычно функции создаются с (возможно, неявным) указанием SECURITY INVOKER. Это означает, что во время выполнения тела функции применяются права доступа, определенные для роли, *вызвавшей* выполнение функции. Такая модель разграничения доступа для функций дает возможность выполнять одну и ту же функцию от имени разных ролей, и при этом будут применяться разные комбинации прав доступа.

Например, если внутри функции генерируется текст запроса, который потом выполняется, то такой запрос может работать только с объектами базы данных, доступными вызывающей роли (и, следовательно, одна и та же функция будет работать с разными объектами для разных ролей или пользователей).

В системе PostgreSQL существует другой вариант определения правил разграничения доступа для функций, который задается явным указанием SECURITY

DEFINER. В этом случае тело функции выполняется с правами роли, *создавшей* эту функцию. При этом пользователь, вызывающий функцию, не обязательно должен иметь права доступа к объектам базы данных, которыми манипулирует функция.

Такие функции позволяют обеспечить более тонкое разграничение доступа, чем обеспечивается обычными привилегиями доступа к объектам базы данных. Для реализации такой схемы необходимо создать как минимум две роли:

1. Роль владельца объектов базы данных, которые подлежат защите (таблиц, представлений, последовательностей и т. п.), а также всех функций, которые будут использоваться для обработки этих объектов базы данных. При этом функции создаются с указанием SECURITY DEFINER.

2. Одна или несколько ролей для пользователей (приложений). Каждой из этих ролей предоставляется право выполнения (EXECUTE) всех или некоторых из функций, определенных владельцем, но не предоставляется право доступа к объектам базы данных непосредственно.

В этой схеме возможности доступа пользователей к объектам базы данных ограничиваются только теми операциями, что могут быть выполнены с помощью функций, право выполнения которых передано этому пользователю. Конечно, внутри функций могут выполняться и дополнительные проверки.

Такая модель разграничения прав доступа для функций аналогична разграничению доступа в объектных системах, в которых пользователю предоставляется только право выполнения некоторых из методов, определенных для класса объектов.

Проиллюстрируем применение SECURITY DEFINER небольшим примером. От имени пользователя-владельца базы данных определим функцию, возвращающую количество строк в таблице (в схеме bookings), затем создадим нового пользователя и передадим ему права доступа к схеме и право выполнения этой функции:

```
demo=# CREATE OR REPLACE FUNCTION aircraft_cnt() RETURNS bigint
LANGUAGE sql SECURITY DEFINER AS $$
    SELECT count(*) FROM aircrafts;
$$;
CREATE FUNCTION

demo=# CREATE USER aircraft_counter;
CREATE ROLE
```

```
demo=# GRANT EXECUTE ON FUNCTION aircraft_cnt() TO aircraft_counter;
GRANT
demo=# GRANT USAGE ON SCHEMA bookings TO aircraft_counter;
GRANT
```

Попытка выполнения этой функции от имени нового пользователя и попытка прямого выполнения содержащегося в ней запроса дают разные результаты:

```
demo=# \connect demo aircraft_counter
You are now connected to database "demo" as user "aircraft_counter".
demo=> SELECT aircraft_cnt();
 aircraft_cnt
--------------
            9
(1 row)
demo=> SELECT count(*) FROM aircrafts;
ERROR:  permission denied for view aircrafts
```

Таким образом, новый пользователь может выполнять функции, право выполнения которых ему передано, но не может выполнять никакие другие действия.

19.6. Разграничение доступа на уровне строк таблиц

Приведенные выше средства дают возможность предотвратить несанкционированный доступ к данным, а также ограничить доступ зарегистрированных пользователей к отдельным объектам схемы или даже к отдельным колонкам таблиц.

Эти средства, однако, не позволяют разграничить доступ *различных* пользователей к строкам *одной* таблицы, что бывает необходимо для многих приложений. В системе PostgreSQL такое разграничение можно задать с помощью *средств безопасности на уровне строк* (row level security). Проиллюстрируем использование этих средств на простом примере. Прежде всего нам необходимо зарегистрировать пользователей, для которых будет применяться разграничение доступа на уровне строк:

```
demo=# CREATE USER alice;
CREATE ROLE
demo=# CREATE USER bob;
CREATE ROLE
```

Поскольку этот пример не имеет прямого отношения к демонстрационной базе данных, создадим в этой базе данных новую схему:

```
demo=# CREATE SCHEMA rowlevel;
CREATE SCHEMA
```

Таблица notes в этой схеме будет использоваться для хранения заметок, сделанных пользователями, при этом каждый пользователь будет иметь возможность видеть, добавлять и изменять только те заметки, которыми он владеет. Мы не будем в этом примере строить более сложные схемы разграничения доступа; например, наши пользователи не смогут предоставлять доступ к своим заметкам другим. Определение таблицы notes может быть таким:

```
demo=# CREATE TABLE rowlevel.notes (
    note_owner text NOT NULL,
    note_key text NOT NULL,
    note_text text NOT NULL,
    PRIMARY KEY (note_key)
);
CREATE TABLE
```

Далее необходимо определить правила разграничения доступа. В PostgreSQL такие правила называются *политиками* (policy) и определяются отдельно для каждой таблицы. Условия для чтения существующих данных (предложение USING) могут отличаться от условий, ограничивающих внесение новых значений в таблицу (задаваемых предложением WITH CHECK). При этом условие USING проверяется при выборке данных, обновлении и удалении (операторы SELECT, UPDATE, DELETE), а проверка новых значений выполняется при вставке и обновлении (операторы INSERT и UPDATE). Кроме этого, при определении политики указывается, для каких операторов будет применяться эта политика. В нашем примере мы определим одну политику, которая будет применяться для всех операций на таблице notes:

```
demo=# CREATE POLICY per_user ON rowlevel.notes
FOR ALL
USING (
    note_owner IN (SELECT current_user)
)
WITH CHECK (
    note_owner IN (SELECT current_user)
);
CREATE POLICY
```

Условия в этом примере записаны не самым экономным способом: тот же самый результат можно получить с помощью условия note_owner = current_user.

Мы использовали вложенный подзапрос, для того чтобы показать, что условия разграничения доступа могут быть довольно сложными, в частности использовать данные из других таблиц.

Для того чтобы политики, определенные на таблице, использовались, необходимо активировать безопасность на уровне строк для этой таблицы:

```
demo=# ALTER TABLE rowlevel.notes
ENABLE ROW LEVEL SECURITY;
ALTER TABLE
```

Далее необходимо разрешить пользователям создавать сеансы для работы с базой данных и разрешить доступ к схеме rowlevel и к таблице notes, находящейся в этой схеме:

```
demo=# GRANT CONNECT ON DATABASE demo TO alice, bob;
GRANT
demo=# GRANT USAGE ON SCHEMA rowlevel TO alice, bob;
GRANT
demo=# GRANT ALL ON TABLE rowlevel.notes TO alice, bob;
GRANT
```

Чтобы убедиться, что владелец таблицы notes имеет доступ ко всем строкам этой таблицы независимо от работы средств разграничения доступа, добавим в эту таблицу несколько строк:

```
demo=# INSERT INTO rowlevel.notes VALUES
    ('alice', 'alice-1', 'first note of Alice'),
    ('alice', 'alice-2', 'second note of Alice'),
    ('bob', 'bob-1', 'first note of Bob'),
    ('bob', 'bob-2', 'one more note of Bob');
INSERT 0 4
```

Выборка данных от имени владельца таблицы показывает, что все добавленные строки доступны:

```
demo=# SELECT *
FROM rowlevel.notes;
 note_owner | note_key |      note_text
------------+----------+----------------------
 alice      | alice-1  | first note of Alice
 alice      | alice-2  | second note of Alice
 bob        | bob-1    | first note of Bob
 bob        | bob-2    | one more note of Bob
(4 rows)
```

Теперь можно показать, как работает разграничение доступа, если сеанс работы с базой данных создается от имени наших пользователей:

```
demo=# \connect demo alice
You are now connected to database "demo" as user "alice".
demo=> SELECT *
FROM rowlevel.notes;
 note_owner | note_key |      note_text
------------+----------+---------------------
 alice      | alice-1  | first note of Alice
 alice      | alice-2  | second note of Alice
(2 rows)
demo=> \connect demo bob
You are now connected to database "demo" as user "bob".
demo=> SELECT *
FROM rowlevel.notes;
 note_owner | note_key |      note_text
------------+----------+---------------------
 bob        | bob-1    | first note of Bob
 bob        | bob-2    | one more note of Bob
(2 rows)
```

Один и тот же оператор выборки возвращает разные результаты в зависимости от того, какой пользователь выполняет этот запрос:

```
demo=> INSERT INTO rowlevel.notes VALUES
    ('alice', 'bob-10', 'Bob attempts to insert on behalf of Alice');
ERROR:  new row violates row-level security policy for table "notes"
demo=> INSERT INTO rowlevel.notes VALUES
    ('bob', 'bob-1', 'Bob attempts to insert duplicate key');
ERROR:  duplicate key value violates unique constraint "notes_pkey"
demo=> INSERT INTO rowlevel.notes VALUES
    ('bob', 'bob-20', 'Bob  inserts correctly');
INSERT 0 1
```

Теперь пользователь bob видит следующее состояние таблицы notes:

```
demo=> SELECT *
FROM rowlevel.notes;
 note_owner | note_key |       note_text
------------+----------+-----------------------
 bob        | bob-1    | first note of Bob
 bob        | bob-2    | one more note of Bob
 bob        | bob-20   | Bob  inserts correctly
(3 rows)
```

Подчеркнем, что средства разграничения доступа на уровне строк основаны на понятии пользователя базы данных. Как отмечено выше, в современных приложениях, как правило, это понятие не используется. Вместо этого пользователи определяются на уровне приложения. Это обстоятельство существенно затрудняет применение средств разграничения доступа в таких приложениях. Можно, конечно, определить политики доступа таким образом, чтобы они зависели от других условий, например от значений параметров, явно или неявно передаваемых из приложения. В этом случае, однако, разграничение доступа будет основано на доверии к значениям, переданным из приложения, что в значительной мере снижает ценность этого инструмента как средства безопасности.

Разграничение доступа на уровне строк требует проверки политик при выполнении каждого оператора SQL для каждой обрабатываемой строки. Очевидно, это создает дополнительную нагрузку на сервер баз данных.

19.7. Регистрация событий и изменений

Регистрацию активности пользователей принято считать одним из важных элементов системы безопасности данных.

В системе PostgreSQL имеются средства для регистрации сообщений в *журнале сообщений* (это не то же самое, что журнал транзакций WAL, который ведется для защиты от отказов). Управление регистрацией осуществляется установкой конфигурационных параметров сервера. В журнале сообщений можно регистрировать создание и завершение сеансов работы с базами данных (параметры log_connections и log_disconnections соответственно), а также выполняемые операторы SQL, включая как операторы модификации данных, так и запросы на чтение, возможно, вместе со временем выполнения (log_statement и log_min_duration_statement).

Журнал сообщений можно использовать не только как средство контроля безопасности, но и для других целей, например для анализа использования отдельных запросов, что может быть полезно для настройки системы.

Более детальную информацию о работе СУБД можно получать с помощью расширения pgAudit. В частности, информация, собираемая этим расширением, может потребоваться для аудита работы компании.

Регистрация изменений в отдельных таблицах может быть реализована на основе триггеров. Если такая регистрация используется в целях безопасности, необходимы дополнительные меры, для того чтобы информация, записываемая триггерами, не была доступна для приложений (и желательно, чтобы факт регистрации не был известен пользователю, действия которого регистрируются). Слабость этого метода состоит в том, что триггеры выполняются в рамках транзакций, поэтому записи о регистрации попыток обновления в оборванных транзакциях будут уничтожены при откате этих транзакций.

Необходимо заметить, что любой вид регистрации событий приводит к существенному увеличению нагрузки на сервер и при неосторожном применении создает очень большое количество данных. Поэтому обычно включают регистрацию только отдельных видов событий или только для отдельных объектов базы данных.

19.8. Итоги главы

В этой главе рассмотрены основные понятия модели разграничения доступа на основе ролей (RBAC) и показано, каким образом эта модель реализуется в системе привилегий PostgreSQL.

Рассмотрены механизмы, предназначенные для разграничения доступа на уровне строк таблиц (ABAC) для зарегистрированных пользователей сервера базы данных.

19.9. Упражнения

Упражнение 19.1. Создайте роль для доступа на чтение к демонстрационной базе данных без права создания сеансов работы с сервером БД.

Упражнение 19.2. Создайте пользователя сервера БД и предоставьте ему привилегию использования роли, созданной в предыдущем упражнении. Проверьте, что этот пользователь может выполнять любые запросы на выборку из таблиц демонстрационной базы данных, но не может их обновлять.

Упражнение 19.3. Постройте пример, показывающий, что для доступа к таблицам схемы необходимо также предоставить право использования (USAGE) этой схемы.

Упражнение 19.4. Реализуйте схему разграничения доступа на уровне строк для пользователей уровня приложения (не зарегистрированных как пользователи на сервере базы данных).

Упражнение 19.5. Реализуйте разграничение доступа на уровне строк, обеспечивающее три уровня:

1) совершенно секретный доступ;
2) секретный доступ;
3) открытый доступ.

При этом пользователь, имеющий доступ к более высокому уровню, должен также иметь доступ и к более низким. Нескольким пользователям сопоставьте различные уровни доступа из упомянутого списка (с помощью вспомогательных таблиц базы данных). Убедитесь, что разграничение действительно работает.

Глава 20

Администрирование баз данных

Администрирование баз данных включает несколько типов технических работ, которые должны проводиться с разной степенью регулярности. Эти работы выполняются человеком, которого принято называть администратором базы данных, хотя вообще-то это далеко не всегда один человек. В крупных организациях его функции может выполнять группа лиц, а в мелких — обязанности администратора иногда совмещают с деятельностью разработчика или системного администратора. Поэтому правильнее говорить, что администратор базы данных — это роль, которая предполагает выполнение технических работ, проводимых с базой данных.

Администратор базы данных отвечает за реализацию политик, которые приняты в организации для обеспечения сохранности и безопасности данных, а также за эффективность доступа к данным, т. е. за производительность сервера баз данных. Для того чтобы выполнить все эти функции, требуется определенное количество разнообразных ресурсов: рабочего времени администратора БД, вычислительных систем, пропускной способности вычислительных сетей, устройств хранения данных. Потребности в ресурсах зависят от выбранных политик, которые, в свою очередь, зависят от требований к информационной системе: чем выше требования (например, по доступности), тем больше ресурсов необходимо выделить для выполнения этих требований.

Поскольку потребность в ресурсах существенно зависит от выбранных политик, выбор должен осуществляться администратором базы данных совместно с администрацией предприятия, которая отвечает за расходование ресурсов. Наиболее сложным аспектом планирования политик, от которых зависит надежность, оказывается оценка возможных рисков. Высокая стоимость защиты от разрушений в сочетании с относительно высокой надежностью современного оборудования приводит к тому, что риски недооцениваются значительно чаще, чем переоцениваются или оцениваются адекватно.

Ошибки в определении политик могут стоить очень дорого. В известных событиях 11 сентября 2001 г. пострадали не только люди, но и базы данных, которые

были установлены на серверах в Северной и Южной башнях Всемирного торгового центра. Отсутствие копий за пределами этих зданий привело к полной потере информации, содержавшейся в этих базах данных. Между тем этих потерь можно было бы избежать при выполнении элементарных правил обеспечения безопасности данных.

В этой главе мы рассмотрим основные задачи, которые сопутствуют роли администратора баз данных. Круг задач, разумеется, может меняться в зависимости от специфики приложений, работающих с базой, однако в любом случае существует определенный список задач, стоящих перед администратором любой базы данных:

- планирование объемов и конфигурации сервера баз данных;
- конфигурация и поддержка средств безопасности и защиты данных;
- создание и конфигурация баз данных;
- мониторинг БД;
- обеспечение производительности БД (настройка);
- обеспечение надежности хранения данных;
- техническое обслуживание БД.

Почти все эти задачи требуют внимания на всех фазах жизненного цикла информационной системы, начиная с определения требований к системе и заканчивая миграцией на новую систему. Очень многое определяется на начальных фазах спецификации и разработки системы, когда эксплуатация еще не началась и поэтому функции администратора могут быть еще не выделены, однако современные методологии разработки предполагают быстрый переход к производственной эксплуатации с последующим итеративным развитием информационной системы.

В таких условиях администратор базы данных должен работать в тесном контакте с разработчиками приложения, однако функции администратора существенно отличаются от функций разработчиков. В частности, администратор базы данных отвечает за четкое разделение баз данных, используемых для разработки, тестирования и производственной эксплуатации системы. Конечно, такое разделение важно независимо от применяемой методологии разработки, однако технологии быстрого прототипирования и быстрого внедрения могут провоцировать отклонения от четкого соблюдения правил разделения баз данных.

20.1. Планирование конфигурации сервисов хранения данных

Важность задачи планирования связана с тем, что при ее решении определяются конфигурации оборудования и программных систем, которые необходимы для выполнения всех функций информационной системы, и, следовательно, стоимость этой системы. Определение конфигурации включает:

- тип и состав оборудования и операционной среды, в которой будет работать система управления базами данных;
- конфигурацию СУБД и, если необходимо, дополнительных пакетов программ, обеспечивающих ее работу;
- объемы памяти, необходимой для хранения данных (диски и т. п.).

Выбор архитектуры оборудования и типа операционной системы, безусловно, важен, однако PostgreSQL может работать практически на любой операционной системе. Поэтому при выборе платформы следует учитывать другие факторы, а конфигурацию СУБД определять уже на основе выбранной платформы. Конфигурация оборудования и операционной системы, однако, зависит от требования к серверу базы данных по доступности, надежности и безопасности.

В последние годы многие предприятия, в особенности небольшие, размещают свои базы данных на «облачных» сервисах, предоставляемых крупными центрами хранения и обработки данных. Преимуществами такого решения оказываются возможность постепенного наращивания объемов данных и необходимых для их обработки вычислительных мощностей, а также относительно высокая надежность и доступность этих сервисов. В качестве недостатков можно назвать ограниченные возможности выбора конфигурации СУБД и зависимость от надежности внешних сервисов.

Программные средства, дополняющие СУБД, обычно не требуют значительных вычислительных ресурсов и упоминаются здесь только для полноты. К числу таких средств можно отнести различные программы для анализа состояния базы данных, мониторинга и т. п.

Наиболее важной частью планирования, непосредственно касающейся администратора баз данных, является оценка необходимого объема дисковой памяти для хранения данных. Кроме объема базы данных, важно также определить необходимые размеры оперативной памяти и мощность вычислительной системы (количество ядер процессоров), которые зависят от ожидаемой нагрузки

на систему. Первоначальная оценка должна быть получена при разработке информационной системы, однако фактические потребности могут отличаться от запланированных, поэтому администратор должен обновлять такие оценки по мере роста объемов в процессе эксплуатации системы и при изменении требований к ней.

В любом случае оценки учитывают текущие потребности и прогноз динамики роста. Конечно, эти оценки должны быть оценками сверху, т. к. незначительное завышение оценки не приведет к существенному удорожанию системы, а заниженные оценки могут вызвать как технические (ухудшение производительности), так и организационные (необходимость срочной закупки дополнительных ресурсов) последствия.

Оценка текущей потребности складывается из нескольких компонентов.

Оценка размеров основной базы данных, включающая:

- Оценки размеров хранимых данных.

  Такая оценка может быть получена на основе спецификаций информационной системы. Например, для грубой оценки размеров данных в таблице можно умножить среднюю длину строки таблицы на количество строк.

- Оценки размеров дополнительных структур хранения.

  Оценка размеров материализованных представлений не отличается от оценки размеров таблиц. Размеры индексов, конечно, зависят от типа индекса и от характеристик данных, однако бывалые администраторы пользуются упрощенными оценками, считая, например, что *размер индекса составляет 15 % от размера таблицы, для которой он построен*.

- Резервы памяти, необходимые для обновлений.

  В системе PostgreSQL все обновления данных всегда записываются на новое место, поэтому количество необходимой дополнительной памяти зависит от интенсивности потока обновляющих транзакций. Этот размер целесообразно оценивать для часто изменяемых таблиц отдельно.

- Потребности в дисковой памяти для временных объектов.

  Оценить объем, занимаемый временными объектами (промежуточными результатами выполнения запросов, временными таблицами

и т. п.) довольно сложно. Оценки должны учитывать характер и интенсивность нагрузки на сервер базы данных (частоту и сложность запросов).

- Дисковая память для ведения журналов транзакций.

  Необходимый объем зависит от интенсивности нагрузки на сервер и от значений конфигурационных параметров, определяющих частоту контрольных точек.

- Резервы памяти на рост объемов хранимых данных.

  Эта оценка, очевидно, связана с прогнозом роста.

Перечисленные оценки в сумме составляют размер основной (рабочей) копии базы данных. Полученное значение используется для оценки потребности в дисковой памяти для целей, перечисленных в двух следующих пунктах.

Ресурсы для хранения резервных копий базы данных.

В эту категорию включается память для хранения архивированных журнальных файлов (которые необходимы для восстановления после отказов носителя) и резервных копий, в том числе серверов в теплом или горячем резерве. Очевидно, что этот размер зависит в первую очередь от требований по выживаемости и доступности, а также от выбранной стратегии копирования и восстановления. В любом случае этот размер может в несколько раз превышать размер основной базы данных.

Базы данных для разработки и тестирования.

Общепринятая практика программной инженерии предполагает ведение баз данных для разработки и тестирования новых версий системы. Требования по надежности хранения для этих баз данных значительно ниже, чем для основной, и, как правило, резервное копирование не требуется. Однако по объему эти базы данных не должны существенно отличаться от основной, потому что использование для разработки базы данных малого размера почти неизбежно приводит к созданию кода, работающего с базой данных крайне неэффективно.

Разумеется, некоторое количество памяти будет занято программными средствами самой СУБД и другими пакетами, однако, по-видимому, эти размеры учесть проще всего.

Получить реалистическую оценку динамики роста значительно сложнее, чем оценку текущих потребностей. Такая оценка зависит в первую очередь от оценки перспектив роста предприятия. Наиболее важный вопрос, ответ на который необходимо получить при перспективном планировании, — в какой мере может потребоваться масштабируемость базы данных.

В подавляющем большинстве случаев, в особенности в небольших компаниях, оценки перспектив оказываются необоснованно завышенными и никак не соотносятся с реальностью, например с потенциальным размером рынка. Другое широко распространенное заблуждение состоит в том, что масштабируемость может компенсировать низкое качество программного кода и вызванную этим недостаточную производительность системы.

20.2. Безопасность и разграничение доступа

Модели, методы и средства защиты данных от несанкционированного доступа и разграничения доступа подробно рассмотрены в главах 5 и 19. По существу, задача администратора базы данных состоит в определении и реализации политик, удовлетворяющих требованиям, описанным в этих главах.

Самое сложное в работе администратора данных по обеспечению безопасности — это соблюдение дисциплины и политик, установленных им самим. Так, не следует без необходимости использовать роли с правами суперпользователя. Для плановой работы с базами данных следует использовать роли с меньшими правами. Не следует забывать, что любые оставленные «для себя» лазейки неизбежно становятся уязвимыми местами во всей системе защиты.

20.3. Конфигурация баз данных

Сложность управления конфигурацией всего кластера баз данных и каждой отдельной базы зависит, прежде всего, от объема данных. Важны, конечно, тип оборудования и тип операционной системы, но они влияют только на способ установки программного обеспечения и взаимодействие с файловой системой.

Для небольших по размеру баз данных и кластеров баз данных (размер которых не превышает единиц гигабайт) и при не слишком интенсивной нагрузке на

сервер баз данных, возможно, никаких изменений в конфигурации не потребуется: значения параметров и размещение данных, заданные при установке сервера, могут обеспечить удовлетворительную работу сервера. Однако следует учитывать, что в этом случае возможности оборудования, скорее всего, будут использованы далеко не полностью, потому что начальная конфигурация подобрана так, чтобы СУБД могла запускаться практически на любом оборудовании (в том числе с малым объемом оперативной памяти и невысокой мощностью других ресурсов).

При высокой интенсивности запросов может понадобиться изменение параметров, влияющих на распределение оперативной памяти, выделенной для работы сервера баз данных. Современные серверы часто оснащаются большим объемом оперативной памяти; на них целесообразно увеличивать размеры буферного кеша и областей для временного хранения промежуточных данных, создаваемых во время выполнения запросов.

Для баз данных большего размера может понадобиться изменение размещения табличных пространств на устройствах постоянного хранения. Напомним, что табличные пространства являются структурой уровня хранения, поэтому при их размещении можно руководствоваться соображениями производительности, но не логической структурой баз данных. В частности, бессмысленно отождествлять табличные пространства со схемами базы данных.

В некоторых СУБД табличные пространства размещаются в одном файле (или в небольшом количестве файлов) операционной системы, а выделение места для отдельных объектов базы данных реализуется СУБД. Для таких систем выбор табличного пространства для объекта оказывается важным. Например, может быть целесообразна группировка в одном табличном пространстве структур хранения, обладающих похожей динамикой изменения размеров.

В системе PostgreSQL память для объектов базы данных выделяется средствами файловой системы, поэтому применение нескольких табличных пространств имеет смысл в случае, если они размещаются на разных устройствах (в разных файловых системах).

Если в составе оборудования имеются носители с существенно разными характеристиками (например, HDD и SSD), то это, конечно, следует учитывать при размещении хранимых объектов. Различие характеристик накопителей влияет и на соотношение стоимости операций ввода-вывода и поэтому должно также учитываться при выборе значений параметров сервера, управляющих оптимизатором. Заметим, что в PostgreSQL значения соответствующих параметров могут назначаться отдельно для каждого табличного пространства.

Для таблиц большого размера часто применяется *секционирование* (partitioning). Первоначально идея секционирования была введена для размещения данных на нескольких серверах параллельной системы баз данных, однако считается, что разбиение данных на секции может быть полезно даже в том случае, если секции размещаются на одном устройстве хранения. Заметим, однако, что реализация секционирования в PostgreSQL может накладывать определенные ограничения, которые постепенно снимаются в последующих версиях.

Для очень больших объемов данных может понадобиться хранение в распределенной системе. Возможности и особенности распределенных систем баз данных обсуждаются в главе 22.

В последние годы наиболее часто применяется распределение данных между серверами на уровне приложения (или на промежуточном уровне между приложением и СУБД), которое принято называть *шардированием* (sharding). Такое распределение дает хорошие результаты, если данные можно разбить на независимые фрагменты, размещаемые на разных серверах распределенной системы. Однако в том случае, когда данные из разных фрагментов необходимо достаточно часто обрабатывать совместно, средства распределенных СУБД могут обеспечить более высокую производительность.

В любом случае управление распределенными системами оказывается достаточно сложным, поэтому вряд ли целесообразно применять распределенную систему для компенсации дефектов кода приложения. Базы данных такого объема, для которого действительно требуется распределенное хранение, встречаются достаточно редко даже в больших организациях.

20.4. Мониторинг баз данных

Администратор БД должен следить за тем, чтобы база данных была доступна как для отдельных пользователей, так и для приложений, работающих с ней. Именно поэтому администратор должен проводить регулярный мониторинг пользователей базы, активности клиентских сеансов и используемых ресурсов. Регулярный мониторинг позволяет уловить моменты, когда в системе внезапно увеличивается потребление процессорного времени или операций ввода-вывода, или значительно возрастает количество соединений, и, как следствие, администратор сможет вовремя определить угрозу исчерпания выделенных ресурсов.

Мониторинг активности клиентских сеансов позволяет определить пиковые часы (или дни) нагрузки на БД и разобраться, что происходит с базой в это время. Такого рода пики носят, как правило, достаточно периодический характер. Периодичность сильно зависит от типа приложений. Это может быть расчет зарплаты в определенные числа каждого месяца, ежедневное открытие торгов на бирже, когда в системе совершается огромное количество сделок, и т. п.

Данные о пиковых нагрузках дают ценнейшую информацию, которая позволяет увидеть наиболее узкие места приложений в целом: какие запросы выполняются в это время, какое количество пользователей их выполняет, достаточно ли ресурсов и т. п. Причем интерес представляет не только максимально возможная, но и минимальная нагрузка. На основе этой информации можно составлять расписание выполнения разного рода регулярных задач, например процедур *извлечения, преобразования и загрузки данных* (extract-transform-load, ETL) в хранилище данных для последующей аналитической обработки. В некоторых системах необходимо устанавливать расписание для обновления статистики, необходимой для работы оптимизатора запросов. В системе PostgreSQL предпочтительнее применять автоматические средства сбора статистики, запуск которых выполняется в зависимости от количества обновлений.

Кроме того, мониторинг нагрузки БД позволит спрогнозировать потребности в дополнительных ресурсах. Знание этой информации даст возможность администратору спланировать свои последующие действия и не допустить сбоев в работе системы в часы пиковых нагрузок.

Для мониторинга активности клиентских сеансов в PostgreSQL предусмотрены многочисленные системные представления и специальные статистические функции, которые по своему назначению могут быть разбиты на две группы. Первая группа ориентирована на получение оперативной информации о том, что только что произошло или происходит в базе. Вторая — на получение общих сведений о работе, размерах базы и статистической информации с момента ее последнего сбора.

Системное представление pg_stat_activity относится к первой группе и может быть использовано для получения системного идентификатора серверного процесса, сведений о подключении, времени начала выполнения активного в данный момент запроса и текста этого запроса, текущего события ожидания процесса и другой информации.

Существенная особенность при работе с базой данных — понимание не только текущего размера, но и прогнозирование роста объема данных на ближайшее

время. Динамика работы с базой отражается в нескольких системных представлениях.

Например, представление pg_stat_database отражает динамику изменения баз с момента предыдущего сбора статистики, количество зафиксированных и оборванных транзакций, сведения об интенсивности доступа к данных (как в строках, как и в страницах) и много другой полезной информации. Следующий запрос позволит увидеть имена баз данных и количество добавленных, измененных и удаленных строк в этих базах:

```
SELECT datname, tup_inserted, tup_updated, tup_deleted
FROM pg_stat_database;
```

Статистику по различным объектам баз данных можно увидеть в таких представлениях, как pg_stat_user_tables, pg_stat_user_indexes, pg_stat_user_functions.

Организация хранения баз данных в файлах операционных систем отличается в различных СУБД, но в любом случае в конечном итоге стационарные объекты размещаются в файлах на диске или дисках.

Кроме стационарных объектов БД, во время выполнения клиентских сеансов могут создаваться временные таблицы, время жизни которых ограничивается транзакцией или клиентским сеансом. Эти объекты, хоть и носят временный характер, могут создавать значительную нагрузку на дисковое пространство. Дело в том, что, в отличие от стационарных таблиц, данные в которых разделяются многими пользователями, временные таблицы создаются и заполняются данными отдельно в каждом клиентском сеансе и, следовательно, необходимый размер дисковой памяти для их хранения напрямую зависит от количества сеансов и количества записей в таких таблицах. Желательно рассчитать объем базы так, чтобы все данные гарантированно помещались в дисковой памяти, предусмотренной для их хранения (рекомендуется использовать под хранение базы не более 50 % объема всего дискового пространства).

Мониторинг занимаемого дискового пространства можно осуществлять разными способами. Следующий запрос демонстрирует использование системного представления pg_class для вывода информации о размерах (в страницах) десяти самых больших таблиц демонстрационной базы данных:

```
demo=# SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC
LIMIT 10;
```

```
                relname                  | relpages
-----------------------------------------+----------
 ticket_flights                          |     8715
 tickets                                 |     6144
 ticket_flights_pkey                     |     5185
 boarding_passes                         |     4263
 boarding_passes_pkey                    |     2875
 bookings                                |     1674
 boarding_passes_flight_id_boarding_no_key |     1592
 boarding_passes_flight_id_seat_no_key   |     1592
 tickets_pkey                            |     1415
 bookings_pkey                           |      723
(10 rows)
```

Такая информация собирается как часть статистики и используется планировщиком запросов. Действительный размер отношений можно смотреть функциями pg_relation_size или pg_total_relation_size (последний вариант включает для таблиц размер всех индексов и данных, вынесенных в таблицы TOAST). Похожие функции есть и для баз данных (pg_database_size), и для табличных пространств (pg_tablespace_size).

Необходим также мониторинг оставшегося свободного дискового пространства средствами операционной системы.

Кроме отслеживания размеров хранимых объектов и заполненности дискового пространства, необходимым элементом мониторинга является контроль нагрузки на сервер, создаваемой запросами. Динамику интегральных характеристик работы сервера баз данных, таких как нагрузка на процессор, систему ввода-вывода, на сети передачи данных и т. п., можно отслеживать средствами операционной системы.

Более детальную информацию о нагрузке, создаваемой отдельными запросами, можно получить с помощью расширения pg_stat_statements, позволяющего получать суммарное, максимальное и минимальное время выполнения для каждого запроса, количество операций чтений и записи, количество повторных выполнений каждого запроса.

20.5. Настройка производительности

Напомним, что настройкой называется комплекс мер по приведению прикладной системы в соответствие с требованиями производительности без изменения ее функциональности. Иначе говоря, выполняемые в процессе настройки

изменения не должны приводить к изменению каких-либо функций системы или особенностей их работы.

Например, чтобы улучшить время ответа, HTML-страницы для отображения на клиентской системе могут заранее подготавливаться на основе информации из базы данных. В этом случае обновления, вносимые в базу данных, будут отображаться с некоторой задержкой. Это приведет к тому, что пользователь, рассчитывающий на оперативный ответ (например, проверяющий баланс своего телефона), будет получать не то, что ожидает, поэтому использование такого типа кеширования вряд ли можно называть настройкой.

В идеальном мире требования по производительности формулируются одновременно с функциональными требованиями, что дает возможность учесть их во время проектирования и разработки системы. В реальности на начальных фазах жизненного цикла зачастую неявно предполагается, что никаких осложнений, связанных с производительностью системы, не будет, и требования по производительности не принимаются во внимание. Чем раньше начинается учет этих требований, тем более эффективны результаты, но обычно существенная часть работы по настройке выполняется администратором базы данных уже во время эксплуатации системы. Конечно, такая работа необходима в любом случае, потому что нагрузка на систему и требования к ней могут изменяться на фазе эксплуатации или в результате развития системы.

Оптимизаторы современных высокопроизводительных СУБД, к числу которых относится и система PostgreSQL, как правило, находят вычислительно эффективные планы выполнения для большинства запросов. Однако работа алгоритмов оптимизации основана на моделях, которые не могут давать точную оценку стоимости планов. Поэтому даже применение точных алгоритмов математической оптимизации, гарантирующих оптимальное значение функции стоимости, не может гарантировать оптимальность плана при фактическом выполнении. По этим причинам могут оказаться необходимыми меры по настройке системы управления базами данных, отдельных запросов или всего вместе.

Исходные данные для настройки получаются в результате мониторинга баз данных, рассмотренного выше. В частности, для настройки нужно знание размеров хранимых объектов. Необходима также информация о том, как данные используются: какие запросы выполняются и с какой частотой, какие данные с наибольшей вероятностью будут использоваться, как будет происходить ввод данных, с какой скоростью будут расти предполагаемые объемы данных и т. п.

Можно выделить следующие типы действий, которые могут выполняться при настройке прикладной системы:

- настройка серверов приложений и серверов баз данных (конфигурирование серверов и установка подходящих параметров);
- настройка схемы базы данных (например, создание дополнительных индексов для более эффективного выполнения некоторых запросов, материализация «тяжелых» представлений, секционирование больших таблиц и т. п.);
- локальная настройка отдельных функций или запросов (изменение кода отдельных запросов или функций приложения).

Разумеется, не все из этих действий может выполнить сам администратор. Так, например, локальная настройка функций или запросов выполняется совместно с разработчиком приложения. Однако обнаружить неэффективные запросы и инициировать действия по настройке приложения, которые позволят повысить производительность системы, должен администратор базы данных.

Во многих руководствах по администрированию баз данных указывается, что в процессе настройки сервера и схемы базы данных фактически происходит перераспределение ресурсов: эффективность выполнения одних действий (скажем, запросов) улучшается за счет ухудшения эффективности других действий. Это утверждение, безусловно, справедливо в том случае, если все запросы, выполняемые в системе, уже настроены так, что их выполнение близко к оптимальному, и, кроме этого, в системе достаточно высока загрузка всех видов ресурсов. Однако в реальности эти предположения выполняются крайне редко. Тщательная настройка запросов, потребляющих наиболее количество ресурсов, во многих случаях приводит к драматическому снижению нагрузки на сервер.

Кроме этого, во многих случаях небольшое ухудшение производительности для одних запросов может привести к очень значительному улучшению для других. Типичным пример — создание индексов, которое обычно замедляет относительно редко выполняемые обновления, но радикально ускоряет выполнение запросов на выборку данных.

Если хорошо настроенные запросы потребляют значительную долю ресурсов сервера, то ничто, кроме увеличения мощности оборудования, не может улучшить производительность.

Во многих случаях, однако, неудовлетворительная производительность приложения наблюдается одновременно с очень низкой (1–3 %) загруженностью сервера баз данных. В этом случае никакие меры по настройке только сервера баз данных не могут дать значительные результаты, потому что причина, скорее всего, состоит в том, что приложение выполняет слишком много слишком мелких запросов. Узким местом в этом случае оказывается количество синхронных обменов данными по вычислительной сети; при этом, как правило, пропускная способность сети также используется далеко не полностью, и поэтому ее увеличение также не может способствовать радикальному улучшению производительности. Единственным решением, которое может обеспечить улучшение работы системы в этом случае, оказывается переработка кода приложения таким образом, чтобы обмены данными с сервером базы данных происходили порциями большего размера, а СУБД выполняла более сложные запросы.

Приемы, применяемые при настройке разных СУБД, примерно одинаковы, хотя, конечно, конкретные особенности есть в каждой системе. Далее в этой главе речь идет в основном о настройке серверов, работающих под управлением СУБД PostgreSQL.

20.5.1. Настройка серверов баз данных

Настройка сервера выполняется с помощью параметров, которые могут задаваться в конфигурационных файлах системы PostgreSQL. Многие из них могут изменяться динамически и влиять как на работу всего сервера, так и на определенный сеанс, в котором эти изменения произведены.

По своим функциям параметры можно разделить на несколько групп.

Управление оперативной памятью. Эти параметры задают размеры различных областей памяти, которые могут использоваться для кеширования базы данных (shared_buffers), для хранения временных данных каждого сеанса (temp_buffers), для хранения промежуточных данных при выполнении отдельных запросов и операций (work_mem, maintenance_work_mem).

Управление дисковым хранилищем. Размещение постоянно хранимых данных определяется с помощью табличных пространств, а параметр сервера позволяет ограничить размер выделяемой области для временно хранимых служебных данных, например для промежуточных результатов операций, если эти результаты не помещаются в выделенной области оперативной памяти (temp_file_limit).

Управление фоновыми процессами. Фоновые процессы выполняют достаточно разнообразные действия, в том числе очистку и уплотнение данных (сборку мусора, vacuum), запись контрольных точек, сбор статистических характеристик хранимых данных. Частота запуска и другие особенности выполнения этих процедур влияют на загруженность системы и, таким образом, на ее производительность. Как правило, каждый фоновый процесс настраивается собственным набором конфигурационных параметров (количество которых может достигать нескольких десятков).

Управление оптимизатором. Параметры, относящиеся к этой группе, могут наиболее существенно повлиять на производительность и поэтому более детально обсуждаются ниже.

Выполнение любого запроса требует некоторого количества разнообразных ресурсов: времени процессора, различных видов операций обмена данными с устройствами хранения, некоторых ресурсов оперативной памяти и, возможно, других. Минимизировать требуемые количества всех этих ресурсов одновременно невозможно, поскольку планы, оптимальные по одному из ресурсов, скорее всего, не будут оптимальными по другим. В оптимизаторах запросов применяются функции стоимости, обеспечивающие некоторый баланс между различными ресурсами. Как правило, функция стоимости, используемая оптимизатором, представляет собой линейную комбинацию нескольких критериев, учитывающих стоимость различных ресурсов.

Коэффициенты этой линейной формы задают некоторое соответствие между стоимостями разнородных ресурсов, с тем чтобы сделать эти стоимости сопоставимыми. В системе PostgreSQL относительная стоимость ресурсов, необходимых для выполнения запроса, задается конфигурационными параметрами. Так, доступ к произвольным блокам (random_page_cost) на вращающихся дисках обычно в десятки раз медленнее, чем последовательный просмотр нескольких соседних блоков (seq_page_cost) на том же устройстве. Это соотношение, однако, может оказаться значительно меньше, если существенная часть базы данных находится в буферах в оперативной памяти; поэтому в своих оценках оптимизатор также учитывает оценку памяти, отведенной под кеш (как на уровне СУБД, так и в операционной системе, effective_cache_size).

При использовании твердотельных носителей (SSD) стоимости различных видов операций обмена отличаются не так существенно. В этом случае значение параметра random_page_cost следует уменьшить, чтобы оптимизатор мог правильно учитывать характеристики оборудования.

Итак, параметры этой группы влияют на вычисление функции стоимости, используемой при работе оптимизатора. Заметим, однако, что статья [37] показывает, что выбор функции стоимости оказывает меньшее влияние на качество планов, чем ошибки в оценках кардинальности промежуточных результатов.

Другая группа параметров влияет на конфигурацию пространства планов, которые будут рассматриваться оптимизатором при поиске оптимального плана. С помощью этих параметров можно исключить из рассмотрения планы, содержащие определенные виды физических операций.

Исключение из рассмотрения любого класса алгоритмов приводит к сокращению пространства поиска планов и поэтому ускоряет работу оптимизатора, однако может привести к потере оптимального плана.

Для операций фильтрации хранимых таблиц параметры дают возможность исключать полный просмотр таблиц (enable_seqscan), использование индексного просмотра (enable_indexscan), построение и сканирование битовых карт, полезное в том числе при наличии нескольких условий фильтрации, поддержанных индексами (enable_bitmapscan), а также чтение данных только из индексов (enable_indexonlyscan).

Для двуместных операций можно запретить оптимизатору использование каждого из классов алгоритмов: хеширования (enable_hashjoin), вложенных циклов (enable_nestloop) и сортировки на основе слияния (enable_mergejoin). Во многих случаях исключение алгоритма вложенных циклов для промежуточных операций соединения приводит к более эффективным планам, однако параметр системы PostgreSQL, к сожалению, одновременно отключает и алгоритмы вложенных циклов, использующие индексы, которые зачастую оказываются наиболее эффективными при выборке небольшого количества строк из хранимых таблиц.

Наконец, имеется группа параметров, влияющих на выбор алгоритма оптимизации. По существу, эти параметры определяют пороговые значения количества операций соединения (join_collapse_limit) или вложенных подзапросов (from_collapse_limit), по достижении которых оптимизатор прекращает исчерпывающий просмотр пространства планов. Еще один параметр (geqo_threshold) определяет пороговое значение количества соединений, включающее стохастический метод поиска оптимального плана (генетический алгоритм).

Комбинируя значения параметров, можно полностью отключить изменение порядка выполнения соединений. В этом случае порядок выполнения будет определяться порядком, в каком операции записаны в исходном запросе.

В системе PostgreSQL параметры, управляющие работой оптимизатора, можно задавать для всего сервера баз данных или для одного сеанса, сложнее — для одного запроса и совсем невозможно — для отдельных операций в запросе. Поэтому использовать параметры в стиле *подсказок* (hints) оптимизатору целесообразно для диагностики или временного изменения конфигурации для выявления причин низкой производительности, но, скорее всего, не в нормальной работе сервера.

20.5.2. Настройка схемы базы данных

Наиболее широко известным и наиболее часто применяемым методом настройки схемы является создание индексов. Несмотря на широкую известность даже среди непрофессионалов, этот метод никак нельзя назвать простым. Для грамотного выбора набора создаваемых индексов необходимо знание особенностей приложения и требований к нему, характеристик нагрузки на базу данных (набора частот появления различных запросов), а также особенностей реализации индексов и оптимизатора конкретной СУБД. Даже для одного типа индекса, например на основе B-дерева, число различных возможных индексов экспоненциально зависит от количества атрибутов индексируемой таблицы.

После создания индекса необходимо проверить, действительно ли он используется при выполнении тех запросов, для ускорения которых создавался, и действительно ли такое ускорение происходит. Необходимо также убедиться в том, что создание индекса не привело к ухудшению планов других запросов.

Напомним, что создание индексов может приводить не только к ускорению выполнения запросов, но и к замедлению. Причиной замедления работы СУБД может быть относительно большая доля операций обновления, при которых, очевидно, наряду с обновлением основной таблицы необходимо также обновить все индексы, включающие значения измененных атрибутов.

Большое разнообразие типов индексов, которые поддерживаются в системе PostgreSQL, делает задачу выбора индексов особенно сложной.

Завершая обсуждение индексов, напомним, что индексы прозрачны для приложения, т. е. создание или удаление индексов не влияет на корректность работы запросов и приложения и не требует модификации ни приложения, ни запросов. Кажущимся исключением являются индексы, реализующие ограничения целостности, однако такие индексы создаются и уничтожаются автоматически

вместе с ограничениями целостности. Но изменение ограничений целостности является изменением логической схемы базы данных, а не только схемы хранения, и такие изменения, конечно, не должны быть прозрачны.

Безусловно, уникальные индексы можно создавать и без объявления ограничений целостности с целью повышения производительности. Возможно, в этом случае указание уникальности необходимо, потому что уникальные индексы определенного типа работают эффективнее, чем неуникальные. Фактически попытка приложения записать неуникальные значения и в этом случае приведет к индикации ошибки, поэтому такие индексы следует считать неявными ограничениями целостности.

Другим методом настройки, который также следует отнести к категории методов настройки схемы, является создание материализованных представлений. В отличие от индексов материализованное представление необходимо явно указывать в запросе, чтобы оно было использовано. С одной стороны, это несколько усложняет применение материализованных представлений, с другой — гарантирует, что запросы, не подвергнутые изменениям, не станут выполняться хуже.

Цель применения материализованного представления состоит в том, чтобы многократно использовать результаты вычислений, выполненных при его создании или обновлении. Данные, хранимые в материализованном представлении, являются вторичными, т. е. вычисляются на основе других хранимых данных. Поскольку обновление материализованных представлений в настоящее время выполняется в PostgreSQL только по расписанию, обновления первичных данных могут не сразу учитываться в материализованном представлении, и приложение может поэтому получать устаревшие значения. В связи с этим важно подчеркнуть, что возможность применения материализованных представлений зависит от того, допустимо ли использование не совсем актуальных данных с точки зрения функций приложения. Как правило, это допустимо для отчетов, обобщающих данные за продолжительное время (скажем, месяцы), но, как правило, недопустимо для оперативного доступа к данным.

С учетом этих особенностей применение материализованных представлений особенно целесообразно, если выполняются следующие условия:

1) запрос, определяющий материализованное представление, выполняет вычисления, результаты которых часто используются в запросах;
2) семантика приложения допускает использование устаревших значений.

Заметим, что как индексы, так и материализованные представления являются избыточными структурами и приводят к увеличению объема памяти, занимаемой базой данных.

Наконец, настройка схемы включает методы и приемы, связанные с размещением данных. Этот довольно разнородный класс методов включает как управление размещением на физическом уровне (т. е. не затрагивает логику работы запросов и приложений), так и на логическом (требует модификации запросов). Перечислим основные виды приемов, относящихся к этому классу.

Табличные пространства. Размещение хранимых объектов (таких как таблицы, индексы, материализованные представления и т. п.) может быть полезно для достижения весьма разнообразного набора целей.

- При наличии в составе оборудования устройств с разной производительностью часто используемые данные целесообразно размещать на более эффективных устройствах (например, SSD).
- Хранимые объекты с различающейся динамикой обновления целесообразно хранить в различных табличных пространствах. В некоторых СУБД это может влиять на выделение дисковой памяти (операционной системой) и на фрагментацию свободной памяти внутри табличного пространства. Например, широко известна рекомендация размещать файлы таблиц и индексы в разных табличных пространствах.
- Табличные пространства можно размещать на разных физических устройствах таким образом, чтобы данные, которые часто нужны в одном запросе, оказались на разных устройствах. Целесообразность применения этого приема зависит от характеристик нагрузки на систему, типов используемого оборудования, а также от целей настройки (пропускная способность или время отклика).
- В некоторых системах (но не в PostgreSQL) для разных табличных пространств можно применять различающиеся стратегии создания резервных копий.
- В системе PostgreSQL размещение объектов базы данных в нескольких табличных пространствах позволяет сократить количество файлов в одном каталоге операционной системы. Это важно, потому что в PostgreSQL для каждого объекта базы данных (таблицы и т. п.) создается несколько файлов операционной системы.

- В некоторых системах (но не в PostgreSQL) состав поддерживаемой статистической информации для оптимизатора может определяться на уровне табличных пространств.

Размещение в табличных пространствах никак не влияет на логику работы приложений.

Вертикальная фрагментация. Таблицы, содержащие большое количество колонок, которые редко извлекаются все в одном запросе, могут быть заменены несколькими таблицами с меньшим числом колонок (но с одинаковыми первичными ключами). Такая фрагментация может быть сделана прозрачной для приложения с помощью представления, соединяющего все колонки, однако при этом может быть полностью потерян выигрыш в производительности.

Горизонтальная фрагментация. В этом случае таблица представляется объединением (UNION ALL) нескольких таблиц с одинаковой схемой, которые называются *секциями* (partitions) исходной таблицы. Горизонтальная фрагментация поддерживается на уровне SQL в системе PostgreSQL, но способ реализации накладывает некоторые ограничения на его использование.

Фрагментация, как вертикальная, так и горизонтальная, может быть особенно полезна в параллельных и распределенных системах, поскольку различные вертикальные фрагменты или секции могут обрабатываться параллельно.

Нормализация. Во многих ситуациях переход к правильно нормализованной схеме может привести не только к улучшению ее логической структуры, но и к повышению производительности.

Денормализация. Этот термин обычно обозначает замену нескольких таблиц результатом их соединения, что может быть полезно, если данные (почти) никогда не обновляются, например для *хранилищ данных* (data warehouse) и в других случаях, когда данные используются для аналитической обработки. В некоторых случаях денормализацию целесообразно реализовывать с помощью материализованных представлений.

Изменение порядка колонок. Размещение наиболее часто используемых колонок в начале может сократить процессорное время, необходимое для выполнения запросов, однако вертикальная фрагментация может дать более существенный выигрыш. В системе PostgreSQL и во многих других системах целесообразно размещать в начале колонки, содержащие данные фиксированной длины (а уже после них — часто используемые).

20.5.3. Настройка запросов

Настройка отдельных запросов оказывается необходимой в том случае, когда прикладная система в целом работает удовлетворительно, однако производительность отдельных запросов недопустимо низкая.

Наиболее частой причиной низкой производительности оказывается неэффективный код самого запроса. Вот далеко не полный перечень типичных ошибок.

- Использование представлений, в которых содержатся соединения с таблицами, ненужными в рассматриваемом запросе. Например, в следующем запросе выполняются избыточные соединения с таблицей airports: все вложенные подзапросы на самом деле выбирают одну и ту же строку, и поэтому одной операции соединения было бы достаточно:

```
SELECT
   (
    SELECT airport_name
    FROM airports
    WHERE airport_code = f.departure_airport
   ),
   (
    SELECT city
    FROM airports
    WHERE airport_code = f.departure_airport
   )
FROM flights f;
```

- Выборка значений первичного ключа из таблицы, соединение с которой выполняется по внешнему ключу. В этом случае операция соединения вообще не нужна, если из этой таблицы не выбираются другие колонки:

```
SELECT a.airport_code
FROM flights f
    JOIN airports a ON a.airport_code = f.departure_airport;
```

- Использование регулярных выражений для проверки условий, которые могут быть проверены по значениям других атрибутов.

- Поиск по значениям первичных данных, которые были обработаны при загрузке в базу данных. Например, одна из колонок таблицы может содержать адрес в том виде, в каком пользователь ввел данные (в виде одной строки), и еще несколько колонок могут содержать тот же адрес после анализа и приведения к некоторому каноническому виду (например, название улицы в верхнем регистре). Условие вида user_address LIKE '%190008%'

для поиска по почтовому индексу вместо условия на колонку, в которой содержится этот индекс, скорее всего, приведет к полному просмотру таблицы.

- Выражения, препятствующие применению индексов.
- Необоснованное использование оконных функций вместо явного агрегирования.
- Применение средств полнотекстового поиска для структурированных атрибутов.

Не имеет никакого смысла применять другие приемы настройки отдельных запросов до тех пор, пока подобные недостатки кода не устранены.

Дальнейшая настройка запросов в значительной мере зависит от того, к какому классу относится запрос. Многие приемы настройки, описанные в книге [68], до сих пор остаются применимыми.

Конечно, все используемые приемы не должны зависеть от конкретных хранимых данных. Предпочтительно применение методов, предполагающих использование оптимизатора, а не подменяющих его.

Начинать настройку следует с применения приемов, не зависящих от конкретной СУБД или тем более от ее версии.

Прежде всего необходимо проверить, что критерии фильтрации, исключающие значительную долю строк, обеспечены индексами и что эти индексы используются. В некоторых случаях может быть полезно добавление избыточных условий фильтрации по значениям атрибутов, которые обеспечены индексами.

Дальнейшие действия по настройке могут включать анализ операций соединения, наличие необоснованных полных просмотров таблиц и индексов, выбор алгоритмов выполнения наиболее сложных операций и др.

Если приемы, не зависящие от системы управления базами данных или версии, оказались недостаточными, остается возможность применения специфических методов, работающих в конкретной версии СУБД, таких, например, как использование особенностей алгоритмов оптимизации, частичное или полное отключение механизмов перебора планов, использование подсказок, если они поддерживаются в конкретной системе.

Главная опасность, возникающая при использовании методов, зависящих от СУБД, состоит в том, что эти приемы могут оказаться неработоспособными после смены версии сервера базы данных, и в этом случае, скорее всего, настройку

придется выполнять повторно. Методы, не зависящие от СУБД, в большей мере устойчивы к таким изменениям.

20.5.4. Целостная настройка приложений

Некоторые из современных методологий проектирования приложений приводят к появлению большого количества очень простых запросов, выбирающих значение одного атрибута из одного кортежа отношения. Такие запросы фактически невозможно оптимизировать на уровне базы данных, а их огромное количество делает работу приложения крайне неэффективной из-за непомерного объема накладных расходов, главным образом связанных с передачей данных по вычислительной сети. Значительную долю накладных расходов могут также составлять компиляция и подготовка запросов к выполнению.

В подобных случаях никакая настройка на уровне базы данных не может существенно улучшить производительность приложения — необходима переработка кода приложения с целью укрупнения запросов, передаваемых в СУБД.

Такой подход к настройке принято называть *сквозным* или *целостным* (holistic). Необходимость в сквозной настройке обычно обнаруживается, когда приложение уже некоторое время находится в эксплуатации, и поэтому его переработка оказывается почти невыполнимой по организационным и экономическим причинам. Решение может быть основано на автоматических средствах обнаружения и выделения фрагментов декларативного кода, пригодного для преобразования в запросы, из императивного (объектного) кода приложения.

Различные подходы к ручному и автоматическому решению задачи сквозной настройки описаны в [11; 25; 72].

20.6. Надежность и доступность

Напомним, что *доступностью* (availability) называют отношение времени, в течение которого система была в работоспособном состоянии, т. е. принимала и выполняла запросы пользователей, к общему (календарному) времени. Интервалы времени, в течение которых система была недоступна, могут быть вызваны как отказами оборудования или программных средства, так и плановыми остановками системы для проведения профилактических работ или модернизации.

Надежность системы определяется тем, насколько малы шансы потери данных. Количественно для оценки надежности можно использовать две характеристики: выживаемость, которая определяется как количество копий данных, которые должны быть разрушены, для того чтобы потеря данных произошла, и время, необходимое, для того чтобы изменения или новые данные попали в резервные копии.

Реалистическая оценка требований к доступности и надежности может оказаться непростым делом. Кроме самих характеристик, необходимо также анализировать гипотетические сценарии, которые могут повлиять на значения этих характеристик. Реализация высоких значений доступности и надежности приводит к существенному усложнению системы и значительно увеличивает ее стоимость, в том числе и стоимость сопровождения. Известны случаи применения как заниженных требований, так и завышенных. Проиллюстрируем это примерами.

Фраза «наш план действий в критической ситуации состоит в том, чтобы не иметь критических ситуаций», которую можно услышать от очень плохих менеджеров, звучит красиво, но не может рассматриваться как серьезное решение. Как ни странно, эту фразу приходилось слышать в одной из наиболее развитых в технологическом отношении стран. По-видимому, плохие управленцы встречаются везде, а для компенсации их усилий требуются меры, не относящиеся к категории технологических.

Другая крайность — завышение требований по доступности в рекламных целях. Допустим, произошло отключение электропитания на целом континенте (маловероятно, однако отключения сопоставимого масштаба за последние два десятилетия происходили несколько раз в разных частях мира). Для того чтобы обеспечить в этой ситуации доступность, необходимы автономные источники электропитания (например, дизель-генераторы), но должна ли информационная система оставаться доступной? Если от работы системы зависит функционирование операционной в больнице или работа реактора, то ответ, очевидно, положительный: должны быть проработаны схемы продолжения работы при отключении питания, независимо от масштабов отключения. С другой стороны, если основная функция системы связана с получением запросов по информационным сетям, то, скорее всего, в нашей гипотетической ситуации запросы пользователей не будут доставлены, поэтому в доступности системы нет большого смысла.

Не менее сложен правильный выбор уровня надежности. Выбор стратегии резервного копирования должен быть основан на реалистическом анализе рис-

ков потери данных и возможных последствий такой потери.

Стратегии обеспечения надежности и доступности взаимосвязаны. При отказе системы продолжительность восстановления влияет на доступность. Важно отметить, что надежность хранения связана также с политиками обеспечения безопасности: все копии данных, из которых можно восстановить состояние основной базы данных, должны быть защищены по крайней мере в такой же степени, как основное хранилище.

Средства обеспечения доступности и надежности, а также стратегии их применения рассмотрены в главе 14.

В распределенной системе надежность и доступность могут обеспечиваться для каждого узла распределенной системы отдельно, однако требуется уточнить понятие доступности. В строгом смысле система доступна, пока доступны все хранящиеся в ней данные, т. е. при отказе одного из серверов система считается доступной, если данные, хранящиеся на отказавшем сервере, имеются также на других серверах. Иначе говоря, для обеспечения высокой доступности необходима репликация.

Стратегии обеспечения надежности могут быть существенно упрощены, если для хранения данных используются облачные сервисы, поскольку эти сервисы предусматривают определенные гарантии надежности хранения. Конечно, такие сервисы не могут обеспечить ни абсолютную надежность, ни абсолютную доступность, но, скорее всего, эти гарантии достаточны для небольших систем.

Однако с точки зрения администрирования недостаточно просто регулярно создавать резервные копии базы. Кроме этого необходимо иметь отлаженные стратегии для восстановления базы из резервных копий. Полноценная стратегия резервного копирования и восстановления должна быть хорошо документирована и включать как минимум следующие пункты:

- периодичность выполнения резервного копирования;
- время запуска регулярного резервного копирования;
- список лиц, допущенных к выполнению резервного копирования и восстановления;
- объекты базы, подлежащие регулярному резервному копированию;
- точное указание места, где будут храниться резервные копии;
- сценарий выполнения резервного копирования;

- сценарий восстановления базы из резервной копии;
- периодичность пересмотра стратегии резервного копирования и восстановления базы.

Как бы ни различались сценарии выполнения резервного копирования и восстановления, их объединяет то, что они должны быть тщательно протестированы. И пусть в реальной жизни никогда не случится разрушение носителя и вам никогда не понадобится сценарий восстановления базы! Тем не менее наличие хорошо продуманной, документированной и оттестированной стратегии обеспечит защиту от отказов систем, человеческих ошибок и внешних обстоятельств.

20.7. Техническое обслуживание базы данных

Функции администратора базы данных, связанные с техническим обслуживанием базы, включают разнообразные задачи, которые необходимо решать для обеспечения нормальной работы системы. Многие из этих задач уже упомянуты в предыдущих разделах. В идеале эти задачи должны решаться так, чтобы они были невидимы для остального мира — для приложений, использующих базу данных, и, конечно, для пользователей. В реальности некоторые процедуры обслуживания могут потребовать временную приостановку доступа к базе данных.

Как правило, процедуры технического обслуживания выполняются фоновыми процессами, запускаемыми автоматически сервером баз данных или средствами операционной системы. Поэтому работа администратора сводится к планированию таких процедур, составлению расписания и, конечно, проверке того, что все процедуры действительно запускаются и работают корректно. Примером процедуры, обычно выполняемой автоматически, является удаление устаревших копий данных, которое в системе PostgreSQL выполняется процессом autovacuum.

Некоторые из функций обслуживания необходимо выполнять довольно редко, поэтому их автоматический запуск может оказаться нецелесообразным. В подобных ситуациях решения о запуске таких процедур принимаются на основе мониторинга базы данных. Примером таких задач может служить перестройка индексов в случае их деградации.

20.8. Итоги главы

Идеальное администрирование предполагает четко прописанные и протестированные сценарии по каждому из аспектов, описанных как в этой главе, так и в документации PostgreSQL. Это позволит избежать многочисленных проблем, которые возникают при эксплуатации приложений, работающих с базами данных, и обеспечит доверие к СУБД и предсказуемую среду для работы пользователей приложения.

20.9. Упражнения

Упражнение 20.1. Создайте нагрузку на базу данных при помощи утилиты pgbench. Определите по системным представлениям производительность системы (в транзакциях в секунду). Сравните с отчетом, который выводит pgbench в конце работы.

Упражнение 20.2. Создайте нагрузку с помощью pgbench и настройте конфигурационные параметры сервера, упомянутые в разделе 20.5.1, чтобы добиться увеличения пропускной способности системы.

Упражнение 20.3. Подумайте, какие индексы было бы полезно создать, чтобы ускорить выполнение запросов из упражнений к главе 4. Проверьте ваши предположения.

Упражнение 20.4. Дан следующий запрос к демобазе:

```
SELECT *
FROM tickets t
WHERE to_tsvector(t.passenger_name) @@
      to_tsquery('PAVEL & IVANOV')
AND bookings.now() - interval '10 days' < (
      SELECT b.book_date
      FROM bookings b
      WHERE upper(b.book_ref) = upper(t.book_ref)
    );
```

Перепишите его с учетом замечаний, сформулированных в разделе 20.5.3. Сравните скорость выполнения запроса до и после модификации.

Упражнение 20.5. Опишите конфигурацию системы, имеющей запасной сервер в теплом резерве, и составьте инструкцию по восстановлению работоспособности системы в случае разрушения носителя на основном сервере.

Глава 21

Репликация баз данных

21.1. Множественные копии данных

Идея избыточного дублирования часто встречается в природе и широко применяется в самых различных областях человеческой деятельности. Дублировать можно любое оборудование, сервисные ресурсы, информационные сообщения и многое другое, необязательно связанное с вычислительной техникой. В живых организмах могут дублироваться как отдельные органы, так и — на клеточном уровне — геномы. Цели дублирования могут быть разными, но в большинстве случаев дублирование позволяет снизить риски, связанные с возможными потерями, или повысить эффективность. Независимо от целей, улучшение характеристик достигается за счет применения дополнительных ресурсов.

В случае баз данных дублирование сводится к организации хранения нескольких копий логически одних и тех же данных. В настоящее время в технической литературе используется термин *репликация*, представляющий собой кальку с английского replication. На это требуются избыточные ресурсы, использование которых позволяет улучшить ряд характеристик системы. Наиболее важными из них являются:

- надежность — данные не будут потеряны при разрушении части копий;
- доступность— обработка будет возможна, даже если некоторые копии временно недоступны;
- производительность — одновременная работа нескольких ресурсов может обслужить большее количество запросов или ускорить выполнение отдельных запросов.

Средства, необходимые для достижения этих целей, могут различаться. Так, для повышения производительности на уровне отдельных запросов (сокращения времени отклика) необходимо, чтобы поддерживалось распределенное параллельное выполнение запросов, обсуждаемое в главе 22. Но оно не требуется

ни для повышения надежности, ни для улучшения доступности, ни для увеличения пропускной способности системы.

Для того чтобы подобные улучшения характеристик были возможны, необходимо размещать разные копии данных на различных устройствах, которые могут работать независимо друг от друга. Следовательно, система с репликацией данных должна быть распределенной.

В принципе, в распределенной системе могли бы одновременно применяться и репликация, и фрагментация данных (рассматриваемая в главе 22). Например, большие таблицы могут быть фрагментированы, а небольшие — реплицированы на несколько вычислительных систем. Однако во многих СУБД средства репликации обеспечивают копирование всей базы данных или всего сервера; например, в системе PostgreSQL копирование выполняется на уровне кластера баз данных. Многие из этих механизмов не приспособлены для частичной репликации (скажем, отдельных таблиц). В этой главе обсуждаются только конфигурации распределенных баз данных с полной репликацией.

Конечно, для достижения любой из перечисленных целей одной только репликации недостаточно. Так, копии баз данных, размещенные на серверах с общим источником электропитания, не могут обеспечить доступность при отключении электричества. Серверы, размещенные в одном центре обработки данных, не могут обеспечить надежность при разрушениях вследствие стихийных или техногенных бедствий. Мы, однако, не будем рассматривать подобные проблемы в этой главе.

21.2. Согласованность реплик

Все упомянутые выше цели репликации никак не связаны с функциями системы, в которой репликация применяется. Поэтому в идеале репликация должна быть прозрачной, т. е. результаты, получаемые пользователем, не должны зависеть от того, какие копии данных были использованы для получения этого результата. Другими словами, поведение базы данных, для которой применяется репликация, с точки зрения приложений (и пользователей) не должно отличаться от поведения одной централизованной базы данных. Такое требование к системе называется требованием *единой логической копии*.

Говорят, что реплики *согласованы*, если они содержат идентичные значения всех элементов данных. Требование единой логической копии является одним

из возможных условий согласованности реплик. Можно также говорить, что выполнение этого требования обеспечивает полную согласованность реплик.

Важно подчеркнуть, что понятие согласованности реплик отличается от понятия согласованности, рассматриваемого в главе 13, которое основано на предположении об атомарности операций чтения и записи данных и при конкурентном обновлении обеспечивает корректность данных в смысле какого-либо из критериев, описанных в этой главе. Чтобы отличать такую согласованность от согласованности реплик, будем называть ее *абстрактной согласованностью*.

Неформально можно сказать, что абстрактная согласованность необходима для учета зависимостей между различными элементами данных, которые устанавливаются приложением (при модификации нескольких элементов данных в одной транзакции). В системах с репликацией операции записи не могут считаться атомарными, поскольку значения каждого элемента данных хранятся на нескольких серверах. Поэтому понятие согласованности реплик дополняет понятие абстрактной согласованности.

В общем случае разные реплики базы данных могут хранить различающиеся версии всех или некоторых элементов данных. Какие именно различия допускаются и какие из этих различий могут быть видны приложениям, зависит от требований согласованности реплик, которым удовлетворяет система.

Требование единой логической копии является наиболее сильным условием согласованности реплик. Оно важно, для того чтобы корректно выполнять распределенные запросы и распределенные транзакции. Это требование обеспечивает *глобальную согласованность* данных, но его буквальная реализация (поддержка идентичности всех реплик) оказывается ресурсоемкой, может ограничивать доступность и ухудшать характеристики производительности системы. В частности, такая реализация может оказаться неустойчивой к разделению сети, т. е. к потере связи с частью реплик.

Можно ослабить требование единой логической копии, применяя его не ко всем приложениям и транзакциям одновременно, а к каждой транзакции отдельно. При таком понимании этого требования для реализации требуется только согласованность реплик, участвующих в выполнении транзакции (а не полная идентичность всех реплик), и такая согласованность может распространяться только на те элементы данных, которые читаются или модифицируются рассматриваемой транзакцией. При этом разные транзакции, даже выполняемые конкурентно, могут наблюдать разные состояния базы данных, однако (глобальная) абстрактная согласованность по-прежнему будет сохранена.

Если ни распределенные запросы, ни распределенные транзакции в системе не используются, то требование единой логической копии можно ослабить, заменив его на требование *локальной согласованности* всех реплик. При этом каждая копия предоставляет приложениям некоторое согласованное состояние, однако не требуется, чтобы это состояние включало все транзакции, выполненные в системе (возможно, на других репликах). Если требование локальной согласованности выполняется, то любая только читающая транзакция может быть корректно выполнена на любой из копий, хотя, возможно, она будет выполнена так, как будто она выполнялась несколько раньше, чем в реальности.

Локальная согласованность полезна в сценариях, в которых оперативная работа выполняется на одной копии или на ограниченном количестве копий, а остальные используются для аналитической обработки, т. е. для решения задач, в которых предельная актуальность данных не требуется.

Важной составной частью любых условий согласованности реплик являются правила обновления и распространения изменений по репликам. Для поддержки требования единой логической копии необходимо все изменения выполнять одновременно на всех репликах. Как для ослабленного требования единой логической копии, так и для локальной согласованности изменения допускаются только на репликах, содержащих самую последнюю версию обновляемого элемента данных. При этом распространение изменений на другие реплики может происходить с некоторой задержкой.

Такие правила обновления обеспечивают полную упорядоченность версий каждого элемента данных и совместимость упорядочений версий разных элементов данных, а также гарантируют невозможность потери обновлений. В результате поведение системы, видимое приложениями, оказывается похожим на поведение системы, в которой диспетчер транзакций использует множественные версии (как описано в главе 13). Отметим, что для ослабленного варианта требования единой логической копии существуют протоколы, устойчивые к разделению сети и обеспечивающие очень высокую доступность на чтение. Такие протоколы рассматриваются в разделе 21.5.

Известно еще несколько критериев согласованности, более слабых, чем локальная согласованность. Многие из них не включают ограничения на модификацию данных, что может приводить к потере обновлений. Упомянем только *отложенную согласованность* (eventual consistency), которая гарантирует, что изменения будут рано или поздно распространены на все копии, однако не предоставляет никаких гарантий для приложений при чтении значений и не

предотвращает потерю обновлений в случае, если на других репликах обновления были выполнены позже. По существу, это означает, что поддержка согласованности перекладывается на приложение.

21.3. Согласованность, доступность, разделение сети

Характеристики доступности применимы к значительно более широкому классу систем, чем системы управления базами данных, однако точное определение того, как понимается доступность, может различаться в зависимости от класса систем и даже от типа выполняемых действий. В любом случае сервер считается доступным, если он в состоянии выполнять запросы клиентов, но даже применительно к базам данных необходимо различать доступность для чтения и доступность для обновления данных.

Термин «согласованность» также используется в различных смыслах. Так, согласованность может рассматриваться для распределенных систем в отрыве от понятия транзакции. В этом контексте была опубликована и доказана [31] «теорема CAP», утверждающая, что невозможно получить распределенную систему, устойчивую к разделению вычислительной сети и одновременно обладающую высокой доступностью и обеспечивающую согласованность.

Эта теорема получила широкую известность, и часто на нее ссылаются как на основание для реализации систем с ослабленными требованиями по согласованности реплик якобы для того, чтобы добиться высокой доступности. В действительности такие ссылки на теорему некорректны, потому что условия теоремы не выполняются в реальных условиях. Критическое обсуждение такой практики и применимости теоремы можно найти в [39].

Определение согласованности, на котором основана теорема, предполагает, что система реализует единую логическую копию данных (любые изменения немедленно распространяются на все реплики) и линеаризуемость операций (каждая операция видит все изменения, сделанные любой предшествующей операцией). Одновременно предполагается, что каждая операция читает или обновляет только один элемент данных.

В реальных СУБД, использующих транзакции, требуется согласованность (как абстрактная, так и согласованность реплик) только для множеств элементов данных, читаемых или записываемых транзакцией, а не всей базы данных. Более того, при конкурентном выполнении большого количества транзакций база

данных практически никогда не находится в полностью согласованном состоянии. Далее, для обеспечения логической корректности как базы данных, так и работы приложений достаточно ослабленного варианта требования единой копии данных, а при отсутствии распределенных транзакций и запросов достаточно локальной согласованности с ограничением на обновление.

Определение доступности, необходимое для справедливости теоремы, также оказывается более сильным, чем требуется в реальных системах. В частности, только читающая транзакция корректна, даже если она читает устаревшие, но локально согласованные версии данных. Такие транзакции могут выполняться нормально при разделении сети и на тех репликах, которые временно утратили связь с остальными.

Современные технологии СУБД обеспечивают возможность создания высокопроизводительных и высокодоступных распределенных баз данных, способных обрабатывать миллионы транзакций в секунду, не жертвуя при этом согласованностью.

21.4. Поддержка единой логической копии

Для того чтобы выполнить требование одной логической копии, необходимо все изменения, вносимые в базу данных, выполнять одновременно на всех копиях, или, более точно, изменения, вносимые любой транзакцией, должны быть выполнены на всех серверах до того, как эта транзакция будет зафиксирована. Возможны различные методы реализации такого подхода:

Глобальные транзакции. Все транзакции сериализуются в рамках одного расписания, например, используются глобальные блокировки.

Главная копия. Все обновляющие транзакции выполняются на одном сервере, и внесенные ими изменения распространяются на все остальные копии. После этого обновляющие транзакции могут быть зафиксированы.

В любом случае система, работающая в таком режиме, не может обеспечить высокую доступность для обновлений, потому что отказ любого из серверов приводит к невозможности выполнения каких-либо обновлений во всей системе. Требование одной копии при этом выполняется, и высокая доступность по чтению обеспечивается.

21.5. Симметричные протоколы синхронизации реплик

Для того чтобы увеличить доступность при обновлении, был предложен *мажоритарный протокол* [10], позволяющий выполнять обновляющие транзакции при частичной недоступности копий. При использовании этого протокола требование одной копии заменяется на более слабое: любой сервер должен предоставлять согласованное состояние данных, но, возможно, несколько устаревшее, т. е. не учитывающее часть транзакций, выполненных на других серверах. Таким образом, этот протокол обеспечивает локальную согласованность.

Пусть общее количество копий равно S. Мажоритарный протокол разрешает выполнить обновление только в том случае, если количество копий, доступных для обновляющего сервера, составляет не менее $S/2 + 1$. Для того чтобы обновить значение некоторого элемента данных, выполняются следующие шаги:

1. Значение элемента, подлежащего обновлению, считывается со всех доступных копий. Если количество считанных копий меньше чем $S/2 + 1$, то обновление невозможно и транзакция обрывается.
2. Из всех полученных значений выбирается самая последняя версия, и выполняется ее обновление в соответствии с запросом приложения.
3. Новая версия элемента распространяется на все доступные серверы.

Корректность мажоритарного протокола основана на том, что любые два подмножества, содержащие не менее чем $S/2 + 1$ элементов, обязательно имеют непустое пересечение. Поэтому среди значений, полученных на первом шаге, обязательно будет хотя бы одна копия самого последнего изменения, выполненного предыдущими транзакциями.

Если серверу не доступно достаточное для обновления количество копий, он может тем не менее выполнять запросы на чтение, возвращая те версии данных, которые на нем имеются. Поэтому мажоритарный протокол может обеспечить высокую доступность при неустойчивой работе вычислительных сетей.

Вариант этого протокола для распределенных систем (не обязательно баз данных) описан в [41].

Важными преимуществами этого класса протоколов является отсутствие единой точки, отказ которой приводит к остановке всей системы, и возможность внесения изменений на нескольких серверах.

Система PostgreSQL (возможно, пока) не содержит никаких средств для поддержки этого класса протоколов, но они используются в системах, надстраиваемых над PostgreSQL для обеспечения возможности выполнения обновлений на нескольких копиях.

21.6. Репликация главной копии

Наиболее часто используется организация репликации, при которой изменения могут выполняться только на одном *главном* (primary) сервере. Изменения, выполненные на этом сервере, переносятся на все копии, которые могут использоваться только для чтения данных и называются *запасными* (standby) или *репликами* (replica). Чтобы подчеркнуть неравноправие главной и запасных копий, возможно, следовало бы по-русски называть реплики «отражениями», что соответствует одному из значений английского термина replica и достаточно точно, хотя и образно, передает суть дела.

Очевидно, что такие методы репликации не могут обеспечить доступность для модификации выше, чем доступность централизованной базы данных. Доступность на чтение может быть весьма высокой, если чтение устаревших копий допустимо.

Существуют разнообразные методы распространения обновлений с главного сервера на запасной. Разные методы по-разному влияют на доступность серверов, на отставание состояния запасного сервера от состояния основного, и на нагрузку, которую они создают на этих серверах. Поэтому нет какого-либо одного варианта, который превосходил бы все остальные. Выбор метода распространения обновлений зависит от требований к системе.

Распространение обновлений может быть синхронным или асинхронным. При *синхронном распространении* все изменения, выполненные на главном сервере, повторяются на запасных и должны быть завершены до того, как будет зафиксирована транзакция, выполнившая эти изменения на главном сервере. Достоинства и недостатки синхронного распространения уже перечислены выше: основной недостаток этой схемы состоит в том, что при отказе запасного сервера обновления на главном становятся невозможными. Отметим также, что при синхронном распространении время, необходимое для фиксации транзакции, может существенно увеличиться, что приведет к снижению пропускной способности основного сервера.

При *асинхронном распространении* обновлений влияние репликации на основной сервер будет значительно меньше, но состояние базы данных на запасных серверах будет повторять состояние главного с некоторой задержкой. Величина этой задержки зависит от выбранного метода распространения обновлений и от конфигурации этого метода.

В некоторых случаях целесообразно ввести дополнительную задержку искусственно. Дело в том, что наиболее частой причиной неполадок в работе системы оказываются не отказы или сбои оборудования, а человеческие ошибки. Зачастую такие ошибки не приводят к остановке системы и обнаруживаются не сразу. В таких случаях может иметь смысл сначала восстановить данные из корректной, хотя и устаревшей копии (например, отдельных таблиц), а затем их актуализировать. Если на запасном сервере обновления применяются с задержкой в несколько часов, такой сервер может использоваться как источник неповрежденных данных. Конечно, переход на такой сервер при отказе основного потребует больше времени и это повлияет на доступность системы. Для предотвращения задержек при переходе на запасной сервер можно поддерживать два запасных сервера, один из которых обновляется немедленно, а другой — с задержкой.

Распространение обновлений можно выполнять на различных уровнях абстракции данных.

Повторение операторов SQL. В этом случае запасной сервер будет фактически повторять всю работу, которая выполняется на главном. Этот метод применяется крайне редко.

Обновление логических записей. При использовании этого метода обновления пересылаются в виде операторов SQL, каждый из которых обновляет ровно одну логическую запись (например, строку таблицы) по ее уникальному идентификатору. Как правило, в этом случае нагрузка на запасной сервер значительно ниже, чем в предыдущем, однако массовые операции обновления на основном сервере приводят к генерации очень большого количества обновлений отдельных записей. Выполнение этих операторов может занимать очень большое время.

В системе PostgreSQL вместо операторов SQL передаются отдельные строки, что дает экономию на синтаксическом анализе, однако, как и в предыдущем случае, операции массового обновления приводят к задержкам распространения обновлений.

Распространение на уровне страниц базы данных. При копировании физических страниц базы данных нагрузка на запасной сервер снижается до минимума, поскольку нет необходимости не только в синтаксическом разборе и выполнении операторов SQL, но и в какой-либо модификации страниц, поступающих с главного сервера. Более того, при таком распространении объем работы, выполняемый на запасном сервере, не может превосходить объем работы, выполненный на главном, поэтому операторы массового обновления никаких проблем не создают.

Для того чтобы распространение на физическом уровне было возможно, в большинстве СУБД требуется, чтобы операционные системы на обоих серверах были почти идентичными и чтобы размещение базы данных в файловой системе сервера также совпадало (т. к. некоторые страницы базы данных могут содержать пути файловой системы).

Распространение записей журнала. Этот вариант используется средствами репликации PostgreSQL. По своим характеристикам он близок к предыдущему. Отличие в том, что в этом случае страницы не копируются полностью, если они не содержатся в журнальных записях. Вместо этого используются записи REDO, как при восстановлении после отказа (см. главу 14).

Рассмотрим основные случаи, в которых применяется репликация по схеме с одним главным сервером.

- Создание копий, отражающих состояние базы данных на основном сервере на некоторый момент времени без последующего распространения изменений. Строго говоря, создание таких копий нельзя считать репликацией. Наиболее очевидное и широко распространенное применение таких копий — разработка и тестирование новых версий приложений, работающих с этой базой данных. Конечно, разработчики должны иметь возможность выполнять операторы обновления, поэтому такие копии не могут получать обновления с главного сервера. При необходимости копии заменяются на новые.

  Заметим, что для такого применения, как правило, необходима модификация копии, поскольку база данных может содержать информацию, не подлежащую распространению (например, персональные данные пользователей системы).

- Снижение нагрузки на главный сервер и повышение производительности системы за счет переноса только читающих приложений на запасные серверы.

- Распределение запросов в зависимости от типов нагрузки: все обновляющие запросы и запросы типа OLTP выполняются на одном сервере, а аналитические запросы типа OLAP — на другом или на других. По существу это типичный частный случай предыдущего.
- Создание резервных копий базы данных для использования в качестве главной копии в случае отказа основного сервера или разрушения носителя данных на нем. Как правило, такое использование репликации сочетается с предыдущим.

Для каждого из этих случаев применения существует много различных вариантов, однако мы не будем их рассматривать более детально. Основное различие между методами создания копий состоит в том, каким образом выполняется распространение изменений после создания запасной копии.

21.7. Резервные серверы базы данных

Как указано в главе 14, механизм репликации применяется для создания систем с очень высокой доступностью, которая достигается за счет быстрого перевода запасного (резервного) сервера в режим главного. Часто упоминаемым требованием к высокодоступным системам считается «пять девяток», т. е. 0,99999 или 99,999 %. Такая доступность означает, что суммарное время недоступности системы в течение года не превышает 316 секунд за год или 6 секунд в неделю.

Схема репликации с главным сервером вполне подходит для создания резервных копий, поскольку в этом случае все изменения распространяются в одном направлении. На запасном оборудовании запускается сервер баз данных с включенным режимом восстановления по журналу (как при восстановлении после разрушений носителя данных). По мере создания новых архивированных сегментов журнала содержащиеся в них изменения повторяются на запасном сервере.

В случае отказа основного сервера на запасном обрабатывается последний файл журнала и после этого он переводится в режим основного сервера. Запасной сервер в такой конфигурации называется сервером *теплого резерва* (warm standby). Для того чтобы применять такую схему, необходимо, чтобы файлы журнала основного сервера были доступны даже в случае его отказа. Для этого файлы журнала записываются в нескольких копиях, размещенных на разных вычислительных системах. Достоинством конфигурации с запасным сервером

в таком режиме является то, что она не создает дополнительную нагрузку на основной сервер, а недостатком можно считать относительно большое время, необходимое для переключения запасного сервера в режим основного.

В системе PostgreSQL (и в других высокопроизводительных системах) поддерживается и другая конфигурация, которая называется сервером в *горячем резерве* (hot standby). В этой конфигурации запасной сервер устанавливает соединение с основным (как клиент) и получает поток сообщений с информацией об изменениях, выполняемых в базах данных основным сервером. Эти изменения практически немедленно (но асинхронно) выполняются и на запасном.

Система PostgreSQL предоставляет более широкий спектр конфигураций серверов в горячем резерве. Обновления могут пересылаться без подтверждения, с подтверждением доставки и с подтверждением применения. Если поддерживается более одного резервного сервера, можно задать режим, при котором подтверждение требуется от меньшего количества серверов. Такая конфигурация позволяет исключить задержки и повысить доступность основного сервера в случаях потери сообщений или отказа одного из резервных серверов.

Поскольку в конфигурации с горячим резервным сервером все изменения, выполненные зафиксированными транзакциями на основном сервере, почти сразу применяются и на запасном, время переключения в случае отказа основного сервера оказывается минимальным. Обычно допускается использование сервера, находящегося в горячем резерве, для выполнения только читающих транзакций.

21.8. Репликация в системе PostgreSQL

Наиболее простой способ создания копии в системе PostgreSQL состоит в использовании программ pg_dump и pg_dumpall. Эти программы позволяют создавать согласованные копии в различных форматах, допускают полное или частичное копирование кластера, базы данных или отдельных схем. Копирование выполняется на логическом уровне, поэтому некоторые из форматов выгруженных данных можно использовать для загрузки на сервер другой конфигурации, возможно, работающий под управлением другой операционной системы или другой СУБД. Эти программы устанавливают соединение с сервером баз данных, как обычные клиенты, и не накладывают никаких ограничений на конфигурацию сервера. Однако в любом случае созданные копии не подлежат использованию в качестве запасных, поскольку они не предназначены для

распространения изменений, внесенных в базы данных после создания копии. Поэтому создание копий с помощью подобных средств вряд ли можно назвать репликацией.

В методах создания копий, пригодных для репликации с распространением обновлений применяется программа pg_basebackup, которая создает копию кластера баз данных. Один из вариантов размещения копии — в файловой системе на другом вычислительном сервере. Такая копия готова для запуска на ней сервера PostgreSQL и может использоваться как независимая база данных.

В системе PostgreSQL существует несколько различных способов организации автоматического распространения изменений с главного сервера на запасные. Один из них основан на *передаче файлов (сегментов) журнала*. Это можно сделать включением режима непрерывного архивирования: при переключении сегментов журнала регистрации изменений вызывается процедура архивирования только что законченного сегмента. Эта процедура должна сохранить данные из сегмента журнала, например просто скопировать его в другую файловую систему, в которой эти файлы накапливаются. Возможно также получение копий сегментов журнала с помощью программы pg_receivewal, которая не дожидается заполнения сегмента, а получает поток записей журнала, устанавливая соединение с главным сервером. Для нормальной работы репликации с распространением обновлений необходимо, чтобы после создания копии с помощью pg_basebackup все записанные (архивированные) файлы журнала оставались доступными.

При запуске сервера баз данных на копии выполняется процесс восстановления, как если бы кластер баз данных восстанавливается после разрушения носителя данных. В отличие от обычного восстановления запасной сервер после обработки всех файлов журнала остается в режиме восстановления и продолжает обрабатывать данные из журнала на главном сервере по мере их возникновения.

При использовании описанного выше метода отставание запасного сервера от главного может быть значительным, зато почти не создается дополнительная нагрузка на главный сервер.

Значительно меньшее отставание может обеспечить *непосредственная связь между серверами*. Возможны различные конфигурации запасного сервера.

- Запасной сервер создает сеанс работы с главным сервером, в рамках которого запрашивает записи журнала главного сервера и обрабатывает их немедленно или с некоторой задержкой.

- Синхронная работа главного и подчиненного серверов. Как и в предыдущем варианте, запасной сервер создает сеанс связи с главным, но если запасной сервер включен в список синхронных, то главный сервер выполняет передачу изменений в рамках транзакции, которая выполнила эти изменения. При этом транзакция фиксируется, но ее завершение задерживается до получения подтверждения с запасного сервера.

В любом из этих случаев запасной сервер может выполнять только читающие транзакции (которые, конечно, будут получать результаты только на основе тех изменений, которые запасной сервер успел обработать до начала транзакций).

Изменения одного главного сервера могут распространяться на несколько запасных. Отношение главный — запасной является относительным: запасной сервер может выступать в роли главного по отношению к другим серверам. Поэтому структура системы, использующей репликацию, может представлять собой дерево с главным сервером в корне, а распространение изменений происходит вдоль путей от корня к листьям. При этом способ распространения может выбираться для каждого ребра этого дерева независимо от других ребер. Например, один из двух запасных серверов может обновляться немедленно, а второй — с некоторой задержкой.

21.9. Итоги главы

Репликация представляет собой инструмент для улучшения некоторых эксплуатационных характеристик системы баз данных. Чаще всего репликацию применяют для повышения доступности данных на чтение и для увеличения производительности. В то же время репликация неизбежно усложняет обновление данных и поддержку их в согласованном состоянии. Известные схемы репликации либо ограничивают возможности обновления базы данных, разрешая их только на одной из копий, либо оказываются весьма тяжеловесными по ресурсам и ограничивающими доступность системы для обновлений. Некоторые программные средства поддержки репликации, допуская обновление нескольких копий, перекладывают ответственность за корректность данных (и, соответственно, необходимую для этого вычислительную нагрузку) на приложения.

Репликация применяется также для обеспечения защиты от разрушения носителей данных и быстрого восстановления с переносом нагрузки на запасной сервер.

21.10. Упражнения

Упражнение 21.1. Создайте копию базы данных с помощью утилиты pg_dump. Загрузите эту копию в другую базу данных на том же кластере. Загрузите эту же копию в другую СУБД. Объясните результаты.

Упражнение 21.2. Постройте систему из двух серверов с репликацией кластера баз данных, используя:

1) пересылку архивированных файлов журнала;
2) асинхронное распространение обновлений;
3) синхронное распространение обновлений.

Продемонстрируйте задержки при распространении обновлений.

Упражнение 21.3. Создайте распределенную систему реплицированных баз данных, допускающую обновления на всех системах и обеспечивающую только отложенную согласованность (eventual consistency). Реализуйте полную согласованность на уровне приложения.

Глава 22

Параллельные и распределенные СУБД

22.1. Архитектуры параллельной и распределенной обработки

Идея использования нескольких взаимосвязанных вычислительных систем для решения одной задачи или нескольких взаимосвязанных задач обсуждается уже несколько десятилетий. Ранние системы параллельной обработки появились ненамного позже электронных вычислительных систем. Первые параллельные и распределенные системы управления базами данных появились в начале 80-х гг., а к середине 90-х были проработаны основные принципы их организации и алгоритмы обработки, применимые на различных конфигурациях аппаратных средств, включающих несколько вычислительных систем.

К настоящему времени значение систем параллельной и распределенной обработки существенно выросло в связи с тем, что:

- возможности увеличения производительности последовательных вычислительных устройств близки к исчерпанию;
- пропускная способность и надежность вычислительных сетей создают условия для эффективной совместной работы даже географически удаленных друг от друга систем.

Применительно к системам обработки данных принято различать параллельные и распределенные системы. Четкой границы между этими классами систем не существует, поэтому в некоторых публикациях эти термины используются как взаимозаменяемые. Некоторые конфигурации оборудования допускают их применение как для параллельных, так и для распределенных СУБД.

В этой книге мы будем различать параллельные и распределенные системы баз данных по целям, для достижения которых эти системы баз данных создаются: основной целью развертывания *параллельной системы* баз данных является

улучшение характеристик производительности, в то время как основной целью создания *распределенной системы* является повышение доступности.

Конфигурация параллельного сервера баз данных схематически представлена на рис. 22.1.1. С точки зрения приложения, работающего в роли клиента, параллельный сервер баз данных не отличается от обычного сервера баз данных (который в этом контексте принято называть *централизованным*). Различие между параллельным и централизованным серверами баз данных состоит только в том, что параллельный сервер использует для выполнения запросов клиентов несколько устройств (процессоров, памяти и дисков), способных работать параллельно.

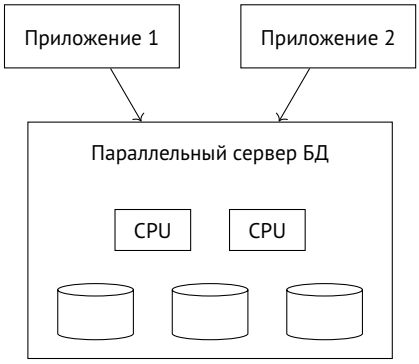


Рис. 22.1.1. Параллельный сервер базы данных

Данные, размещаемые в такой системе, описываются одной схемой (для каждой базы данных), даже если эти данные размещены на разных устройствах.

Основной характеристикой, применяемой для оценки параллельных систем, является *масштабируемость* (scalability). Напомним, что масштабируемость является относительной характеристикой, показывающей соотношение некоторой меры производительности (чаще всего пропускной способности или времени отклика) на централизованной и параллельной системах.

Необходимо подчеркнуть, что само по себе увеличение количества установленного оборудования не обязательно приводит к улучшению характеристик производительности. Для того чтобы использовать возможности параллелизма, необходимо учитывать особенности оборудования и требования прикладной системы при проектировании базы данных, в первую очередь структур хранения данных. Более детальное обсуждение параллельных серверов баз данных содержится в разделе 22.2.

В отличие от параллельных серверов баз данных распределенные системы представляют собой совокупность серверов баз данных, каждый из которых может работать независимо от остальных серверов этой системы. При этом сервер, входящий в состав распределенной СУБД, может выполнять запросы, используя данные, размещенные на нескольких серверах. Пример возможной конфигурации распределенной системы серверов баз данных показан на рис. 22.1.2.

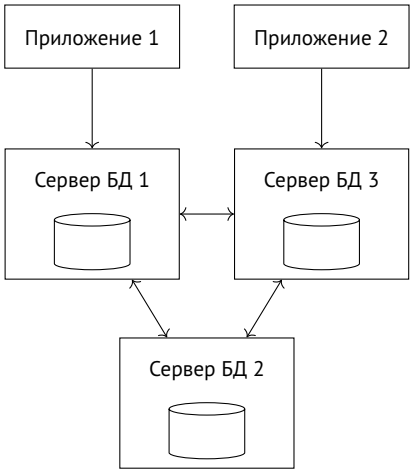


Рис. 22.1.2. Распределенная система баз данных

Для каждой из баз данных, входящих в состав распределенной системы, поддерживается своя схема данных. Для того чтобы на одном сервере использовать данные, размещенные на другом сервере, необходимо включить в схему этого сервера определения структур данных другой (внешней) системы. Обычно, в том числе и в PostgreSQL, такие данные называются *сторонними* или *внешними* (foreign). Распределенные системы баз данных обсуждаются в разделе 22.3.

Однако в более широком сообществе специалистов термин «распределенная система» имеет совсем другой смысл. Когда говорят о распределенных системах, обычно рассматривают конфигурации, в которых приложения, выполняемые на разных серверах, обмениваются данными путем передачи сообщений, а используемые ими серверы баз данных непосредственно друг с другом не взаимодействуют. Приложения такого типа обычно предоставляют функциональность в виде сервисов.

Применение архитектур на основе сервисов может приводить к дублированию функций СУБД на уровне приложения (например, становится невозможной

проверка согласованности на уровне базы данных) и к существенному снижению производительности, однако возможность изоляции сервисов оказывается более важной.

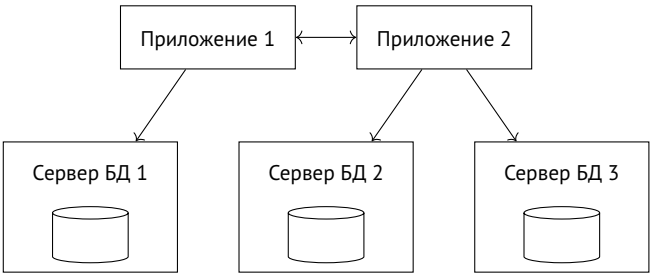


Рис. 22.1.3. Распределенная система сервисов

Организация распределенной системы сервисов иллюстрируется рис. 22.1.3.

22.2. Параллельные серверы баз данных

22.2.1. Конфигурации оборудования

Детальное рассмотрение параллельных серверов баз данных следует начать с конфигураций оборудования. Принято различать следующие разновидности архитектур вычислительных систем, на которых могут работать параллельные серверы баз данных:

SM Многопроцессорные системы с разделяемой памятью (shared memory), иногда называемые симметричными многопроцессорными системами. В таких архитектурах диски также являются общими.

SD Системы с общей дисковой памятью (shared disks). В этой конфигурации несколько вычислительных систем взаимодействуют между собой и с устройствами дисковой памяти через локальную вычислительную сеть.

SN Системы без общих ресурсов (shared nothing). В такой архитектуре каждая вычислительная система не имеет доступа к устройствам других вычислительных систем; все взаимодействия и обмены данными выполняются через вычислительную сеть.

В архитектурах SM и SD нет необходимости в пересылке данных между системами по вычислительной сети, т. к. все хранимые данные непосредственно доступны всем процессорам. Поэтому операции могут распределяться между процессорами так, чтобы по возможности добиться балансировки нагрузки (что не является простой задачей).

Важно заметить, что многоядерные процессоры невозможно использовать для организации параллелизма в СУБД в точности так же, как многопроцессорные системы. Причина состоит в том, что различные ядра одного процессора используют общий кеш, поэтому при больших объемах обрабатываемых данных суммарная производительность всех ядер ограничивается пропускной способностью оперативной памяти. Из этого, конечно, не следует, что возможности многоядерных процессоров нельзя использовать в СУБД, но для этого необходимы другие алгоритмы, учитывающее особенности иерархии памяти.

Возможности масштабирования систем типа SM ограничиваются количеством процессоров, которые можно установить в одной вычислительной системе.

В системах SD необходимы средства синхронизации для предотвращения конфликтующих изменений страниц разными серверами. Параллельные серверы на основе архитектуры SD в некоторой мере обладают функциями распределенных: в случае отказа одной из вычислительных систем, входящих в (аппаратный) кластер, работа может продолжаться, хотя и с несколько уменьшенной производительностью. При этом, однако, база данных остается общей, поэтому такая конфигурация не обеспечивает восстановление в случае разрушения носителя (в отличие от средств репликации, рассмотренных в главе 21).

В системах SN для выполнения запросов обычно требуется значительное количество пересылок данных по сети, что должно учитываться при оптимизации и распределении запросов по вычислительным системам. Достоинством таких систем считается потенциально неограниченная возможность горизонтального масштабирования.

22.2.2. Гранулярность параллелизма

Использовать возможности параллельной обработки можно на различных уровнях:

- между транзакциями;
- внутри транзакций между запросами;

- внутри запросов между операциями, входящими в план;
- в алгоритмах выполнения операций.

Организация параллельного выполнения транзакций не требует специальных мер, кроме низкоуровневой поддержки блокировок. Заметим, что разные операторы одной транзакции могут направляться на разные процессоры. По существу использование параллелизма на этом уровне ограничивается балансировкой нагрузки между вычислительными системами. В системах без общих ресурсов, однако, распределение операторов по вычислительным системам зависит от размещения данных: операторы по возможности направляются туда, где находятся необходимые им данные.

Параллелизм между запросами внутри транзакций возможен, если логика приложения это допускает и приложение использует асинхронные средства взаимодействия с базой данных. Например, возможно параллельное выполнение независимых операций чтения, условия поиска в которых не зависят от результатов других операций чтения. Во многих случаях операции чтения можно запускать, не дожидаясь завершения предшествующих операций записи.

Параллельное выполнение операций, входящих в план одного запроса, потенциально обеспечивается потоковой передачей данных между операциями плана. Для того чтобы использовать параллелизм, операции могут назначаться на различные процессоры. Если эти процессоры не имеют общей памяти, поток данных должен передаваться по вычислительной сети, что может увеличить общую стоимость выполнения запроса и поэтому должно учитываться при оптимизации. Кроме этого, некоторые операции являются блокирующими, т. е. фактически следующая операция может начать выполнение только с некоторой задержкой или после полного завершения блокирующей операции.

Далеко не все СУБД используют эту потенциальную возможность. В частности, в системе PostgreSQL применяется другой подход к параллельному выполнению запросов. Оптимизатор PostgreSQL может выделить подзапрос, который будет выполняться параллельно несколькими процессами. Один из процессов, участвующих в выполнении, собирает результаты работы остальных процессов и выполняет оставшуюся последовательную часть запроса.

Наконец, параллелизм на уровне отдельных операций плана предполагает применение алгоритмов, допускающих параллельное выполнение. Для большинства реляционных операций существуют алгоритмы, допускающие высокоэффективное выполнение на параллельных системах. Такие алгоритмы обсуждаются далее в этом разделе.

22.2.3. Размещение данных

Для того чтобы использовать возможности параллельной обработки при выполнении операций выборки хранимых данных, необходимо распределить данные по нескольким устройствам, которые могут работать параллельно.

Выбор способа распределения данных по разным носителям зависит от того, как именно должен масштабироваться сервер баз данных (какого типа нагрузки наиболее важны), и от характеристик самих устройств.

Так, если сегменты большой таблицы размещены на разных вычислительных системах, то операция полного просмотра может выполняться параллельно (внутриоперационный параллелизм) или могут параллельно выполняться операции выборки отдельных строк (что обеспечивает параллелизм между транзакциями). В первом случае, очевидно, улучшается время отклика, во втором — пропускная способность системы.

Распределение данных может выполняться на физическом уровне (блоки данных) или на логическом (строки таблиц).

При физическом распределении данных может применяться схема чередования (interleave), при использовании которой соседние по номеру блоки размещаются на разных устройствах. Такое размещение позволяет распараллеливать обработку упорядоченных коллекций с сохранением их упорядоченности. Распределение данных на физическом уровне особенно полезно для систем на основе архитектуры SD, потому что в таких системах параллельная работа внешних устройств может управляться любым из вычислителей. Заметим, что планирование размещения на физическом уровне требует значительной работы по анализу характера нагрузки.

Методы физического размещения были хорошо проработаны в тех СУБД, которые содержат средства управления размещением данных на дисках, дублируя функции файловой системы. В СУБД, которые полагаются на файловые системы (в том числе PostgreSQL), реализация подобных средств не имеет смысла. В настоящее время методы такого типа практически вытеснены системами, выполняющими физическое размещение данных автоматически (например, системы RAID).

Значительно чаще используется распределение на логическом уровне, обычно в форме секционирования таблиц. В этом случае таблица представляется как объединение попарно непересекающихся секций (partition). Принадлежность

каждой строки к определенной секции определяется на основе значений некоторого атрибута (или нескольких атрибутов): либо по диапазонам значений, присвоенных каждой секции, либо с помощью функции хеширования, либо на основе списка. Техника разбиения таблиц на секции не связана с параллелизмом непосредственно, однако ее применение в параллельных системах открывает возможности для параллельного выполнения некоторых операций выборки данных.

22.2.4. Параллельные алгоритмы для бинарных операций

В этом разделе кратко описываются параллельные версии основных алгоритмов выполнения операций реляционной алгебры, рассмотренных в главе 11. Все варианты параллельных алгоритмов включают *фазу распределения данных* по вычислительным узлам, выделенным для выполнения операции, и *фазу выработки результата*, которая выполняется параллельно. Конечно, фаза распределения не обязательно включает пересылку данных, т. к. для выполнения операции могут быть выделены процессоры, уже имеющие доступ к аргументам операции.

Как правило, параллельные варианты алгоритмов имеет смысл применять, только если аргументы имеют достаточно большие размеры.

Алгоритм вложенных циклов

На фазе распределения для алгоритма вложенных циклов строится разбиение первого аргумента операции на разделы так, чтобы количество разделов совпадало с количеством процессоров, выделенных для выполнения операции. Если необходима пересылка, разделы рассылаются по процессорам.

На фазе получения результата все кортежи второго аргумента рассылаются на все процессоры, выделенные для выполнения операции, и на каждом из них выполняется алгоритм вложенных циклов, т. е. для каждой строки второго аргумента выполняется просмотр первого аргумента и формируются строки результата.

Этот вариант алгоритма работает эффективно, если разделы первого аргумента, распределенные по вычислителям, полностью помещаются в оперативную память на каждой системе, выделенной для выполнения операции, а второй аргумент имеет большие размеры.

Оценка времени, необходимого для получения результата, очевидно, равна оценке времени, необходимого для выполнения операции на одном процессоре, деленной на число выделенных процессоров (время распределения первого аргумента значительно ниже, чем время выполнения вложенных циклов).

Заметим, что некоторые варианты алгоритма вложенных циклов, эффективные при использовании одного процессора, не всегда хорошо работают в параллельном варианте. Так, алгоритм, использующий индекс, требует обращения всех процессов к одному и тому же индексу, что ограничивает возможности параллельной обработки.

Параллельное соединение на основе хеширования

Параллельный вариант алгоритма на основе хеширования использует функцию хеширования для определения процессора, который будет обрабатывать очередной кортеж. Кроме этого, другие функции хеширования необходимы на каждом процессоре для распределения по корзинам, размещенным на этом процессоре.

На фазе распределения для алгоритма соединения на основе хеширования первый аргумент распределяется по корзинам, которые размещаются на процессорах, выделенных для выполнения операции. Фаза распределения может выполняться параллельно, если первый аргумент к моменту начала операции размещается на нескольких системах (например, вырабатывается предыдущей операцией соединения или извлекается из секционированной таблицы).

По окончании фазы распределения строки второго операнда рассылаются на процессоры в соответствии со значением функции хеширования, т. е. каждая строка посылается только на один процессор. На каждом процессоре выполняется локальный (однопроцессорный) алгоритм соединения. Как и на первой фазе, рассылка может выполняться параллельно, если второй аргумент находится на нескольких вычислителях, которые могут работать параллельно, и, конечно, параллельно происходит вычисление результата на процессорах, выделенных для выполнения операции.

Если распределение значений функции хеширования достаточно равномерно, то алгоритм на основе хеширования масштабируется так же хорошо, как алгоритм вложенных циклов. Именно этот алгоритм (в различных модификациях) наиболее часто применяется для вычисления соединений на параллельных серверах баз данных, поскольку он выигрывает у других алгоритмов на больших объемах данных.

Сортировка и слияние

Алгоритм соединения на основе слияния упорядоченных аргументов можно выполнять параллельно, если оба аргумента упорядочены и разделены на диапазоны. Конечно, диапазоны для обоих аргументов должны быть одни и те же. Данные распределяются между процессорами, выделенными для выполнения операции, и затем слияние выполняется параллельно на всех этих процессорах.

Несколько сложнее обстоит дело с сортировкой. Известно большое количество различных алгоритмов параллельной сортировки, работающих на системах с общей памятью (архитектура SM). Наиболее известным и применяемым чаще других является алгоритм сортировки слиянием. Напомним, что этот алгоритм начинается с построения небольших упорядоченных сегментов, которое выполняется с помощью какого-либо алгоритма сортировки в оперативной памяти. Далее процедура слияния применяется к этим отрезкам для получения все более длинных упорядоченных сегментов, и на последнем шаге все сегменты сливаются в один полностью упорядоченный.

Очевидно, получение начальных упорядоченных сегментов можно выполнять параллельно на нескольких процессорах. Далее каждое слияние двух сегментов выполняется однократным просмотром, при этом результаты слияния могут сразу направляться на вход следующего слияния, если оно выполняется на другом процессоре. Таким образом, при выделении достаточного количества процессоров все слияния сегментов могут выполняться параллельно.

Пусть S обозначает количество начальных упорядоченных сегментов. Тогда для их слияния понадобится $S/2$ процессоров, для слияния результатов — $S/4$ и т. д. Общее число процессоров, необходимых для такой сортировки, составит

$$S + \sum_{i=1}^{\log_2 S} \frac{S}{2^i} \leqslant 2S.$$

При этом оценка времени, необходимого для выполнения сортировки, составит $O(N)$ вместо оценки $O(N \log N)$, справедливой для такой сортировки на одном процессоре. В этих формулах N обозначает размер упорядочиваемой коллекции. Соотношение между S и N определяется длиной начального упорядоченного сегмента $l = N/s$. Получить алгоритм сортировки, который работал бы быстрее, чем за время $O(N)$, невозможно, потому что на последнем шаге слияния выполняется полный просмотр всех упорядочиваемых объектов. Поэтому

применение алгоритма соединения на основе слияния, скорее всего, целесообразно только в том случае, если хотя бы один из аргументов размещен в разделах по диапазонам значений ключа сортировки еще до начала операции.

В системе PostgreSQL параллельные сортировка и слияние не применяются.

22.2.5. Параллелизм между операциями

Возможность параллельного выполнения различных операций одного запроса зависит от того, являются ли операции блокирующими. Наиболее ресурсоемкая операция соединения, если она выполняется на основе хеширования, является блокирующей, т. к. вывод результатов возможен только после завершения фазы распределения. Однако эта операция допускает совмещение по времени фазы проверки второго аргумента (на которой вырабатывается результат) с последующей операцией плана. При этом степень возможного параллелизма зависит от структуры плана выполнения запроса. Для иллюстрации этого рассмотрим вычисление соединения нескольких отношений

$$R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$$

с помощью алгоритма на основе хеширования. Фазу распределения этого алгоритма, примененную к отношению *R,* будем обозначать $d(R)$, а фазу проверки — $t(R)$. Будем считать, что на фазе распределения всегда обрабатывается левый аргумент. Хорошо известно, что в алгоритме соединения на основе хеширования распределять следует аргумент меньшего размера, однако можно считать, что перестановка аргументов, если необходимо, уже выполнена оптимизатором.

Если операции в этом запросе выполняются слева направо:

$$((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4,$$

то каждая фаза распределения, начиная со второй, может выполняться параллельно с вычислением результата предыдущей операции, при этом в каждый момент времени выполняется не более двух операций (хотя, конечно, каждая из них может использовать несколько процессоров):

$$\begin{array}{c|c|c|c} d(R_1) & \begin{array}{c} t(R_2) \\ d(R_1 \bowtie R_2) \end{array} & \begin{array}{c} \\ t(R_3) \\ d(R_1 \bowtie R_2 \bowtie R_3) \end{array} & \begin{array}{c} \\ \\ t(R_4). \end{array} \end{array}$$

На этой диаграмме предполагается, что время увеличивается слева направо, а операции, размещенные в разных строках одна под другой, могут выполняться параллельно.

Если вычисление выполняется справа налево:

$$R_1 \bowtie (R_2 \bowtie (R_3 \bowtie R_4)),$$

то параллельно могут выполняться все операции:

$$\begin{array}{c|c} d(R_3) & t(R_4) \\ d(R_2) & t(R_3 \bowtie R_4) \\ d(R_1) & t(R_2 \bowtie R_3 \bowtie R_4). \end{array}$$

Конечно, выбор между этими стратегиями зависит от оценок стоимости всех операций, от количества доступных процессоров и от общей загруженности сервера. Поэтому сложность задачи оптимизации запросов для параллельного сервера баз данных значительно выше, чем для централизованного.

22.2.6. Не все так просто

Оценки, приведенные выше, показывают, что потенциально параллельные серверы баз данных могут очень хорошо масштабироваться: время выполнения наиболее сложных операций может сокращаться пропорционально количеству выделенных ресурсов. Очевидно, что такие показатели невозможно улучшить.

Эти оценки, однако, справедливы, только если нагрузка на параллельные процессоры распределяется равномерно. Добиться этого на реальных данных практически невозможно. Дело в особенности осложняется тем, что даже при равномерном распределении аргументов результат операции соединения может быть распределен крайне неравномерно. Это явление называется *скосом* (data skew). Известны более сложные варианты алгоритма соединения на основе хеширования, которые обнаруживают скос данных во время выполнения и могут динамически перераспределить нагрузку. Однако даже такие алгоритмы не решают эту проблему полностью.

Выше уже упоминалась очень высокая сложность оптимизации для параллельных систем по сравнению с централизованными. Кроме обычных оценок, оптимизатор должен учитывать стоимость операций пересылки данных между

вычислительными системами, если такая пересылка необходима. При этом необходимость пересылки зависит от того, какие процессоры назначены для выполнения других операций плана.

Планирование размещения данных также представляет собой весьма сложную задачу.

Все это привело к тому, что проработанные и опробованные во многих СУБД к середине 90-х гг. средства реализации параллельных серверов баз данных оказались довольно сложными в управлении и поэтому применяются относительно редко.

Упомянутые сложности, ограничения практических реализаций параллелизма и непомерно высокая стоимость параллельных вариантов коммерческих СУБД привели к широкому распространению мифа о том, что реляционные системы не могут масштабироваться при увеличении объемов данных и нагрузки. В результате появились альтернативные подходы, в которых задачи распределения данных между вычислительными системами и распределение нагрузки переносятся на уровень приложения. При этом зачастую вводится альтернативная терминология, например вместо секционирования (partitioning) используется термин «шардирование» (sharding), хотя, по существу, эти понятия практически не различаются. Одновременно приходится отказываться от применения распределенных декларативных запросов, поддержки транзакций и многих других функций СУБД, что в долгосрочном плане ухудшает характеристики прикладных систем.

22.2.7. Параллельные запросы в PostgreSQL

При организации параллельного выполнения PostgreSQL опирается на средства операционной системы и предполагает наличие общей памяти (архитектура SM). Реализация в настоящее время имеет некоторые ограничения, однако эта часть системы быстро развивается, и поэтому весьма вероятно, что в ближайших версиях эти ограничения будут сняты или ослаблены.

Оптимизатор может строить планы для параллельного выполнения несколькими процессами. Параллельно могут выполняться следующие операции:

- полный просмотр таблицы (с проверкой условий фильтрации);
- просмотр индексов по диапазону значений;
- просмотр битовых карт;

- соединение алгоритмами вложенных циклов, на основе хеширования и на основе слияния;
- агрегирование.

Параллельные операции могут применяться для любых таблиц, т. е. не требуется предварительное разбиение таблицы на секции. Распределение данных между процессами производится на уровне блоков.

Реализация алгоритма соединения на основе хеширования на фазе распределения выполняет параллельный просмотр первого аргумента и строит общую для всех процессов таблицу хеширования. На фазе проверки параллельно обрабатывается второй аргумент.

Алгоритм соединения на основе слияния использует разбиение первого аргумента, но в текущей реализации каждый процессор полностью читает второй аргумент.

Для генерации и выполнения параллельных планов не требуется изменений в запросах. Оптимизатор выбирает параллельные планы автоматически, если их стоимость оказывается ниже стоимости эквивалентных последовательных планов. Параллельное выполнение управляется несколькими параметрами конфигурации сервера баз данных, ограничивающими максимальное количество процессов, которые могут быть задействованы для выполнения всех параллельных запросов (max_parallel_workers) и для выполнении одного запроса (max_parallel_workers_per_gather).

Имеется ряд ограничений на запросы, нарушение которых препятствует включению параллельных операций в планы выполнения этих запросов. Эти ограничения необходимы, для того чтобы гарантировать корректность выполнения запросов в тех случаях, когда параллельное выполнение потенциально могло бы привести к непредсказуемым результатам. В первую очередь это касается вызовов функций, написанных пользователем. Система не может проверить безопасность параллельного выполнения таких функций и полагается на указания, заданные при создании функции:

PARALLEL SAFE означает, что функция может использоваться в параллельных запросах;

RESTRICTED допускает выполнение функции в последовательной части параллельных планов;

UNSAFE запрещает использование параллельных планов.

Не могут выполняться параллельно любые запросы, содержащие операции обновления базы данных. Вплоть до версии 12 параллельное выполнение запросов было невозможно на уровне изоляции Serializable.

22.3. Выполнение запросов в распределенных СУБД

22.3.1. Конфигурации распределенных баз данных

Распределенные системы баз данных, рассматриваемые в этом разделе, обеспечивают возможность обработки данных, находящихся на разных серверах, в одном запросе. Такие системы могут быть однородными (состоять из серверов, на которых выполняется одна и та же СУБД) или неоднородными (включать СУБД различных версий или различных производителей). Также возможен доступ к данным, которые предоставляются некоторыми сервисами, не являющимися серверами баз данных.

В этом разделе мы рассматриваем только выполнение запросов в распределенной системе баз данных, но не затрагиваем вопросы, связанные с поддержкой согласованности в таких системах. Эти вопросы обсуждаются в разделе 22.4.

Любой из серверов баз данных, входящих в состав распределенной системы, может принимать и выполнять запросы. В этом смысле в распределенной системе отсутствует какой-либо центр, координирующий работу системы, или компоненты, централизующие какие-либо функции, что считается важным для распределенных систем. При отказе отдельных серверов, входящих в систему, или сетевых соединений между ними работа остальных серверов может продолжаться, хотя, возможно, часть данных с некоторых серверов станет временно недоступной.

Конечно, требование отсутствия единого центра или единой компоненты, отказ которой приводит к прекращению работы всей системы, выполняется далеко не всегда и часто с некоторыми ограничениями. Например, схема репликации с главным сервером, по существу, не удовлетворяет этому требованию, поскольку обновления могут выполняться только на одном из серверов.

Во многих реализациях распределенных систем могут не выполняться или выполняться не в полной мере и другие требования, обычно предъявляемые к СУБД. Например, во многих случаях не реализуются требования ACID для распределенных транзакций.

Одним из аргументов в пользу создания ранних распределенных систем была возможность размещения данных там, где они чаще всего используются. Например, данные предприятия, подразделения которого размещены в географически удаленных друг от друга местах, могут быть разделены между базами данных, размещенными в этих подразделениях, даже если логически эти данные представляют собой одну таблицу. В настоящее время этот аргумент уже не всегда актуален, однако даже при наличии широкополосных сетевых соединений время отклика при большом количестве промежуточных узлов может значительно превышать время отклика для таких же запросов, выполняемых на близко расположенных серверах.

Распределенное размещение логически взаимосвязанных данных называется *фрагментацией*. Принято различать *горизонтальную фрагментацию*, при которой секции таблицы размещаются на различных серверах, и *вертикальную фрагментацию*, при которой на разные серверы разносятся группы колонок. Однако в этой главе для нас не будет иметь значения, используется фрагментация или нет. Мы также не будем специально обсуждать возможности репликации отдельных секций или вертикальных фрагментов.

Далее мы будем рассматривать конфигурацию и работу распределенной системы баз данных с точки зрения одного клиента, который создал сеанс работы, установив соединение с одним из серверов такой системы. По отношению к сеансу клиента этот сервер будет далее называться локальным, а все остальные серверы — удаленными.

22.3.2. Организация доступа к удаленным данным

Для того чтобы данные, размещенные на удаленном сервере, стали доступными, необходимо определить их в локальной базе данных. Существуют механизмы доступа к удаленным базам данных и другим источникам данных, которые не создают и не используют какие-либо описания данных в локальной базе данных, однако в этом случае факт распределенности не может быть прозрачным для приложения, выдающего запрос.

Для этого могут применяться разные механизмы. Мы рассмотрим механизм *оберток сторонних данных* (foreign data wrapper), который дает возможность описывать источники данных, такие как таблицы или наборы таблиц, являющиеся внешними по отношению к локальной схеме. Для того чтобы использовать этот механизм в системе PostgreSQL, требуется выполнить несколько действий.

Прежде всего необходимо, чтобы программные модули, обеспечивающие доступ к нужному типу источника данных, были реализованы в системе. Некоторые модули такого назначения входят в состав основной версии PostgreSQL как расширения. Например, для того чтобы получить доступ к внешним базам данных, находящимся на серверах PostgreSQL, необходимо включить расширение postgres_fdw в локальной базе данных:

```
local=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

В состав основной версии входит также обертка file_fdw для доступа к файлам нескольких форматов (в том числе текстовым и CSV), размещенным в файловой системе локального сервера баз данных. Количество оберток для других типов источников данных исчисляется десятками. Прежде чем писать свою реализацию, следует проверить, не сделал ли это кто-нибудь раньше. Обертки для дополнительных внешних источников создаются оператором CREATE FOREIGN DATA WRAPPER, но мы не будем обсуждать разработку новых оберток внешних данных.

После того как создана обертка, необходимо определить, на каком сервере находятся внешние данные, с помощью оператора CREATE SERVER. В этом операторе указываются имя создаваемого сервера и, если необходимо, дополнительные параметры, определяющие, например, сетевой адрес системы, на которой находится сервер, имя базы данных на этом сервере и т. п. Какие именно параметры требуются, зависит от типа обертки, который тоже указывается в этом операторе. Указанное имя сервера используется только в локальной базе данных и может никак не быть связано с именами, используемыми на источнике данных.

В следующем примере создается сервер, который дает доступ к демонстрационной базе данных:

```
local=# CREATE SERVER foreign_demo
    FOREIGN DATA WRAPPER postgres_fdw
    OPTIONS (host 'localhost', port '5432', dbname 'demo');
CREATE SERVER
```

В качестве сетевого имени указан адрес локального компьютера. Это, конечно, не имеет большого смысла, однако все средства создания распределенных систем работают точно так же, как если бы эта база данных размещалась на другом компьютере.

Если удаленный сервер поддерживает разграничение доступа (как, например, PostgreSQL), то необходимо для каждой локальной роли (или пользователя), от имени которой будет выполняться обработка данных на удаленном сервере, определить, какая удаленная роль (или пользователь) будут использоваться для этой цели. Такое соответствие устанавливается с помощью оператора CREATE USER MAPPING, в котором указывается имя локальной роли, удаленный сервер (определенный ранее оператором CREATE SERVER) и параметры, определяющие роль на удаленном сервере.

Если на локальном сервере используется роль local_user, а к демонстрационной базе данных требуется подключаться под именем demo, то команда может иметь следующий вид:

```
local=# CREATE USER MAPPING FOR local_user
    SERVER foreign_demo
    OPTIONS (user 'demo');
CREATE USER MAPPING
```

Конечно, определять отображение ролей не нужно, если удаленный источник не поддерживает разграничение доступа. Например, это не требуется для доступа к файлам на локальном сервере через обертку file_fdw.

После того как определены внешний сервер и отображения ролей, можно определять внешние таблицы с помощью оператора CREATE FOREIGN TABLE. В этом операторе указывается список колонок с их типами точно так же, как в обычном операторе CREATE TABLE. При этом структура таблицы (порядок и типы колонок) должны в точности соответствовать тем, которые имеются на внешнем сервере. Кроме списка колонок необходимо указать сервер, на котором находятся данные. Мы не будем создавать внешние таблицы для нашего примера, так как обертка postgres_fdw позволяет использовать импорт схемы.

Если обертка предоставляет такую возможность, вместо явного определения внешних таблиц можно импортировать их из удаленной схемы с помощью оператора IMPORT FOREIGN SCHEMA. Этот оператор создает в указанной локальной схеме корректные определения внешних таблиц, соответствующие всем (или указанным) таблицам и представлениям, имеющимся в заданной схеме на удаленном сервере. В нашем случае нужно создать схему и выполнить оператор импорта:

```
local=# CREATE SCHEMA fn_demo;
CREATE SCHEMA
```

```
local=# IMPORT FOREIGN SCHEMA bookings
    FROM SERVER foreign_demo
    INTO fn_demo;
IMPORT FOREIGN SCHEMA
```

Теперь можно получить список внешних таблиц в только что созданной схеме:

```
local=# \det fn_demo.*
          List of foreign tables
 Schema  |      Table      |    Server
---------+-----------------+--------------
 fn_demo | aircrafts       | foreign_demo
 fn_demo | aircrafts_data  | foreign_demo
 fn_demo | airports        | foreign_demo
 fn_demo | airports_data   | foreign_demo
 fn_demo | boarding_passes | foreign_demo
 fn_demo | bookings        | foreign_demo
 fn_demo | flights         | foreign_demo
 fn_demo | flights_v       | foreign_demo
 fn_demo | routes          | foreign_demo
 fn_demo | seats           | foreign_demo
 fn_demo | ticket_flights  | foreign_demo
 fn_demo | tickets         | foreign_demo
(12 rows)
```

Кроме оберток сторонних данных известны и другие способы организации доступа к данным, хранящимся за пределами одной базы данных или кластера баз данных PostgreSQL. Механизм dblink позволяет выполнять запросы на удаленном сервере, однако эти запросы не могут быть частью распределенного запроса.

22.3.3. Подготовка и выполнение запросов

После создания внешних таблиц в словаре данных (в информационной схеме PostgreSQL) находятся описания структуры этих таблиц. Синтаксически внешние таблицы не отличаются в операторах SQL от обычных, поэтому компиляция и другие этапы подготовки запросов для выполнения не отличаются от соответствующих этапов для локальных запросов.

Важная отличительная особенность внешних таблиц состоит в том, что возможность выполнения, например, операций обновления зависит от того, позволяет ли такие операции обертка.

Выполним выборку данных из одной внешней таблицы:

```
local=# SELECT *
FROM fn_demo.aircrafts;
 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Боинг 777-300        | 11100
 763           | Боинг 767-300        |  7900
 SU9           | Сухой Суперджет-100  |  3000
 320           | Аэробус A320-200     |  5700
 321           | Аэробус A321-200     |  5600
 319           | Аэробус A319-100     |  6700
 733           | Боинг 737-300        |  4200
 CN1           | Сессна 208 Караван   |  1200
 CR2           | Бомбардье CRJ-200    |  2700
(9 rows)
```

Для того чтобы оптимизатор мог строить эффективные планы, необходима статистическая информация, которую можно, как обычно, получить командой ANALYZE, если, конечно, обертка это поддерживает:

```
demo=# ANALYZE fn_demo.aircrafts;
ANALYZE
```

Важно подчеркнуть, что внешние таблицы всегда представляют собой только описания, а данные всегда находятся на удаленном сервере. В этом можно убедиться, получив план выполнения запроса:

```
demo=# EXPLAIN (verbose)
SELECT aircraft_code
FROM fn_demo.aircrafts;
                          QUERY PLAN
---------------------------------------------------------------
 Foreign Scan on fn_demo.aircrafts
   (cost=100.00..100.27 rows=9 width=4)
   Output: aircraft_code
   Remote SQL: SELECT aircraft_code FROM bookings.aircrafts
```

Появление в запросе внешних таблиц влияет на работу оптимизатора. Операции выборки данных из внешних таблиц отличаются от обычных операций, возможность использования индексов для локального оптимизатора недоступна. Некоторые обертки, в частности postgres_fdw, позволяют передавать условия фильтрации в обертку. В этом случае локальный оптимизатор может передать условия фильтрации, а оптимизатор удаленного сервера сможет построить план, использующий подходящие индексы. Точно так же, если источник данных описан запросом, этот запрос оптимизируется и выполняется на удаленном сервере.

В тех случаях, когда обертка это поддерживает, оптимизатор может направить на удаленный сервер операторы соединения таблиц, находящихся на одном и том же сервере, а если это невозможно, то данные должны пересылаться на локальный сервер. Развитые обертки, в том числе postgres_fdw, позволяют получать статистику для внешних таблиц и другую информацию, необходимую для работы оптимизатора, однако такие возможности реализованы далеко не для всех оберток.

22.4. Согласованность в распределенных системах

22.4.1. Распределенные транзакции

Для того чтобы обсуждать обработку транзакций в распределенных системах, напомним, что распределенная система баз данных состоит из нескольких серверов, каждый из которых может принимать запросы, в которых используются данные, размещенные на разных серверах распределенной системы. При этом некоторые элементы данных могут храниться в нескольких копиях на разных серверах, и в то же время некоторые объекты данных могут быть фрагментированы, и разные фрагменты могут храниться на разных серверах. С точки зрения поддержки согласованности фрагментация не имеет значения, а задача согласования копий оказывается довольно сложной.

Транзакции, которые могут содержать операции обработки элементов данных, размещенных на разных серверах, называются *глобальными* или *распределенными*, а транзакции, все операции которых обрабатывают данные, находящиеся на одном сервере, — *локальными*. Каждая глобальная транзакция может рассматриваться как совокупность своих локальных проекций, т. е. подмножеств операций глобальной транзакции, выполняемых на одном сервере. При этом мы рассматриваем упрощенную модель, в которой не предусмотрена пересылка данных с одного сервера на другой. Такие операции пересылки могут быть нужны, для того чтобы оптимизатор запросов мог обрабатывать распределенные запросы так же, как локальные, но в страничной модели транзакций не рассматриваются сложные запросы.

В рамках модели, предусматривающей только операции *r* и *w*, пересылка представляется как чтение данных на одном сервере и запись на другом. Это дает возможность описывать транзакции в нашей модели, однако накладывает дополнительные ограничения на синхронизацию локальных проекций.

В обертках сторонних данных системы PostgreSQL (таких как postgres_fdw) пока не реализована поддержка глобальных транзакций, поэтому материал этого раздела применим к другими СУБД или к некоторым из надстроек над PostgreSQL, но не к основной открытой версии этой системы.

В связи с выполнением транзакций в распределенных системах необходимо рассматривать две задачи:

1) на каждом сервере должны генерироваться такие расписания, чтобы глобальные транзакции удовлетворяли требуемому уровню изоляции (например, можно требовать, чтобы расписание было сериализуемым);
2) все локальные подтранзакции одной глобальной транзакции должны завершаться одинаково (т. е. либо все фиксироваться, либо все обрываться).

22.4.2. Протоколы управления транзакциями

Начнем с обсуждения однородной распределенной системы, в которой все серверы работают под управлением одной СУБД. В этом случае все серверы могут применять одинаковые протоколы управления транзакциями и потенциально есть возможность получить глобальную согласованность.

Следующий вариант расписания 13.2

$$r_2(x)\ r_1(x)\ w_1(x)\ w_1(z)\ r_3(y)\ w_3(y)\ w_3(z)\ r_2(y)$$

доказывает, что глобальная сериализация не может быть достигнута с помощью локальной сериализации проекций на каждом сервере. Действительно, если элементы данных x и y размещены на одном сервере, а z находится на другом, то локальные проекции транзакций 1 и 3 могут быть сериализованы на этих серверах в разном порядке.

Следовательно, для достижения глобальной согласованности необходимы глобальные протоколы управления транзакциями. Важнейшими требованиями к распределенным системам и, следовательно, к диспетчерам транзакций являются высокая доступность, отсутствие централизованных механизмов управления и горизонтальная масштабируемость.

Для того чтобы использовать глобальные блокировки, необходимо либо централизовать управление блокировками на одном из серверов, либо раскопировать информацию об установленных блокировках на все серверы распределенной системы (в том числе не участвующие в выполнении данной транзакции).

В том и другом случае система не будет масштабируемой. Кроме того, в первом случае появится централизованный механизм управления, что потенциально снизит доступность, а во втором количество сообщений, необходимых для установки и снятия блокировок, окажется недопустимо большим.

Поэтому применение протоколов, основанных на использовании блокировок, в распределенных системах практически невозможно. Это привело к тому, что многие разработчики приложений отказываются от согласованности или используют ослабленные критерии.

В отличие от протоколов на основе блокировок протокол на основе меток времени (рассмотренный в разделе 13.2.6) допускает распределенную реализацию с локальной проверкой согласованности. В такой реализации метки времени для каждой транзакции назначаются локально тем сервером, который начинает выполнение транзакции. Возможность выполнения операций также проверяется локальным сравнением меток времени. В результате задача поддержки согласованности сводится к задаче синхронизации часов в распределенной системе, при этом неточная синхронизация приводит к увеличению доли оборванных транзакций, но не к нарушениям согласованности.

Чтобы гарантировать уникальность меток времени, присваиваемых транзакциям, можно использовать механизм часов Лэмпорта: локально уникальная метка времени конкатенируется с порядковым номером сервера в системе.

В сочетании с поддержкой множественных версий элементов данных распределенный протокол SI, реализованный на основе меток времени, позволяет получить очень высокую пропускную способность: на экспериментальных реализациях были получены значения пропускной способности, на несколько порядков превышающие показатели систем, применяющих блокировки [48]. В этой работе метки времени назначались централизованно.

В работе [58] представлены результаты экспериментов, в которых была достигнута пропускная способность в несколько миллионов транзакций в секунду, также с применением протокола SI.

При этом системы на основе блокировок обычно работают в режиме изоляции Read Committed, а в этих экспериментах обеспечивалась согласованность, обычная для протокола SI. Необходимо, однако, отметить, что масштабируемость в этих экспериментах достигалась по количеству серверов, но не по размеру транзакций, которые во всех случаях были ограниченными.

22.4.3. Завершение распределенных транзакций

Рассмотрим теперь задачу завершения распределенных транзакций. Для того чтобы обеспечить одинаковое завершение всех локальных подтранзакций, с середины 80-х гг. стандартизован и применяется двухфазный протокол фиксации (two-phase commit, 2PC). Внимание: этот протокол никак не связан с протоколом 2PL.

В соответствии с этим протоколом среди серверов, участвующих в выполнении транзакции, выделяется один, который называется *координатором* (для этой транзакции). Все остальные серверы называются *участниками*.

После завершения всех операций распределенной транзакции координатор начинает первую фазу, рассылая всем участникам сообщение prepare.

Получив такое сообщение, сервер-участник может ответить координатору одним из двух способов:

1. Подтвердить готовность к фиксации этой транзакции (ready to commit).

   В этом случае участник обязан принять все меры, для того чтобы обеспечить фиксацию, т. е. записать в журнал информацию обо всех обновлениях, выполненных транзакцией, и сделать запись о том, что было отправлено сообщение ready to commit.

   Другими словами, принимаются такие же меры, чтобы транзакция не могла быть потеряна, как и при фиксации, но при этом сохраняется возможность обрыва.

2. Сообщить о невозможности фиксации и, следовательно, о необходимости обрыва транзакции.

   Решение об обрыве транзакции любой участник может принять в одностороннем порядке. Он выполняет действия, которые обычно выполняются при обрыве, и сообщает об этом координатору, не ставя в известность других участников.

   Если сообщение о невозможности фиксации не будет доставлено координатору, транзакция все равно будет оборвана, но это займет больше времени, потому что координатор будет ждать истечения контрольного срока (time out).

Координатор ожидает ответ на свое сообщение prepare и после получения ответов от всех участников или по истечении предельного времени ожидания переходит ко второй фазе.

- Если все участники подтвердили готовность к фиксации, координатор рассылает сообщение commit, получив которое, участник завершает фиксацию транзакции.
- Если хотя бы один из серверов сообщил о невозможности фиксации или не ответил на первой фазе, координатор принимает решение об обрыве транзакции и рассылает соответствующее сообщение всем участникам.

Если на одном из серверов-участников во время двухфазной фиксации произошел отказ системы, то после рестарта транзакция может оказаться в так называемом сомнительном состоянии. В этом случае участник пытается узнать, как завершилась эта транзакция, посылая сообщение координатору, а если он не отвечает, то другим участникам. Однако существуют ситуации, в которых выяснение статуса транзакции невозможно, и поэтому требуется вмешательство администратора данных.

Двухфазный протокол завершения закреплен в международных стандартах. Он поддерживается во многих СУБД и поэтому может применяться в неоднородных распределенных системах, в которых разные серверы работают под управлением различных систем управления базами данных.

Протокол 2PC предусматривает синхронную передачу значительного количества сообщений с получением ответов, и поэтому широко распространилось мнение о том, что этот протокол является недопустимо медленным. Считается, что его применение существенно ухудшает время отклика и косвенно влияет на пропускную способность, т. к. блокировки, установленные для транзакций, находящихся в фазе готовности к фиксации, но еще не зафиксированных, должны сохраняться. В связи с этим протокол 2PC относительно редко используется разработчиками прикладных систем.

В действительности, однако, задержки вызываются не столько количеством сообщений, которые передаются (почти) одновременно, а ситуациями, в которых участник не отвечает, что приводит к необходимости завершения ожидания по времени (time out). Поскольку современные серверы достаточно надежны, вероятность подобных ситуаций достаточно низка. В случае если данные реплицированы, нет необходимости в получении подтверждения от всех участников, а достаточно получить подтверждение от большинства, как в мажоритарных

протоколах репликации, что практически исключает необходимость ожидания по времени.

Проблема с блокировками снимается применением протоколов на основе меток времени, в том числе SI. В работе [12] показано, как можно получить очень высокую пропускную способность при некоторых ограничениях на распределенные транзакции, позволяющих координировать фиксацию параллельно с выполнением транзакции. В статье [58] показано, как можно добиться масштабируемости транзакционных систем за счет применения нового оборудования. Сравнительный анализ распределенных систем обработки транзакций можно найти в [5].

22.5. Итоги главы

Методы и технологии реализации и применения параллельных и распределенных систем управления базами данных достаточно хорошо проработаны несколько десятилетий назад. Операции, реализующие высокоуровневые языки запросов, хорошо распараллеливаются и могут применяться в высокопроизводительных системах.

Однако такие системы оказываются весьма сложными для конфигурации и эксплуатации и поэтому при разработке прикладных систем зачастую используются менее эффективные альтернативные архитектуры.

22.6. Упражнения

Упражнение 22.1. Настройте параметры конфигурации сервера PostgreSQL так, чтобы могли генерироваться и выполняться параллельные планы. Выполните запросы, в которых требуется полный просмотр таблиц демонстрационной базы данных большого размера с использованием параллелизма и без него. Сравните планы и фактическое время выполнения запросов.

Упражнение 22.2. Создайте сервер и внешнюю таблицу на основе файла с помощью обертки внешних данных file_fdw. Сравните чтение из этой таблицы с применением оператора SQL COPY.

Упражнение 22.3. Определите два внешних сервера, ссылающихся на одну и ту же удаленную базу данных. Выполните запрос, содержащий операцию соединения двух таблиц, указывая источники этих таблиц в соответствии со следующими вариантами:

1) обе таблицы в локальной базе данных;
2) одна из таблиц в локальной базе данных, другая в удаленной;
3) обе таблицы в одной удаленной базе данных;
4) таблицы в разных удаленных базах данных.

Используйте обертку postgres_fdw.

Заключение

Дойдя до конца этой книги, внимательный читатель познакомился с основными идеями и принципами построения систем управления базами данных и получил представление о том, как эти идеи развивались и изменялись на протяжении десятилетий. Мы показали, как идеи и принципы влияют на модели и языки, а модели и языки реализуются в промышленных СУБД.

Не переполняя текст деталями, мы рассказали, как организовывать взаимодействие различных моделей и языков, необходимых для реализации прикладных систем, для того чтобы эти системы получались высококачественными, надежными и эффективными.

Во второй части книги мы подробно рассмотрели методы и алгоритмы, применяемые для реализации систем управления базами данных.

Мы показали, как рекомендации теории реализуются в системе управления базами данных PostgreSQL. За три десятилетия своего развития эта система из небольшого экспериментального прототипа объектно-ориентированной СУБД превратилась в мощную систему, которая поддерживает разнообразные модели данных и методы разработки приложений, включает высокоэффективные средства выполнения запросов и развитые средства управления конкурентным доступом. Современные версии PostgreSQL предоставляют все необходимое для создания высокопроизводительных и высоконадежных информационных систем.

Благодаря многообразным средствам расширения система PostgreSQL открыта для добавления новой функциональности, а возможность добавления новых источников данных, не обязательно хранящихся под управлением PostgreSQL, позволяет создавать неоднородные распределенные системы.

В настоящее время система PostgreSQL является наиболее мощной среди систем с открытым кодом и успешно конкурирует с коммерческими системами.

Технологии баз данных постоянно развиваются. Это связано как с появлением новых областей применения и новых средств разработки приложений, так и с расширением возможностей вычислительных систем, на которых работают серверы баз данных. Среди многочисленных новых направлений развития можно выделить следующие:

Автономные серверы баз данных. Сейчас для достижения оптимальной производительности, особенно при большой нагрузке, требуется настройка многих параметров сервера баз данных. Автономные серверы смогут автоматически настраивать необходимые параметры с учетом фактической нагрузки. Это в особенности важно для систем, работающих в облачной среде. При создании автономного сервера не обойтись без технологий машинного обучения.

Улучшение моделей стоимости для оптимизации запросов. Как правило, оптимизаторы хорошо справляются со своей работой и вырабатывают планы высокого качества. В некоторых случаях, однако, планы получаются далекими от оптимальных, и наиболее частой причиной этого оказываются неточные оценки стоимости. Ожидается, что применение методов машинного обучения позволит получать существенно более точные оценки и, следовательно, повысить качество оптимизации.

Автоматическая настройка индексов. Используемые в настоящее время индексные структуры обладают высокими эксплуатационными качествами, и, казалось бы, в этой области существенные улучшения уже невозможны. Однако производительность индексов можно улучшить, применяя адаптивные методы, учитывающие особенности распределения индексируемых значений.

Применение возможностей современных и будущих процессоров. Алгоритмы параллельного выполнения запросов, применяемые в настоящее время, не могут в полной мере использовать большое количество (десятки и сотни) ядер, которые будут доступны в обозримом будущем.

Энергонезависимая память с произвольным доступом (NVRAM). Этот вид памяти может применяться для постоянного хранения данных уже сейчас, но чтобы использовать все возможности такого оборудования, необходимо пересмотреть все структуры хранения данных, применяемые в настоящее время в промышленных системах.

Распределенные системы на основе широкополосных сетей. Вычислительные сети с высокой пропускной способностью уже применяются в экспериментальных системах, однако, для того чтобы возможности таких сетей были использованы в полной мере, требуется пересмотр методов и средств взаимодействия между серверами баз данных, входящими в распределенную систему.

Широко применяемые в настоящее время методы взаимодействия приложений с базами данных обладают рядом недостатков, что зачастую приводит к неэффективному использованию СУБД. Новые модели доступа к базам данных могли бы упростить разработку приложений и позволили бы более интенсивно применять все возможности баз данных.

Однако и сейчас современные высокопроизводительные системы, в том числе PostgreSQL, предоставляют приложениям все необходимые средства для эффективной работы. Но правильное применение этих средств и возможностей требует от разработчиков приложений определенных знаний и усилий.

Прикладные системы должны работать, не доставляя неудобств нашим драгоценным конечным пользователям. Авторы надеются, что эта книга поможет читателю сделать так, чтобы в сообщениях «Пожалуйста, подождите...» не было необходимости.

Список литературы

1. A Critique of ANSI SQL Isolation Levels / H. Berenson [et al.] // Proceedings of the 1995 ACM International Conference on Management of Data. — ACM, 1995. — Pp. 1–10. — (SIGMOD '95). — ISBN 0-89791-731-6. — DOI: 10.1145/223784.223785.
2. A Survey of Qualitative Spatial and Temporal Calculi: Algebraic and Computational Properties / F. Dylla [et al.] // ACM Comput. Surv. — 2017. — Vol. 50, no. 1. — Pp. 7:1–7:39. — ISSN 0360-0300. — DOI: 10.1145/3038927.
3. *Agrawal R.*, *Dar S.*, *Jagadish H. V.* Direct Transitive Closure Algorithms: Design and Performance Evaluation // ACM Trans. Database Syst. — 1990. — Vol. 15, no. 3. — Pp. 427–458. — ISSN 0362-5915. — DOI: 10.1145/88636.88888.
4. *Allen J. F.* Maintaining Knowledge About Temporal Intervals // Commun. ACM. — 1983. — Vol. 26, no. 11. — Pp. 832–843. — ISSN 0001-0782. — DOI: 10.1145/182.358434.
5. An Evaluation of Distributed Concurrency Control / R. Harding [et al.] // Proc. VLDB Endow. — 2017. — Vol. 10, no. 5. — Pp. 553–564. — ISSN 2150-8097. — DOI: 10.14778/3055540.3055548.
6. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging / C. Mohan [et al.] // ACM Trans. Database Syst. — 1992. — Vol. 17, no. 1. — Pp. 94–162. — ISSN 0362-5915. — DOI: 10.1145/128765.128770.
7. *Bachman C. W.* The Programmer As Navigator // Commun. ACM. — 1973. — Vol. 16, no. 11. — Pp. 653–658. — ISSN 0001-0782. — DOI: 10.1145/355611.362534.
8. *Bayer R.*, *McCreight E. M.* Organization and Maintenance of Large Ordered Indices // Acta Inf. — 1972. — Vol. 1. — Pp. 173–189. — DOI: 10.1007/BF00288683.
9. *Bernstein P. A.* Synthesizing Third Normal Form Relations from Functional Dependencies // ACM Trans. Database Syst. — 1976. — Vol. 1, no. 4. — Pp. 277–298. — ISSN 0362-5915. — DOI: 10.1145/320493.320489.
10. *Bernstein P. A.*, *Hadzilacos V.*, *Goodman N.* Concurrency Control and Recovery in Database Systems. — Addison-Wesley, 1987. — ISBN 0-201-10715-5.
11. *Cao W.*, *Shasha D.* AppSleuth: a tool for database tuning at the application level // Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings. — 2013. — Pp. 589–600. — DOI: 10.1145/2452376.2452445.

12. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data / X. Yan [et al.] // Proceedings of the 2018 International Conference on Management of Data. — ACM, 2018. — Pp. 231–243. — (SIGMOD '18). — ISBN 978-1-4503-4703-7. — DOI: 10.1145/3183713.3196912.

13. *Cattell R.*, *Barry D.*, *Berler M.* The Object Data Standard: ODMG 3.0. — Morgan Kaufmann, 2000. — ISBN 978-1-558-60647-0.

14. *Celko J.* Joe Celko's SQL for Smarties: Advanced SQL Programming. — 5th ed. — Morgan Kaufmann, 2014. — P. 852. — ISBN 978-0-128-00761-7.

15. *Celko J.* Joe Celko's Thinking in Sets: Auxiliary, Temporal, and Virtual Tables in SQL. — Morgan Kaufmann, 2008. — P. 384. — ISBN 978-0-123-74137-0.

16. *Ceri S.*, *Cochrane R.*, *Widom J.* Practical Applications of Triggers and Constraints: Success and Lingering Issues (10-Year Award) // Proceedings of the 26th International Conference on Very Large Data Bases. — Morgan Kaufmann, 2000. — Pp. 254–262. — (VLDB '00). — ISBN 978-1-558-60715-6.

17. *Chen P. P.-S.* The Entity-relationship Model: Toward a Unified View of Data // SIGIR Forum. — 1975. — Vol. 10, no. 3. — Pp. 9–9. — ISSN 0163-5840. — DOI: 10.1145/1095277.1095279.

18. *Codd E. F.* A Relational Model of Data for Large Shared Data Banks // Commun. ACM. — 1970. — Vol. 13, no. 6. — Pp. 377–387. — ISSN 0001-0782. — DOI: 10.1145/362384.362685.

19. *Codd E. F.* Extending the Database Relational Model to Capture More Meaning // ACM Trans. Database Syst. — 1979. — Vol. 4, no. 4. — Pp. 397–434. — ISSN 0362-5915. — DOI: 10.1145/320107.320109.

20. *Curtice R. M.* Data Base Design Using IMS/360 // Proceedings of the December 5–7, 1972, Fall Joint Computer Conference, Part II. — ACM, 1972. — Pp. 1105–1110. — (AFIPS '72). — DOI: 10.1145/1480083.1480146.

21. Data Base Task Group Report to the CODASYL Programming Language Committee, October 1969 / C. W. Bachman [и др.]. — New York, NY, USA: ACM, 1969.

22. *Deshpande A.*, *Ives Z. G.*, *Raman V.* Adaptive Query Processing // Foundations and Trends in Databases. — 2007. — Vol. 1, no. 1. — Pp. 1–140. — DOI: 10.1561/1900000001.

23. *Deshpande A.*, *Ives Z.*, *Raman V.* Adaptive Query Processing: Why, How, When, What Next? // Proceedings of the 33rd International Conference on Very Large Data Bases. — VLDB Endowment, 2007. — Pp. 1426–1427. — (VLDB '07). — ISBN 978-1-59593-649-3.

24. Extendible Hashing — a Fast Access Method for Dynamic Files / R. Fagin [et al.] // ACM Trans. Database Syst. — 1979. — Vol. 4, no. 3. — Pp. 315–344. — ISSN 0362-5915. — DOI: 10.1145/320083.320092.

25. Extracting Equivalent SQL from Imperative Code in Database Applications / K. V. Emani [et al.] // Proceedings of the 2016 International Conference on Management of Data. — ACM, 2016. — Pp. 1781–1796. — (SIGMOD '16). — ISBN 978-1-4503-3531-7. — DOI: 10.1145/2882903.2882926.

26. *Fagin R.* The Decomposition Versus Synthetic Approach to Relational Database Design // Proceedings of the 3rd International Conference on Very Large Data Bases. Vol. 3. — VLDB Endowment, 1977. — Pp. 441–446. — (VLDB '77).

27. *Faroult S.*, *Robson P.* The Art of SQL. — O'Reilly Media, 2006. — P. 372. — ISBN 978-0-596-55536-8.

28. Feature Analysis of Generalized Data Base Management Systems: CODASYL Systems Committee, May 1971 / B. K. Bhargava [et al.]. — ACM, 1971.

29. *Fekete A.*, *O'Neil E.*, *O'Neil P.* A Read-only Transaction Anomaly Under Snapshot Isolation // SIGMOD Rec. — 2004. — Vol. 33, no. 3. — Pp. 12–14. — ISSN 0163-5808. — DOI: 10.1145/1031570.1031573.

30. *Fender P.*, *Moerkotte G.* Counter Strike: Generic Top-down Join Enumeration for Hypergraphs // Proc. VLDB Endow. — 2013. — Vol. 6, no. 14. — Pp. 1822–1833. — ISSN 2150-8097. — DOI: 10.14778/2556549.2556565.

31. *Gilbert S.*, *Lynch N.* Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services // SIGACT News. — 2002. — Vol. 33, no. 2. — Pp. 51–59. — ISSN 0163-5700. — DOI: 10.1145/564585.564601.

32. *Gotlieb L. R.* Computing Joins of Relations // Proceedings of the 1975 ACM SIGMOD International Conference on Management of Data. — ACM, 1975. — Pp. 55–63. — (SIGMOD '75). — DOI: 10.1145/500080.500089.

33. *Guttman A.* R-trees: A Dynamic Index Structure for Spatial Searching // SIGMOD Rec. — 1984. — Vol. 14, no. 2. — Pp. 47–57. — ISSN 0163-5808. — DOI: 10.1145/971697.602266.

34. *Hanani M. Z.* An Optimal Evaluation of Boolean Expressions in an Online Query System // Commun. ACM. — 1977. — Vol. 20, no. 5. — Pp. 344–347. — ISSN 0001-0782. — DOI: 10.1145/359581.359600.

35. *Harris E. P.*, *Ramamohanarao K.* Join Algorithm Costs Revisited // The VLDB Journal. — 1996. — Vol. 5, no. 1. — Pp. 064–084. — ISSN 1066-8888. — DOI: 10.1007/s007780050016.

36. *Hegner S. J.* Transaction Isolation in Mixed-Level and Mixed-Scope Settings // Advances in Databases and Information Systems / ed. by T. Welzer [et al.]. — Cham: Springer International Publishing, 2019. — Pp. 390–406. — ISBN 978-3-030-28730-6.

37. How Good Are Query Optimizers, Really? / V. Leis [et al.] // Proc. VLDB Endow. — 2015. — Vol. 9, no. 3. — Pp. 204–215. — ISSN 2150-8097.

38. Is This Document Relevant?…Probably: A Survey of Probabilistic Models in Information Retrieval / F. Crestani [et al.] // ACM Comput. Surv. — 1998. — Vol. 30, no. 4. — Pp. 528–552. — ISSN 0360-0300. — DOI: 10.1145/299917.299920.

39. *Kleppmann M.* Please stop calling databases CP or AP. — URL: https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html.

40. *Kossmann D.*, *Stocker K.* Iterative Dynamic Programming: A New Class of Query Optimization Algorithms // ACM Trans. Database Syst. — 2000. — Vol. 25, no. 1. — Pp. 43–82. — ISSN 0362-5915. — DOI: 10.1145/352958.352982.

41. *Lamport L.* The Part-time Parliament // ACM Trans. Comput. Syst. — 1998. — Vol. 16, no. 2. — Pp. 133–169. — ISSN 0734-2071. — DOI: 10.1145/279227.279229.

42. *Lehman P. L.*, *Yao S. B.* Efficient Locking for Concurrent Operations on B-trees // ACM Trans. Database Syst. — 1981. — Vol. 6, no. 4. — Pp. 650–670. — ISSN 0362-5915. — DOI: 10.1145/319628.319663.

43. *Litwin W.* Linear Hashing: A New Algorithm for Files and Tables Addressing // ICOD. — 1980. — Pp. 260–276.

44. *Maier D.* Theory of Relational Databases. — Computer Science Press, 1983. — P. 656. — ISBN 978-0-914-89442-1.

45. Making Snapshot Isolation Serializable / A. Fekete [и др.] // ACM Trans. Database Syst. — New York, NY, USA, 2005. — Июнь. — Т. 30, № 2. — С. 492—528. — ISSN 0362-5915. — DOI: 10.1145/1071610.1071615. — URL: http://proxy.library.spbu.ru:4267/10.1145/1071610.1071615.

46. *Mishra P.*, *Eich M. H.* Join Processing in Relational Databases // ACM Comput. Surv. — 1992. — Vol. 24, no. 1. — Pp. 63–113. — ISSN 0360-0300. — DOI: 10.1145/128762.128764.

47. *Normann R.*, *Ostby L. T.* A Theoretical Study of Snapshot Isolation // Proceedings of the 13th International Conference on Database Theory. — ACM, 2010. — Pp. 44–49. — (ICDT '10). — ISBN 978-1-60558-947-3. — DOI: 10.1145/1804669.1804677.

48. Omid: Lock-free transactional support for distributed data stores / D. G. Ferro [et al.] // IEEE 30th International Conference on Data Engineering, ICDE 2014 / ed. by I. F. Cruz [et al.]. — IEEE Computer Society, 2014. — Pp. 676–687. — ISBN 978-1-4799-3480-5. — DOI: 10.1109/ICDE.2014.6816691.

49. The Oracle PL/SQL. — URL: http://www.oracle.com/technetwork/database/features/plsql/index.html.

50. *Pavlo A.* What Are We Doing With Our Lives?: Nobody Cares About Our Concurrency Control Research // Proceedings of the 2017 ACM International Conference on Management of Data. — ACM, 2017. — Pp. 3–3. — (SIGMOD '17). — ISBN 978-1-4503-4197-4. — DOI: 10.1145/3035918.3056096.

51. *Ports D. R. K.*, *Grittner K.* Serializable Snapshot Isolation in PostgreSQL // Proc. VLDB Endow. — 2012. — Vol. 5, no. 12. — Pp. 1850–1861. — ISSN 2150-8097. — DOI: 10.14778/2367502.2367523.

52. *Robinson J. T.* The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes // Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data. — ACM, 1981. — Pp. 10–18. — (SIGMOD '81). — ISBN 0-89791-040-0. — DOI: 10.1145/582318.582321.

53. *Salton G.* Expert Systems and Information Retrieval // SIGIR Forum. — 1987. — Vol. 21, no. 3–4. — Pp. 3–9. — ISSN 0163-5840. — DOI: 10.1145/30075.30076.

54. *Salton G.* Recent Studies in Automatic Text Analysis and Document Retrieval // J. ACM. — 1973. — Vol. 20, no. 2. — Pp. 258–278. — ISSN 0004-5411. — DOI: 10.1145/321752.321757.

55. *Salton G.*, *Fox E. A.*, *Wu H.* Extended Boolean Information Retrieval // Commun. ACM. — 1983. — Vol. 26, no. 11. — Pp. 1022–1036. — ISSN 0001-0782. — DOI: 10.1145/182.358466.

56. The Software AG ADABAS Natural. — URL: http://www.softwareag.com/corporate/products/adabas_natural/natural/default.

57. *Stonebraker M.* The Design of the POSTGRES Storage System // Proceedings of the 13rd International Conference on Very Large Data Bases. — 1987. — Pp. 289–300. — (VLDB '87). — ISBN 0-934613-46-X.

58. The End of a Myth: Distributed Transactions Can Scale / E. Zamanian [et al.] // Proc. VLDB Endow. — 2017. — Vol. 10, no. 6. — Pp. 685–696. — ISSN 2150-8097. — DOI: 10.14778/3055330.3055335.

59. The Notions of Consistency and Predicate Locks in a Database System / K. P. Eswaran [et al.] // Commun. ACM. — 1976. — Vol. 19, no. 11. — Pp. 624–633. — ISSN 0001-0782. — DOI: 10.1145/360363.360369.

60. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles / N. Beckmann [et al.] // SIGMOD Rec. — 1990. — Vol. 19, no. 2. — Pp. 322–331. — ISSN 0163-5808. — DOI: 10.1145/93605.98741.

61. *Ullman J. D.* Principles of Database and Knowledge-Base Systems, Volume II. — Computer Science Press, 1989. — ISBN 0-7167-8162-X.

62. *Weikum G.*, *Vossen G.* Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. — Morgan Kaufmann, 2002. — ISBN 1-55860-508-8.

63. *Winand M.* SQL Performance Explained: Everything Developers Need to Know about SQL Performance. — Markus Winand, 2012. — P. 204. — ISBN 978-3-9503078-2-5.

64. Информационные системы общего назначения: аналитический обзор систем управления базами данных / под ред. Е. Ющенко. — М.: Статистика, 1975. — С. 472.

65. *Кузнецов С. Д.* Базы данных: учебник для студ. учреждений высшего проф. образования. — М.: Академия, 2012. — С. 496. — (Университетский учебник. Прикладная математика и информатика). — ISBN 978-5-7695-8430-5.

66. *Мейер Д.* Теория реляционных баз данных. — М.: Мир, 1987. — С. 608.

67. *Моргунов Е.* PostgreSQL. Основы языка SQL.: учеб. пособие / под ред. Е. Рогова, П. Лузанова. — СПб.: БХВ-Петербург, 2018. — С. 336. — ISBN 978-5-9775-4022-3. — URL: https://postgrespro.ru/education/books/sqlprimer.

68. *Новиков Б.*, *Домбровская Г.* Настройка приложений баз данных. — СПб.: BHV, 2006. — ISBN 5-94157-840-7.

69. *Рогов Е.* Индексы в PostgreSQL. — 2017. — URL: https://habr.com/company/postgrespro/blog/326096/.

70. *Селко Д.* SQL для профессионалов. Программирование. — М.: Лори, 2004. — С. 456. — ISBN 978-5-85582-219-9.

71. Лекции лауреатов премии Тьюринга за первые 20 лет / под ред. Ю. Баяковского. — М.: Наука, 1993. — С. 560. — ISBN 978-5-03-002130-0.

72. *Черкасова П.*, *Новиков Б.* Бесшовное проектирование: снова о несоответствии импеданса // Программирование. — 2006. — Т. 32, № 5. — С. 268—275.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.

При оформлении заказа следует указать адрес (полностью), по которому
должны быть высланы книги; фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: www.a-planeta.ru.
Оптовые закупки: тел. +7 (499) 782-38-89.
Электронный адрес: books@alians-kniga.ru.

Новиков Борис Асенович
Горшкова Екатерина Александровна
Графеева Наталья Генриховна

Основы технологий баз данных

Учебное пособие

При поддержке Postgres Professional
https://postgrespro.ru

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Редактор *Рогов Е. В.*
Корректор *Абросимова Л. А.*
Дизайн обложки *Климковский А. В.*

Формат $70\times100^{1}/_{16}$.
Гарнитура «ПТ Сериф». Печать цифровая.
Усл. печ. л. 47,3. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.ru