Владимирский государственный университет

ОСНОВЫ ПРОЕКТИРОВАНИЯ БАЗЫ ДАННЫХ НА POSTGRESQL

Учебное пособие

Владимир 2026

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

ОСНОВЫ ПРОЕКТИРОВАНИЯ БАЗЫ ДАННЫХ НА POSTGRESQL

Учебное пособие

Электронное издание

Владимир 2026

ISBN 978-5-9984-2155-6

УДК 004.65
ББК 32.972.134

Автор-составитель С. А. Курьерова

Рецензенты:
Кандидат технических наук, доцент
доцент кафедры информатики и защиты информации
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
Д. А. Полянский

Старший преподаватель кафедры информационных систем
и программной инженерии Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых,
ведущий разработчик компьютерного программного обеспечения
ООО «КУЛ» (г. Владимир)
А. И. Петрова

Основы проектирования базы данных на PostgreSQL [Электронный ресурс] : учеб. пособие / авт.-сост. С. А. Курьерова ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2026. – 82 с. – ISBN 978-5-9984-2155-6. – Электрон. дан. (1,10 Мб). – 1 электрон. опт. диск (CD-ROM). – Систем. требования: Intel от 1,3 ГГц ; Windows XP/7/8/10 ; Adobe Reader ; дисковод CD-ROM. – Загл. с титул. экрана.

Материал учебного пособия охватывает базовую часть дисциплины «Основы проектирования базы данных» и излагается на примере проектирования СУБД PostgreSQL. Представлено создание рабочей среды, приводятся основные определения, а также операции манипулирования данными.

Предназначено для студентов СПО очной формы обучения направлений подготовки 09.02.07 – Информационные системы и программирование, 09.02.09 – Веб-разработка.

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС СПО.

Табл. 4. Ил. 6. Библиогр.: 13 назв.

ISBN 978-5-9984-2155-6

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Базы данных играют ключевую роль в современных информационных системах, обеспечивая эффективное хранение, управление и доступ к данным. Одной из самых мощных и широко используемых открытых систем управления базами данных (СУБД) является PostgreSQL, которая предоставляет множество возможностей как для разработчиков, так и для администраторов баз данных.

В учебном пособии излагаются основы проектирования баз данных, приводятся основные концепции, описана работа с PostgreSQL. При изучении материала пособия обучающийся должен освоить навыки работы с PostgreSQL − создание, управление, составление запросов и др.

Пособие включает семь глав, которые последовательно знакомят с принципами работы с БД, начиная с общих понятий и заканчивая синтаксисом языка SQL. В конце каждой главы приводятся вопросы для самоконтроля, а также задания для самостоятельного выполнения.

Первая глава представляет собой вводный материал – излагаются основы и история развития баз данных, их виды и необходимость использования в современном информационном обществе. Описаны различные модели данных, чем они отличаются друг от друга. Приводятся существующие типы систем управления базами данных, описано предназначение. Рассматривается проектирование, начиная от общего подхода до более глубокого осмысления концептуального проектирования. Содержатся сведения об информационных, логических и физических моделях. Поэтапно представлены шаги создания базы данных, связь между объектами.

Во второй главе изложен более специализированный материал – система управления базами данных PostgreSQL. Описаны тонкости установки, настройки и работы с этой популярной СУБД, лексическая структура SQL в PostgreSQL. Осваивая материал главы, обучающийся должен научиться различать и правильно задавать идентификаторы и ключевые слова. В главе описаны основные операторы и правила использования комментариев, существующие типы данных.

В третьей главе обучающийся на практике познакомится с такими инструментами, как pgAdmin и psql, а также операторами и для чего они используются.

В четвертой главе рассматриваются работа со связанными таблицами, способы ограничения и объединения таблиц для создания сложных структур данных.

В пятой главе описаны индексы в SQL, их виды, как использовать различные индексы для повышения производительности запросов. Излагается материал о привилегиях пользователей и ролях в PostgreSQL.

В шестой главе даются понятия функции, процедуры и триггера, чем они отличаются друг от друга. Описаны работа с курсорами для обработки результатов и создание перечислительного типа. Перечисления открывают новые возможности для структурирования данных.

Глава седьмая содержит материал о том, как работать с данными в формате JSON в контексте PostgreSQL.

Изучение материала пособия будет полезной отправной точкой для дальнейшего освоения дисциплин и формирования профессиональных компетенций в сфере информационных технологий у обучающихся направлений подготовки «Информационные системы и программирование» и «Веб-разработка».

Глава 1. ЗНАКОМСТВО С БАЗАМИ ДАННЫХ И ИХ ПРОЕКТИРОВАНИЕМ

Глава посвящена теоретическим аспектам баз данных. Понимание основ базы данных является ключевым элементом для профессионала, поскольку базы данных представляют собой основу хранения и организации данных в современных приложениях.

В главе даны понятие и назначение базы данных. Рассматриваются основные модели данных, различные типы систем управления базами данных.

Материал глава раскрывает основы проектирования базы данных, будет способствовать пониманию бизнес-требований и операционных аспектов, а именно, какие данные должна хранить БД и как они будут использоваться. Рассматривается процесс моделирования данных, чем логическая модель отличается от физической.

Описаны процесс организации данных с целью устранения избыточности и уменьшения объема хранимой информации, этапы создания базы данных, что происходит на каждом этапе.

Обучающийся на практике освоит базовые операции для работы с таблицами; как создавать таблицы и их заполнять, обновлять данные и др.

Изложены приемы концептуального моделирования, которые используются для фиксации данных, а также фиксации между пользовательскими запросами. В основу ложатся существующие отчеты.

В качестве модели предметной области рассмотрим «Кинотеатр», а также базу данных этой модели. База данных позволяет пользователям извлекать данные, получать необходимую информацию, вести обработку данных, которые относятся к предметной области. Манипуляции с данными можно производить разными способами. Пользователям нужна информация и для этого «Кинотеатр» должен успешно работать. То есть можно говорить о том, что модели выступают в качестве важного средства, которое может отвечать требованиям пользователей.

Необходимо разобрать понятия, связанные с таблицами, определить, какие взаимосвязи могут возникать между таблицами, рассмотреть разные операции, которые можно выполнять в тот момент, когда происходит удаление или же обновление данных.

1.1. База данных и зачем она нужна

В наше время хранение данных становится необходимым ввиду постоянного роста объемов информации. Структурированный набор данных называют базой данных. Для контроля за информацией, которая превращается в данные, созданы специализированные инструменты, известные как системы управления базами данных. Они представляют целый комплекс программных средств, задача которых – обеспечение удобства и эффективности управления данными. СУБД нацелены на оптимизацию работы с информацией, позволяя обеспечивать порядок в огромном количестве данных. Рассмотрим на примере, зачем используют базы данных.

Допустим, мы открыли кинотеатр. Для продажи билетов в онлайн-режиме необходимо создать сайт, на котором будет находиться и поддерживаться актуальная информация о сеансах кинопоказов. Первый шаг – создание базы данных, которая будет содержать информацию о сеансах. В нашем случае информацию удобно записать в виде таблицы, где каждая строка – определенный сеанс, а каждый столбец – свойство. Следующий шаг – необходимо понять, что нужно делать с имеющимися данными. Есть основные операции, которые выполняют пользователи:

1. *Запись новых данных*, чтобы постоянно добавлять новые сведения о сеансе.

2. *Изменение старых данных*, например для изменения количества билетов на сеанс.

3. *Поиск данных* – нахождение сеансов на завтра, чтобы показать их клиенту.

4. *Разграничить права для сотрудников и клиентов*, чтобы клиенты не смогли изменить данные.

5. *Поддержание порядка в данных*, чтобы в категории «Доступные места» было именно количество свободных мест, без учета зарезервированных.

6. *Масштабирование базы данных* для добавления новых данных и исключения ограничений по объему.

7. *Обеспечение сохранности*, чтобы всегда было можно восстановить базу данных.

За последние десятилетия специалисты последовательно разработали несколько различных систем, работающих на основе трех моделей данных.

Модель данных – концептуальный способ структурирования данных для более удобного восприятия и использования. К моделям относятся иерархическая, сетевая и реляционная. Каждая из моделей имеет свои уникальные особенности, позволяя на их основе создавать структурированные и гибкие СУБД.

Иерархические и сетевые модели данных

Первая информационная система, которая использовала базы данных, появилась в 60-х годах XX столетия, она основывалась на иерархической модели. ***Иерархическая модель*** – модель данных, в которой связи между данными имеют вид иерархической структуры.

В иерархической базе данных отношения между элементами устанавливаются с помощью указателей (рис. 1). Указатели являются физическим адресом, обозначающим место хранения записи на диске. Благодаря указателям, начиная с вершины дерева (корневой вершины) можно легко определить спецификацию любого объекта данных. Тем не менее иерархический подход имеет свои ограничения, поскольку не все отношения можно представить в виде иерархии. Также необходимо предварительно использовать определенную структуру дерева и наличие только одного родительского элемента для каждого дочернего, что ограничивает гибкость при обработке более сложных сценариев.

В этом контексте были разработаны сетевые системы управления базами данных в конце 1960-х гг. Они представляли собой усовершенствованную версию иерархического подхода и обеспечивали более гибкую структуризацию данных. Именно этот фактор расширяет область применения сетевого подхода в отличие от иерархического. Как и в иерархических, в сетевых системах управления базами данных для связывания данных использовались указатели. Сетевые базы данных используют связь «многие-ко-многим», что позволяет удовлетворить широкий спектр требований (рис. 2).

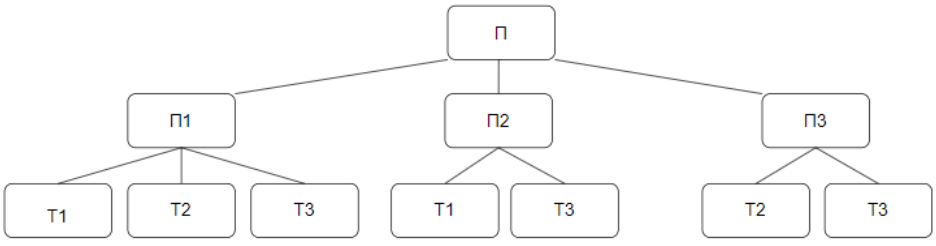

Рис. 1. Иерархическая модель

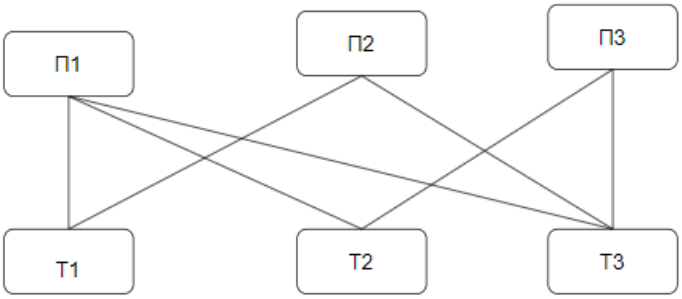

Рис. 2. Сетевая модель

Реляционные системы управления данными

Указатели в иерархических и сетевых СУБД были их сильной стороной, но в то же время были ограничителями и требовали настройки перед работой. Достоинством указателей являлось быстрое извлечение данных, связанных определенными отношениями. Недостаток заключался в том, что отношения определялись до запуска системы.

Впервые реляционные СУБД появились во второй половине 1970-х годов. Они поддерживали такие языки, как SQL (Structured Query Language, язык структурированных запросов), Quel (Query Language, язык запросов) и QBE (Query-by-Example, запросы по образцу). Массовое распространение персональных компьютеров в повседневной жизни в 1980-х годах привело к появлению реляционных СУБД для микрокомпьютеров. Причем наиболее привлекательными оказались СУБД семейства xBase, включая такие, как Clipper, dBase и FoxPro. Внедрение этих инноваций значительно расширило возможности СУБД.

Основным фундаментом СУБД является реляционная алгебра, которая определяет систему над отношениями (таблицами) – объединение, пересечение, вычитание, соединение и др. Все эти операции выражаются через язык SQL для выполнения запросов. Углубляться в детали реляционной алгебры в данном учебном пособии не будем, но она играет важную роль в работе СУБД.

Современные реляционные базы данных стали стандартом для коммерческой работы с данными, их используют малые предприятия и глобальные корпорации. Они управляют информацией веб-сайтов, систем электронной коммерции, банковских операций и др. Между тем,

стоит обозначить, что файловые, иерархические и сетевые базы данных не утратили своего значения в практической сфере.

Технологии, связанные с реляционными базами данных, претерпевают продолжительные изменения и активно развиваются, адаптируясь к новым потребностям. Развитие направлено на обеспечение предоставления нужных инструментов для решения более сложных проблем. Особое внимание заслуживает прогресс в области объектно-ориентированных баз данных. Также стоит отметить усиление перехода к технологиям «клиент-сервер» при работе с данными.

Если говорить о том, как представить реляционную базу данных, то здесь можно провести параллель с электронными таблицами по типу MS Excel. Они также состоят из множества столбцов и строк, организованных в таблицы (рис. 3). Каждая строка символизирует отдельную запись, в то время как столбец представляет специфическое поле в базе данных. В общем смысле база данных воплощает некое собрание данных, тесно связанных друг с другом.

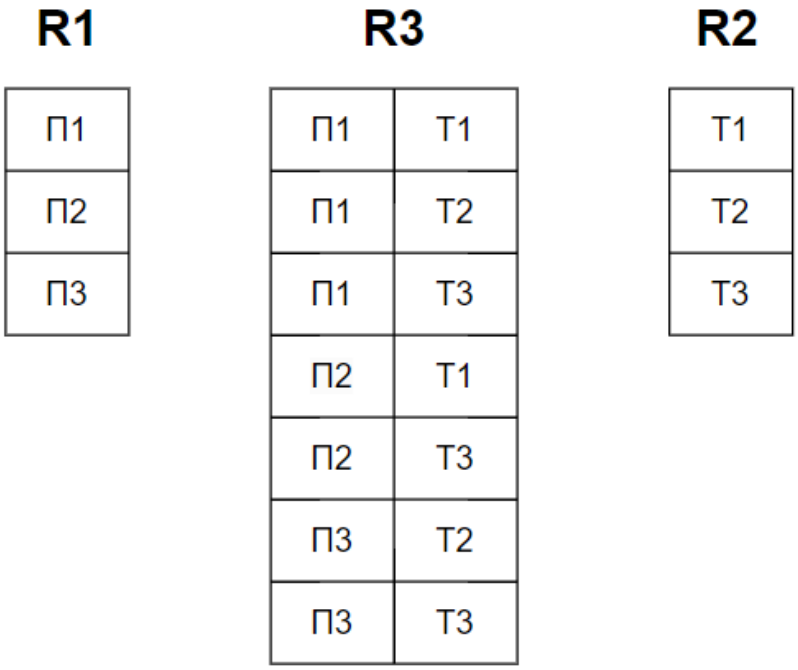

Рис. 3. Схема реляционной базы данных

Как итог, в настоящее время существует большое количество разнообразных баз данных. Иерархические и сетевые реализации не потеряли своего значения. Исполняемый тип СУБД зависит от специфических особенностей и целей конкретного проекта.

1.2. Типы СУБД

Системы управления базами данных отвечают за поддержку языка БД, оптимизацию процессов и механизмов извлечения данных. Рассмотрим основные типы СУБД более подробно.

Все серверные файлы должны храниться на сервере, при этом располагаются они централизованно. Для доступа к ним используется СУБД, которая находится на клиентском компьютере, что дает возможность клиенту получить доступ к серверному файлу, а значит к базе данных. Такая модель имеет свои преимущества. В качестве основного можно отметить низкую нагрузку на сервер. Если говорить о недостатках, можно отметить достаточно высокую нагрузку, которая наблюдается в локальной сети, что часто приводит к сложностям в сферах безопасности, доступности, надежности. Кроме того, файл-серверные СУБД не способны справляться с современными высокими нагрузками, связанными с обработкой колоссального количества данных, такая технология считается устаревшей.

В случае клиент-серверных систем и СУБД, и БД размещаются на сервере, к которому осуществляется доступ удаленно с клиентских машин. Все клиентские машины обрабатываются централизованно самой СУБД. Слабым звеном в данном случае становится сервер, на который приходится вся нагрузка. Однако сервера неплохо масштабируются и обеспечивают высокую производительность.

В настоящее время клиент-серверные СУБД являются самыми популярными, с их помощью можно добиться приемлемого быстродействия, обеспечить высокую доступность, надежность и безопасность.

Отдельный вид – встраиваемые СУБД. Современные клиент-серверные СУБД в большинстве случаев тяжеловесные, мощные комплексы программных систем. Как правило, их не просто установить на любой машине и эту проблему решают встраиваемые СУБД. Они могут использоваться в современных смартфонах, в данном случае будут выступать как составные части некоторого программного обеспечения. Таким образом, каждое приложение может тащить за собой небольшую копию этого СУБД для хранения своих локальных данных, обеспечивая быструю работу с ними. Соответственно, такие СУБД не рассчитаны на использование по сети, то есть они не инсталлируются на каком-то сервере. Встраиваемая СУБД поставляется в качестве библиотеки, которая подключается к некоторому программному продукту и обменивается с ним данными.

1.3. Информационные модели

База данных – это информация, которая относится к определенной области.

Для того чтобы понимать данные, применяются информационные модели. К примеру, можно рассмотреть область медицины. В ней используются модели, которые дают возможность анализировать заболевание, прогнозировать процесс лечения пациентов. Чтобы можно было использовать базу данных в определенных областях, необходимо поставить цель и определить область действий. В наше время все чаще используют вычислительные средства, при этом они зависят от области.

Модель данных отображает реальные меры и включает в себя основные детали. Предметная область в этом случае – данные, которые необходимо сохранить, то есть на которых основывается база данных.

В каждой предметной области содержатся принципиально важные понятия и данные, а также те данные, которые практически не имеют значимости. Разработка структуры базы данных предполагает акцентирование на первом типе данных, в то время как менее важные данные могут быть опущены.

Структура данных является составляющей для каждой модели и состоит из трех ключевых компонент:

1) структура данных предназначена для отражения взгляда пользователя на базу данных;

2) допускает определенные операции, которые могут быть выполнены на ее основе. Подчеркнем, что даже самая удачно построенная структура данных без возможности управления мало ценна;

3) модель данных предусматривает ограничения для контроля поддержки и защиты целостности. С помощью этих ограничений обеспечиваются долговечность и надежность.

Особенный тип данных, подразумевающий специфический вид модели, называется даталогической моделью, принадлежащей к логическому слою. Эта модель реализуется через встроенные инструменты системы управления базами данных и отличается от других своими особенностями. Она обеспечивает представление взаимосвязей между данными, не затрагивая их связь с местом хранения или конкретным содержимым данных.

При построении логической модели присутствуют определенные факторы, на них стоит обратить свое внимание – различные ограничения, которые накладывает на нее сама СУБД, а также особенности темы, которую описывает данная модель.

Также стоит выделить описание предметной области инфологической моделью, которое формулируется без привязки к программным и техническим возможностям, использующимся в будущем. Иногда в контексте инфологической модели возможно рассмотрение информационных потребностей пользователей. Эта модель служит отправной точкой перед началом создания логической модели.

В процессе связки логической модели со средой хранения используются модели физического уровня. Эти модели содержат всю необходимую информацию об установленных правилах расположения данных и соответствующих устройствах хранения информации. Строить модель физического уровня необходимо, включая ограничения, накладываемые СУБД и операционной системой.

Деление моделей различных уровней абстракции позволяет:

1) разбить сложный процесс представления данных «предметная область – база данных» на отдельные, более удобные для восприятия и воспроизведения этапы;

2) гарантировать обеспечение специализации для разработчиков баз данных и предоставление возможности работы разных групп пользователей соответствующего уровня;

3) вовлекать в процесс создания базы данных людей, у которых отсутствуют профессиональные навыки в области обработки данных и предоставлять им возможности активного участия;

4) формировать условия для автоматизации процесса проектирования баз данных через формализированный переход от одного уровня модели к другому.

Следует учесть, что техническое и программное обеспечение современных информационных систем довольно быстро обновляется и меняется. Вследствие чего модели также подвержены изменениям. Однако инфологическая модель отражает характеристики предметной области, поэтому она сравнительно стабильна.

1.4. Создание базы данных

Построение базы данных включает шесть важных этапов.

Первый этап – стадия планирования, которая подразумевает определение внутренних особенностей любой информационной системы. На данном этапе происходят тщательное изучение и фиксация всех имеющихся данных, которые документируются в единой концептуальной модели.

Следующий шаг – сбор необходимой информации. Разработчик на основе собранной информации устанавливает связи между различными приложениями, определяет, как информация может использоваться этими приложениями, выявляет потенциальные потребности в информационной системе.

Для реализации созданной БД подготавливается вся необходимая документация по важным аспектам. В качестве первого пункта изучают, есть ли технология, необходимая для создания планируемой базы данных, далее анализируют ресурсы компании. Итог – оценка эффективной работоспособности базы данных

Следующий важный шаг – исследование вопроса окупаемости системы: сколько времени понадобится, чтобы она окупилась, оцениваем риски, возможность обмена данными между отделами, соотнесение с долгосрочными планами развития, анализ преимуществ.

Второй этап – определение требований к проекту. На этом шаге дополняют и углубляют сведения, собранные на первом этапе. Более детально определяют требования к программным продуктам и оборудованию, которое потребуется. Для лучшего понимания, какую информацию нужно обрабатывать, проводятся опросы и исследуются анкеты.

Далее проектируют связи, то есть создают систему базы данных: строят модели, которые затем объединяют в единую модель. Каждая модель должна охватывать определенные данные, то есть разрабатываются детализированные модели данных для различных отделов, потом модели соединяются воедино. В результате получается общая схема базы данных на концептуальном уровне.

Третий этап – реализация результатов, достигнутых на предыдущих этапах. Выбирают и приобретают систему управления данными. Идет процесс преобразования концептуальной модели в физическую. После чего формируют словарь данных, в итоге происходит заполнение базы данных необходимой информацией.

Следующим шаг – создание прикладных программ для работы с базой и обучение пользователей.

Необходимо произвести оценку построения БД с помощью сбора обратной связи от пользователей. Это поможет исправить недочеты, расширить и дополнить базу данных новыми возможностями.

1.5. Процесс нормализации таблиц

Проектирование БД включает в себя несколько ключевых целей, рассмотрим их более детально.

Первая, и самая важная, – обеспечить хранение необходимых данных в базе. Подразумевается, что вся ценная информация находится в удобном, доступном и защищенном месте для последующего использования.

Для обеспечения эффективности избыточность данных стоит исключить, так как повторяющиеся данные не только занимают объем, но и могут усложнить их обработку.

Следующая цель – стремление к максимальному сжатию количества таблиц в базе данных. С одной стороны, разделение большой таблицы на несколько маленьких поможет исключить некоторые проблемы, например уменьшить вероятность дублирования данных. С другой стороны, это усложнит процесс разработки и увеличит затрачиваемое время на генерацию отчетов для пользователей.

Для решения всех вопросов, связанных с обновлением, удалением и исключением избыточности, разработчики прибегают к механизму нормализации. Этот процесс нацелен на преобразование исходной таблицы в несколько таблиц по определенным правилам и является ключевым в теории проектирования баз данных.

Важный принцип, который является основой реляционной системы первой нормальной формы (1НФ), заключается в том, что таблица должна быть двумерной и не включать ячейки со множественными значениями (табл. 1).

Таблица 1. Таблица 1НФ

Фильм	Режиссер
Черная молния. Джентльмены, удачи.	Дмитрий Кисилев
Сталинград. Притяжение.	Федор Бондарчук

Вторая нормальная форма (2НФ) − когда ни один из неключевых атрибутов не подчиняется функциональной зависимости от первичного ключа. Стоит отметить, что нарушение второй нормальной формы возможно при наличии составного ключа с участием нескольких атрибутов. Рассмотрим процесс разделения таблицы на две 2НФ.

В первоначальной таблице определяются атрибуты, зависящие только от ключа. Затем формируется новая таблица, в которой атрибутами будут элементы первоначальной таблицы. Определитель становится основным элементом этой таблицы. Атрибут, расположенный в правой части функциональной зависимости, удаляется из первоначальной таблицы. Если второй нормальной таблице противоречит более чем одна функциональная запись, то действия, описанные выше, повторяются для каждой из них. Если один определитель включен в несколько функциональных зависимостей, все они обрабатываются вместе (табл. 2).

Таблица 2. 2НФ

Номер фильма	Название	Жанр	Номер сеанса
1	Притяжение	Научная фантастика. Боевик	1
1	Притяжение	Научная фантастика. Боевик	2
2	Черная молния	Научная фантастика. Боевик	1

Термин «третья нормальная форма» (3НФ) относится к реляционной структуре. Если рассмотреть функциональную зависимость X→Y, то в 3НФ X всегда будет ключом. Это условие одновременно

обеспечивает выполнение второй нормальной формы, так как любая таблица в третьей нормальной форме автоматически удовлетворяет условиям таблицы второй нормальной формы. Относительно первой нормальной формы таблицы необходимо избавить от атрибутов, представляющих множество значений.

Следует разбить таблицу фильмов на две таблицы: Т1 (номер фильма, название, жанр) и Т2 (номер фильма, номер сеанса), внешним ключом является НОМЕР ФИЛЬМА.

Четвертая нормальная форма появляется, когда таблица, обладающая свойствами третьей нормальной формы, дополнительно не имеет множественных зависимостей. Это условие является решением множественных атрибутов. Каждый атрибут такого типа размещается в отдельной таблице вместе с ключом, от которого он зависит.

Кроме вышеупомянутых, выделяют и другие нормальные формы, которые предлагаются с целью устранения различных аномалий и привлекательны только с теоретической точки зрения, поэтому в пособии рассматривать их не будем.

1.6. Базовые операции для работы с таблицами и столбцами

В предыдущих параграфах мы познакомились с базами данных в целом и изучили принципы их построения. Теперь перейдем к практической части и рассмотрим основные операции для работы с таблицами и столбцами.

Для создания структуры данных, таких как таблицы, существует специальная команда «CREATE TABLE». Она должна содержать название будущей таблицы и перечисление столбцов с указанием типа данных, которые вы хотите в ней видеть. Например:

```
CREATE TABLE movies (
id SERIAL PRIMARY KEY,
name VARCHAR (255),
release_year INT,
);
```

В примере мы создали таблицу movie с указанием названий столбцов «id», «name», «release_year» и типами данных. При этом столбец «id» имеет ограничение в виде первичного ключа.

После создания таблицы перейдем к ее заполнению. Для вставки данных используется команда «INSERT INTO» с указанием имени таблицы и перечислением значений для каждого столбца.

```
INSERT INTO movies (name, release_year)
VALUES ('Мастер и Маргарита', 2005);
```

Таким образом, мы создали таблицу и заполнили ее некоторыми данными. Если на этапе заполнения произошла ошибка, в этом случае на помощь приходит команда «UPDATE». С ее помощью выбираем таблицу и конкретный столбец для обновления, указывая новое значение для замены старого.

```
UPDATE movies
SET release_year = 2024
WHERE name = 'Мастер и Маргарита';
```

Обновляем значение столбца «release_year» для фильма «Мастер и Маргарита». Теперь проверим наши данные путем извлечения из таблицы. Для выборки используется команда «SELECT» с указанием столбцов, данные из которых необходимо получить.

```
SELECT name, release_year
FROM movies;
```

В примере выводим данные из столбцов «name» и «release_year».

Бывают ситуации, когда необходимо удалить определенные строки из таблицы. Для выполнения этого воспользуйтесь специальной командой «DELETE FROM». Она требует названия таблицы и условия, которые определяют, какие строки подлежат удалению.

```
DELETE FROM movies
WHERE release_year < 2024;
```

В данном примере удаляем все записи из таблицы «movies» для фильмов, год выпуска которых был раньше 2024 года.

Все эти команды образуют основной набор инструментов для работы с данными в таблицах и столбцах. В совокупности с другими командами мы сможем эффективно использовать их для более сложного манипулирования данными.

1.7. Объекты и отношения

В качестве основной цели процесса моделирования данных выступает представление о концептуальной схеме. Такая схема может быть представлена в одной или нескольких моделях, эти модели могут стать основой баз данных. Объекты, а также отношения, возникающие между объектами, являются элементами концептуальной модели [1].

В качестве объектов могут выступать зрители, фирмы и предметы, которые имеют значение. Отношение – это очередь, возникающая между объектами. Отношения ложатся в основу при составлении объективного множества.

Мощность отношений – это наибольшее количество элементов, которые включены в один объект. Все эти элементы имеют взаимосвязь между собой. На рис. 4 приведены примеры отношений разной мощности.

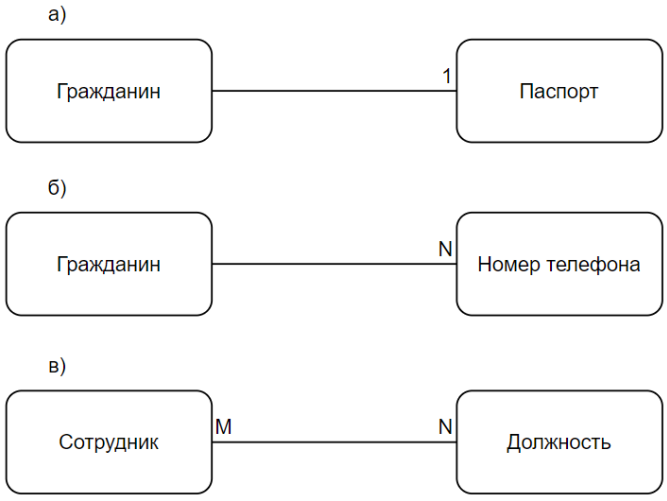

Рис. 4. Отношения разной мощности

Говорить о максимальной мощности можно во всех отношениях, которые протекают в обоих направлениях, то есть в данном случае об отношениях, которые можно назвать «один к одному» (см. рис. 4, а).

Если максимальная мощность отношения в одном направлении равна одному, а в другом − многим, то такое отношение является отношением «один-ко-многим» (см. рис. 4, б).

Если максимальная мощность в обоих направлениях равна многим, то такое отношение называется отношением «многие-ко-многим» (см. рис. 4, в).

1.8. Атрибуты и ключи

В зависимости от атрибутов множеств формируются элементы. Атрибуты используются для того, чтобы различать элементы одного множества. Если рассматривать отношения от объекта к атрибутам, то такие отношения можно назвать функциональными. *Атрибут* – это функциональное отношение объектного множества, он может принадлежать одному объекту или же нескольким из множества. Возможно это в том случае, если атрибуты правильно используются, то есть для каждого элемента должен быть однозначно определен атрибут.

Рассмотрим это на примере: если говорить о фильме, то в качестве атрибута может выступать дата его выхода. Атрибут может иметь и пустое значение в том случае, если однозначно определить атрибут будет невозможно.

Каждое множество объектов имеет как минимум один ключ, по которому можно идентифицировать конкретный экземпляр множества. Этот ключ может быть принят в качестве первичного.

Набор атрибутов, который исключает невозможность однозначной идентификации объекта, называется минимальным ключом. При выборе первичного ключа стоит использовать некомпозитные ключи, избегая длинных текстовых значений в пользу целочисленных атрибутов.

Составной ключ содержит более одного атрибута. Например, для идентификации фильма можно использовать идентификационный номер или набор, включающий название фильма, ФИО режиссера и другую дополнительную информацию. Это важно, так как возможно наличие фильмов с одинаковыми названиями.

Атрибут, который является первичным ключом, не может иметь неопределенного значения, так как это в большинстве случаев приводит к появлению неуникальных экземпляров объектов. Именно поэтому важно обеспечить уникальность первичного ключа.

Для установления связей между объектами в базах данных служат внешние ключи. Например, если множество А связывает множества В и С, то оно должно содержать внешние ключи, которые буду соответствовать первичным ключам сущностей этих множеств.

Элементы объектов могут иметь свои внутренние номера. Эти номера не имеют смысла в определенной системе. Такие номера получили название суррогатных ключей или же идентификаторов объектов. Они однозначно определяют объекты реального мира. Их целесообразно использовать, если атрибут должен играть роль ключа, но не обладает уникальностью или может менять свое значение со временем. Например, название киностудии или номер кинозала. Также для исключения многократного дублирования атрибута, имеющего большую длину, целесообразно использовать суррогатный ключ в виде кода.

Не всякому объекту нужен ключ. Например, в таблице, отслеживающей посещаемость, необязательно иметь уникальный ключ для каждого просмотра, потому что для анализа посещения важна информация о фильме, времени сеанса, возрастной категории зрителя и т. д.

1.9. Реляционная модель данных

В настоящее время ученые разработали несколько подходов, которые применяются для проектирования реляционной базы данных.

Первый подход основывается на концептуальной модели. После построения такой модели она преобразуется в реляционную модель. Для этого используются определенные правила.

Второй подход говорит о том, что на этапе концептуального проектирования необходимо выстраивать реляционные таблицы. Процесс проектирования в дальнейшем будет заключаться в том, что происходит нормализация этих таблиц. Для этого применяются стандартные процедуры.

В реляционной модели вся информация представлена в виде таблиц. Сама же реализация заключается в построении двумерной таблицы, она состоит из строк и столбцов, которые выступают в качестве

атрибутов. Число атрибутов показывает степень реляций, а значение атрибутов принадлежит их области. Возможно также наличие пустых значений, присваиваемых атрибутам в случае неприменимости или неизвестности.

Реляционная схема базы данных представляет собой список реляционных таблиц с перечислением их атрибутов и определений внешних ключей.

Контрольные вопросы и задания

1. Назовите принципы нормализации и перечислите проблемы, которые они позволяют решить.
2. Что такое база данных и для чего она используется? Какие плюсы использования баз данных по сравнению с обычными файлами вы можете назвать?
3. Какие этапы следует пройти при проектировании базы данных?
4. Чем объекты отличаются от атрибутов?
5. Опишите взаимосвязь трех видов моделей данных.
6. Проанализируйте предметную область, выделите основные объекты и их атрибуты, укажите связи между таблицами.
7. Разработайте схему базы данных для небольшой компании, например, интернет-магазина, библиотеки, университета. Приведите разработанную схему к третьей нормальной форме.
8. Чем хранение информации в базах данных отличается от хранения в файлах?
9. Дайте сравнительную характеристику иерархических, сетевых и реляционных информационных систем.
10. Посетите сайт одной из крупных организаций и попробуйте определить, какие базы данных могут использоваться при хранении информации на сервере компании.

Глава 2. ВВЕДЕНИЕ В POSTGRESQL И ЛЕКСИЧЕСКУЮ СТРУКТУРУ ЯЗЫКА SQL

В этой главе рассмотрим, как может использоваться язык программирования SQL, а также познакомимся с базой данных PostgreSQL. На текущий момент продолжают развиваться инструменты визуального программирования, которые автоматизируют команды SQL.

SQL является непроцедурным языком, который ориентирован на выборку данных, результат запросов – набор данных, чаще всего это таблица. Язык содержит различные категории запросов, рассмотрим каждую из них отдельно.

Почему PostgreSQL?

Во-первых, PostgreSQL бесплатная система управления базами данных, но эта характеристика не является ключевой. PostgreSQL поддерживает большую часть стандарта SQL и предлагает множество функций − от сложных запросов до управления многоверсионным параллелизмом. Кроме того, система может быть расширена пользователем многими способами. Например, путем добавления новых типов данных, функций, операторов и т. д. Стоит отметить, что в любом случае 90 % возможностей диалекта SQL можно без изменения применять в других СУБД.

Поговорим о синтаксисе SQL. Команда представляет собой последовательность компонентов, завершающуюся точкой с запятой (";"), конец входного потока считается концом команды. Конкретные допустимые компоненты для каждой команды зависят от ее синтаксиса.

Компонентом команды могут быть ключевое слово, идентификатор, идентификатор в кавычках, строка (или константа) или специальный символ. Компоненты обычно разделяются пробельными символами (пробел, табуляция, перевод строки), но это не обязательно, если нет неоднозначности (например, спецсимвол находится рядом с другим компонентом). Кроме того, ввод SQL может содержать комментарии, которые эквивалентны пробелам.

В отношении того, какие компоненты идентифицируют команды, а какие являются операндами или параметрами, синтаксис не очень последовательный.

Рассмотрим, основные типы данных, которые позволяют хранить различные данные в БД [2].

2.1. Что такое PostgreSQL?

PostgreSQL (сокращенно от Postgre Structured Query Language) – мощная, надежная и популярная объектно-реляционная система управления базами данных (СУБД). Она предоставляет широкий спектр возможностей для хранения, организации, поиска и обработки структурированных данных.

PostgreSQL является клиент-серверной СУБД, когда СУБД и БД располагаются на сервере, а к СУБД осуществляется удаленный доступ с клиентских машин. Все клиентские машины обрабатываются централизованно этой самой СУБД. Слабым звеном в данном случае становится сервер, на который идет вся нагрузка. Однако серверы неплохо масштабируются и имеют неплохие характеристики производительности [3].

В наше время клиент-серверные СУБД – самые популярные. С их помощью можно добиться приемлемого быстродействия, обеспечить высокую доступность, надежность и безопасность.

PostgreSQL разработана как открытое программное обеспечение и предоставляет свободный доступ к своему исходному коду. Это позволяет разработчикам и администраторам настраивать и расширять функциональность системы в соответствии с потребностями своего проекта.

2.2. Основные особенности PostgreSQL

Рассмотрим некоторые ключевые особенности PostgreSQL, которые способствуют ее выбору разработчиками и администраторами.

Во-первых, это поддержка стандартного языка SQL и предоставление богатого набора функций для запросов. Поддержка множества типов данных и возможность создания пользовательских типов, адаптирующих базу данных под индивидуальные потребности проекта.

Во-вторых, поддержка триггеров и хранимых процедур, которые позволяют автоматизировать сложные запросы и бизнес-логику. Также поддержка операций с геопространственными данными, предоставление расширения PostGis, которое позволяет хранить и оперировать географическими данными.

В-третьих, предоставление различных методов репликаций данных, поддержка транзакций, блокировок и обработок ошибок. Эти возможности позволяют создавать отказоустойчивые, масштабируемые системы, обеспечивая надежность и целостность данных.

Еще одна особенность – активная и дружественная поддержка сообществом разработчиков и пользователей, которые готовы предоставить помощь, если возникают вопросы или проблемы.

2.3. Как начать использовать PostgreSQL?

Чтобы начать использовать PostgreSQL, необходимо скачать и установить ее на свой ПК. PostgreSQL доступна для различных операционных систем, включая Windows, macOS и Linux. После установки вы сможете создать базу данных, настроить пользователей и начать работу с SQL и администрированием базы данных.

Для начала работы с PostgreSQL вам потребуется выбрать дистрибутив, который вы хотите установить. Устанавливать рекомендуется последнюю стабильную версию. Рассмотрим этот процесс более детально:

Первым делом, необходимо посетить официальный веб-сайт PostgreSQL – https://www.postgresql.org/ и перейти в раздел "Загрузки". Затем выбрать версию, соответствующую вашей операционной системе, и загрузить ее. Далее следуйте инструкциям по установке, указывая параметры установки по вашему усмотрению (например, расположение установки, пароль для учетной записи "postgres" и т. д.).

После успешной установки вы можете запустить PostgreSQL, используя командную строку или графический интерфейс, если он предоставляется для вашей операционной системы.

2.4. Идентификаторы и ключевые слова

Идентификаторы, а также ключевые слова должны иметь одну и ту же лексическую структуру, так как определить их при использовании ключевого слова или же идентификатора, не зная SQL, невозможно.

Все буквы начинаются с названия идентификатора, а также названия ключевых слов. При этом не допускается использование подчеркивания. Вторым символом в идентификаторе могут выступать как буквы, так и цифры.

С точки зрения стандарта SQL использовать подчеркивание в начале или в конце идентификатора запрещено [2]. Объясняется это тем, что в таком случае идентификатор защищен от различных конфликтов, которые в будущем могут возникать.

Есть еще один тип идентификаторов, в которых используются кавычки или запятые, и чаще всего последний символ заключается в кавычки.

Идентификаторами могут быть любые символы, кроме символа с кодом 0. Иногда появляется необходимость использования кавычек внутри идентификатора. В этом случае кавычки должны дублироваться. Использование данного правила дает возможность создавать столбцы или целые таблицы, в именах которых будут пробелы. Здесь стоит помнить, что ограничения относительно длины идентификатора сохраняются.

Если идентификатор заключен в кавычки, он выступает в качестве регистрозависимого, если идентификатор без кавычек, во всех случаях он должен преобразовываться в нижний регистр.

Есть еще один формат для идентификатора. Он дает возможность использовать символы Unicode. Данный идентификатор начинается с «U&», после этого ставится двойная кавычка, в качестве примера рассмотрим «U&"foo"». Используя данную информацию, стоит помнить, что оператор & имеет неоднозначность, которую можно исключить, если применить вокруг & пробел.

2.5. Операторы

Все операторы должны иметь не больше 63 символов, в качестве символов могут быть использованы: + - * / <> = ~! @ # % ^ & | ` ?

Есть ограничения, которые относятся к именам для операторов. В качестве первого ограничения можно отметить то, что сочетание символов -- и /* нельзя использовать в имени оператора. Объясняется это тем, что сочетание данных символов рассматривается как начало комментария.

Если имя оператора состоит из нескольких символов, заканчиваться знаком минус или плюс оно не может, если оно не содержит таких символов, как ~! @ # % ^ & | ` ?. Использовать в качестве имени оператора символ @- можно; в качестве имени оператора символ*- – недопустим.

Рассматривая работу операторов, которые отличаются от стандартов, использующихся для SQL, стоит помнить, что необходимо применять пробелы для разделения соседних операторов. Это даст возможность избежать двусмысленности.

2.6. Комментарии

Документация – ключевой элемент каждого программного продукта. Специфическая последовательность символов, игнорируемая при исполнении кода и предназначенная для разъяснения действий кода, известна как комментарий. В SQL обычные комментарии обозначаются парой символов --. Существуют два типа комментариев, охватывающих несколько строк:

/* длинный комментарий на несколько строк

* с возможностью вложения: /* комментарий внутри другого комментария */

/, где комментарий начинается с / и продолжается до соответствующего вхождения */. Эта возможность блоковых комментариев соответствует стандарту SQL, позволяя эффективно отключать большие сегменты кода, даже если в них уже присутствуют комментарии. Обычно комментарии изымаются из кода до начала его синтаксического анализа, эффективно превращаясь в пробелы.

2.7. Специальные символы

Некоторые символы, отличные от буквенно-цифровых, обладают уникальными функциями, не связанными с операторами, такие символы называются общеупотребимыми.

Общеупотребимые символы и их предназначения:

• символ доллара $, следующий за числом, предназначен для обозначения позиционного аргумента внутри определения функции

или оператора. В других случаях он может служить частью идентификатора или строковой константы, ограниченной долларовыми кавычками;

- круглые скобки ( ) используются в их привычном значении для организации выражений и увеличения приоритета. В определённых ситуациях символ необходим как элемент фиксированного синтаксиса некоторых SQL команд;
- квадратные скобки [ ] применяются для выборки элементов из массива;
- запятые (,) играют роль в различных синтаксических структурах, разделяя элементы списка;
- точка с запятой (;) заканчивает SQL команду. Нахождение этого символа внутри команды допустимо только в строковых константах или кавычках, обрамляющих идентификатор;
- двоеточие (:) используется для обозначения выборки части или «среза» массивов. В некоторых вариациях SQL двоеточие также служит префиксом для имен переменных;
- звезда (*) в некоторых контекстах указывает на все поля строки или составного значения. Особое значение приобретает при использовании в качестве аргумента агрегатной функции, обозначая отсутствие необходимости в явном параметре;
- точка (.) встречается в числовых константах и служит для разграничения имён схемы, таблицы и столбца.

2.8. Типы данных

При создании таблиц и БД нужно понимать, что данные представляются определенными типами – типами данных. Таблица может содержать числа, строки, даты и т. д. Основные типы данных можно разбить на несколько категорий.

Первая категория – целые числа. Они представлены тремя основными типами данных: smallint, integer – стандартизированный выбор пользователей, bigint – выбирается реже. К целочисленным типам данных также можно отнести следующие: smallserial, serial, bigserial; разница заключается в том, что когда мы колонку объявляем одним из

трех перечисленных типов, то PostgreSQL при вставке строки в таблицу будет автоматически инкрементировать, то есть увеличивать это значение на единицу и записывать его.

Вторая категория – вещественные числа или числа с «плавающей» точкой, например, 1.4 или 15.65 и т. д. Выделяют три основных типа:

decimal/numeric – этот тип поддерживает большой диапазон значения, самое главное, что эти значения обычно связаны с денежными расчетами. Он позволяет делать более точные расчеты в то время, как другие типы склонны к накоплению ошибок при проведении между ними арифметических операций;

real/float4 и double/precision/float8/float чаще всего применяются для дробных чисел, когда эти значения используются в научных вычислениях. Почему эти типы данных не поддерживают точных вычислений объясняется сложной математической теорией, в данном учебном пособии рассматривать этот вопрос нет необходимости.

Следующая категория – символы. Char(n) представляет из себя строку фиксированной длины, где n – количество символов в строке. Char(n) применяется в тех случаях, когда мы точно знаем, что строка будет той длинны, которую мы задали. Если в такой столбец попытаться вставить строку большей длины, будет возникать ошибка, если вставлять строку меньшей длины, чем задано, то она будет добиваться справа пробелами до требуемой длины. Поддерживаемый диапазон значений зависит от кодировки. Varchar(n) отличается от char(n) тем, что, если задано, например varchar(10), и кто-то передаст строку меньшей длины, например 5, она не будет расширяться до 10 – это основное отличие. К сожалению, если длину задали 10, но пытаемся вставить строку большей длины, будет также возникать ошибка.

Тип text поддерживает строки совершенно случайной длины. Мы не задаем n, количество символов зависит от того, какой текст вставили.

Отдельный тип – логический, используется для логических вычислений. Такие строки, как y, yes, n, no, 1 и 0, конвертируются без проблем соответственно в true или false [4].

Также стоит выделить категорию типов, которые отвечают за представление даты и времени. Более детально характеристики показаны в табл. 3.

Таблица 3. Основные типы данных PostgreSQL

Название	Байт	Описание	Диапозон
smallint	2	Целое число с малым диапозоном	− 32.768 до 32.768
integer	4	Стандартный выбор для целых чисел	− 2.147.483.648 до 2.147.483.648
bigint	8	Целое число с большим диапозоном	−9.223.372.036.854.775.808 до 9.223.372.036.854.775.808
smallserial	2	Автоувеличение целого числа малого диапозона	1 до 32.767
serial	4	Автоувеличение числа среднего диапозона	1 до 2.147.483.647
bigserial	8	Автоувеличение числа большого диапозона	1 до 9.223.372.036.854.775.807
decimal/ numeric	variable	Заданная пользователем погрешность, точная	$\pm 3.4*10^{38}$ до $+3.4*10^{38}$
real/float4	4	Заданная пользователем погрешность, неточная	Точность до 6 знаков после запятой
double preci-sion/float8/ float	8		Точность до 15 знаков после запятой
char	variable	Строка символов фиксированной длины	Базовая кодировка
varchar	variable		Базовая кодировка
text	variable		Базовая кодировка
boolean/ bool	1	Используется в логике	True/ false
date	4	Сохраняет только дату	4713 год до н.э. → 294.276 год
time	8	Сохраняет только время	00:00:00 → 24:00:00
timestamp	8	Сохраняет дату и время	4713 год до н.э. → 294.276 год
interval	16	Сохраняет разницу между временными метками	− 178.000.000 → + 178.000.000
timestampt z	8	Сохраняет временную метку и часовой пояс	4713 год до н.э. → 294.276 год + часовой пояс

Контрольные вопросы и задания

1. Опишите основные особенности работы с PostgreSQL по сравнению с другими СУБД.

2. Какие типы СУБД вы узнали?

3. Посетите официальный сайт PostgreSQL, выберите необходимый дистрибутив и установите на свой компьютер.

Глава 3. ОСНОВЫ И РАБОТА SQL В POSTGRESQL

В этой главе обсудим применение PostgreSQL на практике. Создадим базу данных с помощью таких инструментов, как psql и pgAdmin.

Рассмотрим различные операторы фильтрации, сортировки и группировки, которые помогают оперировать данными – эффективно извлекать и анализировать их, делая более структурированными и удобными для использования.

Фильтрация позволяет выбирать только те строки, которые соответствуют этим условиям. Сортировка данных упорядочивает результаты запроса, а группировка позволяет объединить результирующие строки по значениям определенного столбца.

Функции являются важной частью SQL и позволяют выполнять различные операции над данными. В PostgreSQL есть множество встроенных функций, а также возможность создания пользовательских функций.

Строковые функции дают возможность выполнять операции над строковыми значениями, временные − работают с датами и временем, математические – выполняют различные операции над числами.

Вложенные запросы позволяют избежать использования временных таблиц или производить более сложные операции, используя результаты подзапросов.

Всё вышеперечисленное позволяет делать запросы более гибкими, а также упрощает их исполнение, что важно для обеспечения эффективности работы с данными.

3.1. Создание базы данных

После установки и запуска PostgreSQL попробуем создать саму базу данных. Необходимо открыть инструмент для работы, такой как psql для командной строки или pgAdmin для графического интерфейса, авторизоваться под пользователем postgres. При работе с графическим интерфейсом авторизация происходит автоматически.

pgAdmin позволяет нам делать множество операций с помощью диалоговых окон без использования языка SQL. Чтобы получить доступ к диалоговому окну, следует развернуть дерево управления и правой кнопкой мыши нажать по необходимому типу объекта, после чего выбрать опцию Создать. В нашем случае мы будем создавать базу данных: нажимаем на узле Databases и выбираем Create Database, после этого откроется диалоговое окно, в котором происходит базовая настройка БД с помощью вкладок «Общие», «Определение», «Безопасность» и «Параметры».

На вкладке «Общие» необходимо присвоить имя БД, указать владельца базы данных, по умолчанию это пользователь «postgres», а также внести заметки о БД (рис. 5).

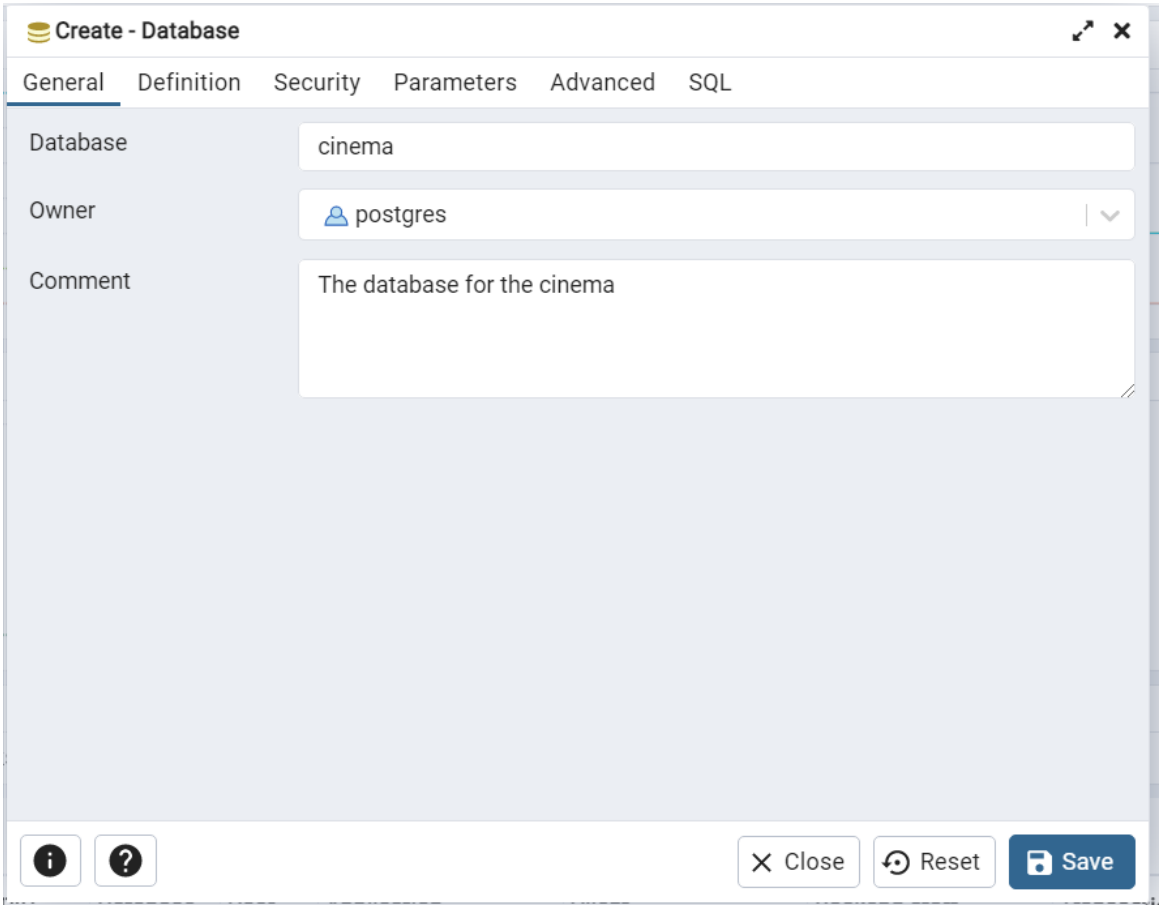

Рис. 5. Диалоговое окно «База данных»

Перейдём к следующей вкладке «Определение» (рис. 6), необходимой для задания свойств кодировки, по умолчанию стоит стандартная кодировка UTF-8. Далее задается шаблон, на основе которого создается база данных, указываем табличное пространство, параметры

сортировки, тип символа. Задание ограничения на количество подключений по умолчанию равно –1 – это значит, что ограничений нет. Также можно указать, что создаваемая база данных будет шаблонной.

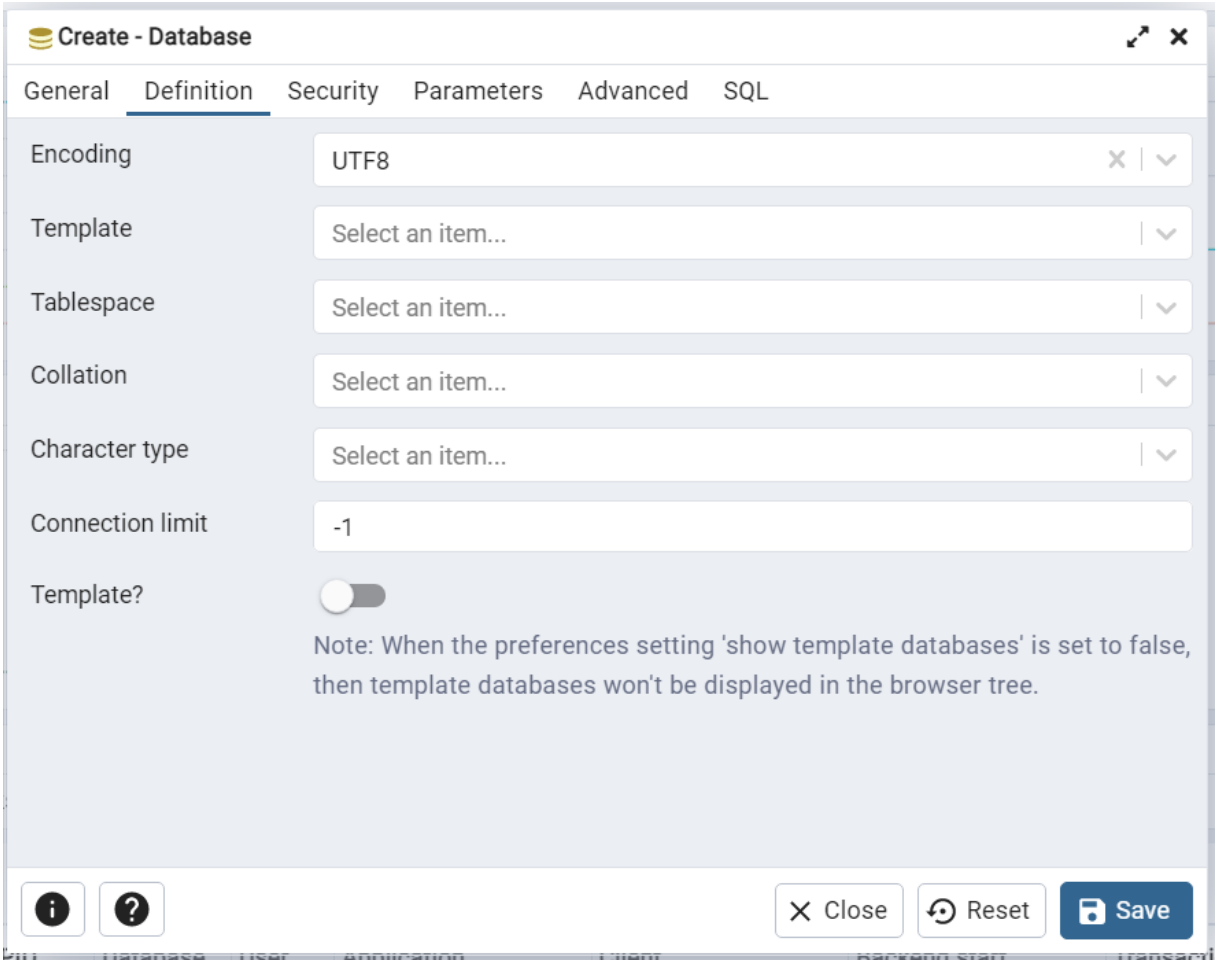

Рис. 6. Диалоговое окно «Определение»

На вкладке «Безопасность» можно управлять пользователями и давать им конкретные привилегии, но это отдельная большая тема. Вкладки «Параметры» и «Дополнительно» предназначены для задания определенных параметров БД. На вкладке «SQL» можно просмотреть код SQL, сгенерированный в результате настроек [5].

После установки необходимых параметров ваша база данных создана, но она в настоящий момент пустая.

Теперь рассмотрим создание базы данных через командную строку. Для авторизации под пользователем «postgres» необходимо ввести следующую команду:

```
sudo -u postgres psql
```

После успешной авторизации переходим к созданию базы данных. Поскольку в pgAdmin мы не изменяли никакие параметры по умолчанию, можно воспользоваться командой:

```
CREATE DATABASE name_database;
```

Итак, база данных создана, теперь можно приступить к использованию PostgreSQL для хранения и обработки информации.

3.2. Операторы условий

Операторы условий позволяют фильтровать данные, основываясь на определенных заданных условиях. Рассмотрим основные операторы и некоторые примеры для запросов.

Операторы для сравнения значений: ">", "<", ">=", "<=", "=". Например, выведем результат запроса, в котором возрастные ограничения равны 16+:

```
SELECT * FROM films
WHERE age_restrictions = 16;
```

Логические операторы "AND", "OR" используются для комбинирования условий, а оператор "NOT" – для инвертирования условий. Например, выведем результат запроса, где возрастные ограничения меньше или равны 16+ и фильм снят в драматическом стиле:

```
SELECT * FROM films
WHERE age_restrictions <=16 AND genre='Драма';
```

Операторы "LIKE", "ILIKE" используются для сопоставления строковых значений с шаблоном. Оператор "LIKE" чувствителен к регистру, а оператор "ILIKE" – нет. Если шаблон не содержит знака процента или подчёркивания, то представляется поиск в точности по строке. Подчёркивание означает, что вместо него подходит любой символ. Знак процента говорит о любой последовательности символов. Например:

```
SELECT * FROM films
WHERE name LIKE 'Мастер%';
```

В нашем примере мы выбираем все фильмы, у которых имя начинается с "Мастер".

Специальные оператор "IN" используется для проверки: содержится ли значение в списке значений; оператор "BETWEEN" предназначен для проверки нахождения значения в указанном диапазоне. Например, выведем фильмы с годом выпуска 2022, 2023 и 2024:

```
SELECT * FROM films
WHERE year_release IN (2022, 2023, 2024);
```

3.3. Операторы сортировки

Операторы сортировки позволяют упорядочивать результаты запроса и задавать порядок, в котором они возвращаются.

Оператор "ORDER BY" используется для сортировки результатов по определенному столбцу. "ASC" и "DESC" указывают направления сортировки. "ASC" означает сортировку по возрастанию (по умолчанию), а "DESC" – по убыванию. Например:

```
SELECT * FROM films
ORDER BY year_release DESC;
```

В примере мы выбираем все фильмы и сортируем их по году выпуска в убывающем порядке.

Если нам необходимо выполнить сортировку по нескольким столбцам, добавляем имеющиеся в таблице столбцы к списку сортировки:

```
SELECT * FROM films
ORDER BY year_release DESC, name ASC;
```

Представленный запрос выполняет упорядочивание результата сначала по году выпуска в убывающем порядке, а затем в алфавитном порядке по названию фильма.

Операторы "NULLS FIRST" и "NULLS LAST" используются для указания порядка сортировки NULL значений. "NULLS FIRST" означает, что NULL значения будут располагаться в начале списка (по умолчанию), а "NULLS LAST" – в конце списка. Например, необходимо проверить, у всех ли фильмов добавлено описание:

```
SELECT * FROM films
ORDER BY descriptuon ASC NULLS FIRST;
```

Выбираем все фильмы и сортируем их по описанию в порядке возрастания с NULL значениями, расположенными в начале списка.

Оператор "LIMIT" используется для ограничения количества возвращаемых строк. Например, выведем первые десять фильмов в алфавитном порядке:

```
SELECT * FROM films
ORDER BY name
LIMIT 10;
```

Таким образом, с помощью операторов сортировки можно управлять порядком данных, что позволяет предоставлять пользователю отсортированные и упорядоченные результаты запроса.

3.4. Операторы агрегации

Операторы агрегации в SQL используются для вычисления статистических данных на основе группировки и фильтрации. Эти функции выполняют вычисления над группами строк и возвращают единственное значение. Ниже приведены некоторые операторы агрегации, которые можно использовать в PostgreSQL.

Оператор "COUNT" используется для подсчета общего количества строк в выборке. Например:

```
SELECT COUNT(*) FROM visitors;
```

В примере подсчитывается общее количество записей зрителей, которые записаны в системе.

Оператор "SUM" используется для вычисления суммы значений столбца. Например, сделаем подсчет общей суммы, полученной за один сеанс:

```
SELECT s. id AS session_id,
SUM(ts.price) AS total_sales
FROM sessions s
JOIN tickets t ON s.id = t.id_session
JOIN reservation r ON t.id = r.id_ricket
JOIN ticket_sales ts ON r.id = ts.id_ reservation
GROUP BY s.id;
```

Не стоит бояться такого запроса, далее будет рассмотрено, для чего нужны операторы "JOIN" и "GROUP BY". В этом запросе мы объединяем необходимые нам таблицы по соответствующим внешним ключам, после этого применяем агрегатную функцию SUM() для вычисления общей суммы по идентификатору сеанса.

Оператор "AVG" используется для вычисления среднего значения столбца. Например:

```
SELECT AVG(duration)
FROM films;
```

В этом примере вычисляется средняя продолжительность всех фильмов.

Оператор "MIN" и "MAX" используют для определения минимального и максимального значений столбца соответственно. Например, найдем минимальное и максимальное значения стоимости сеанса фильма:

```
SELECT MIN(price), MAX(price)
FROM sessions;
```

Операторы агрегации позволяют совершать вычисления на основе данных в таблицах при анализе и обработке больших объемов данных. В зависимости от потребностей их можно комбинировать и использовать в разных контекстах.

3.5. Операторы группировки

Операторы группировки в SQL позволяют сгруппировать строки в результате запроса и выполнять агрегацию данных для каждой группы. Это необходимо, когда вы хотите рассчитать статистику или проанализировать данные в контексте определенных групп. Ниже приведены некоторые операторы группировки и их применение на практике.

Оператор "GROUP BY" используется для группировки результатов запроса по одному или нескольким столбцам. Синтаксис использования оператора:

```
SELECT column1, column2, aggregate_operator (column3)
FROM table_name
GROUP BY colunm1, column2, …;
```

Вернёмся к примеру, где мы подсчитывали сумму общей продажи (п. 3.4), используя оператор "GROUP BY" для группировки результатов по идентификатору кинозала (id_cinema_hall). Эта процедура объединяет строки в итоговом датасете на основании значений в одном или нескольких указанных столбцах. SQL группирует строки по заданным столбцам, после чего к каждой группе применяются агрегатные функции.

Оператор "HAVING" используется для фильтрации групп после выполнения группировки и применения агрегатных операторов. Например, у нас есть таблица с информацией о сеансах и продажах билетов. Найдем кинозалы, где общая сумма продаж билетов превышает определённое значение:

```
SELECT s.cinema_hall_id,
SUM(ts.price) AS total_sales
FROM sessions s
JOIN
ticket_sales ts ON s.id = ts.session_id
GROUP BY
s.cinema_hall_id
HAVING
SUM(ts.price) > 1000;
```

В этом запросе объединялись таблицы сессии и продажа билетов по соответствующему ключу, далее группировались результаты по идентификатору кинозала, вычисляя общую сумму продаж билетов для каждого кинозала, результаты фильтровались, оставляя только те кинозалы, где общая сумма продаж превышает 1000.

Таким образом, операторы группировки позволяют агрегировать данные в соответствии с заданными группами и выполнять операции над этими группами. Это весомый инструмент для анализа данных и получения сводной информации.

3.6. Строковые функции

Строковые функции используются для работы со строковыми значениями. Рассмотрим самые распространённые:

CONCAT() / CONCATENATE() объединяет две или более строки в одну. Например:

```
SELECT CONCAT(name, ' ', surname) AS full_name
FROM visitors;
```

В этом запросе объединяются имена и фамилии зрителей в одну строку.

LENGTH() / LEN() возвращает длину строки. Например, узнаем длину описания каждого фильма:

```
SELECT LENGTH(description) AS description_length
FROM films;
```

UPPER и LOWER преобразуют строку в верхний или нижний регистр соответственно:

```
SELECT UPPER(email)
FROM visitors;
```

В этом запросе преобразуем все электронные адреса зрителей в верхний регистр.

TRIM() удаляет пробелы или указанные символы с начала и конца строки. Синтаксис функции:

```
SELECT TRIM ('  ' FROM column_name)
FROM table_name;
```

3.7. Дата и временные функции

PostgreSQL предоставляет множество функций для работы с датами и временем. Рассмотрим основные функции.

CURRENT_DATE и CURRENT_TIME возвращают текущую дату и время соответственно. CURRENT_TIMESTAMP (или NOW) возвращает текущую метку времени (дату и время). Например:

```
SELECT CURRENT_DATE;
```

В этом запросе мы получаем текущую дату.

DATE_TRUNC усекает дату или время до указанной единицы. Например, округлим дату резервации билета до месяца:

```
SELECT DATE_TRUNC('month', date_reservation) AS month
FROM reservations;
```

EXTRACT извлекает часть (год, месяц, день и т.д.) из даты или времени. Например, извлечём месяц и день рождения посетителя:

```
SELECT
EXTRACT(MONTH FROM b_date) AS b_date_month,
EXTRACT(DAY FROM b_date) AS b_date_day
FROM visitors;
```

AGE() возвращает разницу между двумя датами. Например, найдем текущий возраст посетителя:

```
SELECT AGE(NOW(),b_date) AS age_of_visitors
FROM visitors;
```

3.8. Математические функции

PostgreSQL дает возможность применять множество математических функций для выполнения различных операций над числами: поддержка стандартных арифметических операций, тригонометрических функций, логарифмов и экспоненты, а также генерация случайных чисел RANDOM.

Возведение в степень POWER и извлечение квадратных корней SQRT. Например, вычисление квадратного корня из числа 16:

```
SELECT SQRT(16);
```

Возвращение абсолютного значения числа, то есть его удаление от нуля находится с помощью функции ABS. Например:

```
SELECT ABS(-10);
```

В этом запросе мы возвращаем абсолютное значение числа −10.

PostgreSQL предоставляет функции округления чисел (ROUND, CEIL, FLOOR) и усечения десятичных чисел TRUNC ():

```
SELECT ROUND(3.14159, 2);
```

В этом запросе было выполнено округление числа π до двух знаков после запятой.

Функции в SQL позволяют выполнять различные операции над данными, что значительно расширяет возможности запросов.

3.9. Вложенные запросы

В PostgreSQL вложенный запрос представляет SQL-запросы, которые содержатся в других SQL-запросах. Это может использоваться для выполнения более сложных операций, например: поиск значений с использованием подзапросов или комбинирование результатов нескольких запросов в одном. Вложенные запросы могут быть полезны для решения разнообразных задач, таких как анализ данных, сравнение значений, фильтрация результатов и многих других операций. Однако при создании вложенных запросов важно помнить о том, что они могут оказывать влияние на производительность базы данных, особенно при выполнении сложных и объемных операций.

Вложенные запросы могут встречаться в различных частях SQL запроса, таких как SELECT, FROM, WHERE, HAVING, и даже в блоках UPDATE и DELETE.

Пример вложенного запроса, который находит среднюю продолжительность фильмов, чьи названия начинаются с буквы "A":

```
SELECT AVG(duration)
FROM films
WHERE name IN (SELECT name FROM films WHERE name LIKE 'A%');
```

Этот запрос содержит вложенный запрос: сначала необходимо найти названия фильмов, начинающиеся с буквы "A", а затем использовать их для выбора продолжительности и вычисления среднего значения.

В PostgreSQL вложенный запрос может быть в различных частях запроса. Однако при использовании вложенных запросов важно учитывать производительность запроса, так как большое количество вложенных запросов может привести к медленной обработке запроса.

Кроме того, в PostgreSQL можно применять вложенные запросы для выполнения операций с несколькими уровнями вложенности. Например, можно создать запрос, который включает несколько уровней вложенности для выполнения сложных вычислений или получения более детальной информации из базы данных.

Рассмотрим пример комплексного подзапроса, цель которого: определить общий доход пользователей с именами на букву "A", чьи доходы выше среднего у пользователей с аналогичными именами:

```
SELECT name, SUM(income) as total_income
FROM users
WHERE name IN (
SELECT name
FROM users
WHERE name LIKE 'A%' AND income > (SELECT AVG(income) FROM users WHERE name LIKE 'A%')
)GROUP BY name;
```

Этот запрос содержит вложенные подзапросы в операторе WHERE для поиска имен пользователей, чьи доходы превышают средний доход пользователей с теми же именами, затем эти имена используются для вычисления общего дохода их суммарного значения.

Вложенные запросы увеличивают сложность SQL-запроса, но также предоставляют значительные возможности для анализа и обработки данных. При их использовании необходимо уделять внимание оптимизации запросов, избегать чрезмерной вложенности, чтобы обеспечить эффективность выполнения запросов.

Контрольные вопросы и задания

1. Что представляет собой лексическая структура в контексте баз данных?

2. Перечислите ключевые команды и особенности лексической структуры.

3. Напишите SQL-запрос для создания таблицы "users" с полями "id" (целочисленный), "name" (строка), "email" (строка) и "age" (целочисленный) в базе данных PostgreSQL.

4. В созданной вами базе данных создайте несколько таблиц и заполните их тестовыми данными. Напишите несколько SQL-запросов для выборки данных из ваших таблиц с использованием операторов.

Глава 4. СВЯЗАННЫЕ ТАБЛИЦЫ. ОГРАНИЧЕНИЕ И ОБЪЕДИНЕНИЕ ТАБЛИЦ

Таблицы, которые между собой связаны, называются связанными таблицами. Данные из разных таблиц соотносятся с помощью ключевых полей. Связи между таблицами – ключевой аспект проектирования баз данных. Связи позволяют эффективно организовывать и анализировать данные.

Ограничение данных исключительно с помощью типов данных может привести к недостоверной информации приложений. К примеру, столбец для цен должен допускать только положительные числа, однако не существует стандартного типа данных, ограничивающего значения исключительно положительными числами. Также может возникнуть необходимость в ограничении данных столбца в зависимости от значений других столбцов или строк.

Ограничения предоставляют возможность управлять данными в таблицах с любой степенью строгости. Если пользователь попытается ввести данные, нарушающие установленное ограничение, будет выдана ошибка. Это правило действует даже в случаях, когда нарушающее ограничения значение было получено в результате применения определения значения по умолчанию.

4.1. Первичный ключ

Ограничение первичного ключа подразумевает, что определённый столбец или комбинация столбцов служат уникальным идентификатором для записей в таблице. Это условие требует, чтобы их значения были уникальны и отличны от нуля:

```
CREATE TABLE films (
id SERIAL PRIMARY KEY,
name VARCHAR,
year_release INT,
 movie_director TEXT
);
```

Теория реляционных баз данных утверждает, что у каждой таблицы должен быть первичный ключ. Эти ключи не только играют важную роль в документации, но и критически значимы для клиентских приложений. Например, приложения для модификации данных в строках должны иметь возможность определения первичного ключа таблицы для точной идентификации записей.

4.2. Внешние ключи

Ограничение внешнего ключа устанавливает требование, чтобы значения в определённом столбце соответствовали значениям в столбце другой таблицы. Этому ограничению можно присвоить уникальное имя, подобно любым другим ограничениям. Внешний ключ может применяться и к набору столбцов, образуя связь между группами данных. Например:

```
CREATE TABLE t1 (
a integer PRIMARY KEY,
b целое число,
c целое число,
FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Естественно, количество и тип ограничивающих столбцов должны быть идентичны количеству и типу столбцов, на которые они ссылаются.

4.3. Ссылочная целостность таблиц

Исходя из ограничения внешних ключей поддерживается ссылочная целостность между двумя связанными таблицами. Допустим, есть таблица с фильмами, которую мы уже несколько раз использовали:

```
CREATE TABLE films (
id SERIAL PRIMARY KEY,
name VARCHAR,
year_release INT,
movie_director TEXT
);
```

Предположим, что есть таблица сеансов на эти фильмы. Мы хотим убедиться, что в таблице сеансов содержатся только те фильмы, которые действительно существуют, поэтому определяем ограничение внешнего ключа:

```
CREATE TABLE sessions (
id_film INT REFERENCES films(id),
date_session DATE,
time_session TIME,
);
```

Теперь невозможно создать записи сеанса с не NULL записями id_film, которые не присутствуют в таблице films. В этой ситуации таблица сеансов – это ссылающаяся таблица, а таблица фильмов – это ссылочная таблица.

Мы знаем, что внешние ключи запрещают создание сеансов, не относящихся ни к одному фильму. Существуют несколько вариантов, если фильм удаляется после создания сеанса, который на него ссылается, рассмотрим каждый вариант.

```
CREATE TABLE films (
id SERIAL PRIMARY KEY,
name VARCHAR,
);
CREATE TABLE sessions (
id_film INT REFERENCES films(id) ON DELETE RESTRICT,
date_session DATE,
);
```

Если кто-то хочет удалить фильм, на который все еще ссылается сеанс, следует запретить это делать. RESTRICT предотвращает удаление ссылающегося ряда, если эти строки всё ещё существуют, будет выдана ошибка. Эквивалентная RESTRICT опция NO ACTION: операции обновления или удаления в родительской таблице будут запрещены, если существуют соответствующие записи в дочерней таблице.

```
CREATE TABLE films (
id SERIAL PRIMARY KEY,
name VARCHAR,
);
CREATE TABLE sessions (
id_film INT REFERENCES films(id) ON DELETE CASCADE,
date_session DATE,
);
```

Каскадное удаление (CASCADE) означает, что при удалении записи в родительской таблице автоматически удаляются соответствующие записи в дочерних таблицах. Другими словами, действия, выполняемые в родительской таблице, распространяются на связанные записи в дочерних таблицах.

Два других варианта – SET NULL и SET DEFAULT – предусматривают, что при удалении записи, на которую есть ссылка, в связанных строках устанавливаются значения NULL или значения по умолчанию соответственно. Важно понимать, что использование этих вариантов не освобождает от необходимости соблюдать установленные ограничения. Так, если для действия задано SET DEFAULT, но значение по умолчанию не соответствует ограничениям внешнего ключа, операция не будет успешной [6].

Выбор опции зависит от требований вашего приложения и того, как вы хотите обрабатывать изменения в связанных данных при обновлении или удалении записей в родительской таблице.

4.4. Проверка ограничений

Ограничение CHECK – одно из наиболее универсальных типов ограничений, позволяющее проверять, соответствует ли значение в конкретном столбце определённому условию, которое должно быть истинным. Например, чтобы гарантировать, что цена сеанса всегда положительна, нужно указать следующее условие:

```
CREATE TABLE sessions (
date_session DATE,
time_session TIME,
price DECIMAL CHECK (price > 0)
);
```

Определение ограничения размещается сразу после указания типа данных, как и определение значений по умолчанию. При этом порядок указания значений по умолчанию и ограничений может быть любым. Ограничение типа CHECK задаётся ключевым словом CHECK, после которого в скобках следует условие. Условие в ограничении должно включать в себя столбец, иначе ограничение не будет иметь практического смысла.

Ограничения могут применяться к нескольким столбцам одновременно. Например, если имеются столбцы с обычной ценой и ценой со скидкой, можно использовать ограничение, чтобы гарантировать, что цена со скидкой всегда ниже обычной цены:

```
CREATE TABLE sessions (
date_session DATE,
time_session TIME,
price DECIMAL CHECK (price > 0),
discounted_price DECIMAL CHECK (discounted_price > 0),
CHECK (price > discounted_price)
);
```

Первые два ограничения проверяют, что значения стоимости являются положительными. Третье ограничение использует другой подход: оно не привязано к конкретному столбцу и представлено как независимый элемент в перечне столбцов таблицы. Таким образом, если первые два являются ограничениями на уровне столбца, то третье действует как ограничение таблицы в целом. Хотя ограничения на уровне столбца могут быть выражены как ограничения таблицы, обратное преобразование не всегда применимо, так как предполагается, что ограничение столбца непосредственно относится к столбцу, к которому оно привязано.

4.5. Именованное ограничение

Ограничению можно задать уникальное имя, что облегчает понимание сообщений об ошибках и дает возможность ссылаться на ограничение при необходимости его модификации. Синтаксис выглядит следующим образом:

```
CREATE TABLE sessions (
date_session DATE,
time_session TIME,
price DECIMAL CONSTRAINT positive_price CHECK (price > 0)
);
```

Чтобы задать именованное ограничение, применяется ключевое слово CONSTRAINT, после которого следует идентификатор, предшествующий определению самого ограничения. В случае отсутствия указания имени система автоматически генерирует его. Имена могут быть присвоены ограничениям на уровне таблицы, как и ограничениям на уровне столбцов:

```
CREATE TABLE sessions (
date_session DATE,
time_session TIME,
price DECIMAL CHECK (price > 0),
discounted_price DECIMAL CHECK (discounted_price > 0),
CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

Следует отметить, что ограничение проверки выполняется, если контрольное выражение принимает значение true или null.

4.6. Ненулевые ограничения

Ненулевое ограничение просто указывает, что столбец не должен принимать значение null. Пример синтаксиса:

```
CREATE TABLE films (
name VARCHAR NOT NULL,
year_release int NOT NULL,
movie_director TEXT NOT NULL,
description TEXT
);
```

Ограничение NOT NULL функционально эквивалентно созданию проверки ограничения CHECK (column_name IS NOT NULL). В PostrgeSQL создание явного ограничения NOT NULL более эффективно. Недостатком является то, что нельзя задавать явные имена ненулевым ограничениям, созданным таким образом.

Столбец может иметь более одного ограничения. В таком случае ограничения записываются одно за другим, порядок не имеет значения:

```
CREATE TABLE films (
name VARCHAR NOT NULL,
year_release int NOT NULL CHECK (year_release > 0),
 movie_director TEXT NOT NULL,
description TEXT
);
```

В подавляющей части проектов баз данных большинство столбцов должны быть обозначены как NOT NULL.

4.7. Уникальные ограничения

Уникальные ограничения гарантируют, что данные, содержащиеся в столбце или группе столбцов, уникальны среди всех строк таблицы. Чтобы определить уникальное ограничение для группы столбцов, его следует записать как ограничение таблицы, разделив имена столбцов запятыми:

```
CREATE TABLE example (
a integer,
b integer,
c integer,
UNIQUE (a, c)
);
```

В итоге имеем – комбинация значений в указанных столбцах уникальна во всей таблице, хотя любой из столбцов не обязательно и обычно не является уникальным.

Добавление уникального ограничения автоматически создаст уникальный индекс для столбца или группы столбцов. Как правило, ограничение уникальности нарушается, если в таблице имеется более одной строки, в которой значения всех столбцов равны. По умолчанию два нулевых значения не считаются равными, то есть при наличии уникального ограничения можно сохранить повторяющиеся строки, содержащие нулевое значение. Это можно изменить, добавив NULLS NOT DISTINCT, например:

```
CREATE TABLE films (
name VARCHAR UNIQUE NULLS NOT DISTINCT,
year_release int NOT NULL CHECK (year_release > 0),
movie_director TEXT NOT NULL,
description TEXT
);
```

4.8. Внешнее и внутреннее объединения

Запросы могут обращаться к нескольким таблицами одновременно или к одной и той же таблице таким образом, что несколько строк таблицы обрабатываются одновременно. Запросы, которые обращаются к нескольким таблицами, называются объединенными запросами. Они объединяют строки из одной таблицы со строками таблицы, причем оператор указывает, какие строки должны быть объединены.

Возможны внутренние, внешние и перекрестные соединения. Соединения всех типов могут быть выстроены цепочкой или вложены друг в друга. Общий синтаксис объединенной таблицы выглядит следующим образом:

```
Table1 join_type Table2 [join_condition]
```

Например, чтобы вернуть все записи о фильме с указанием сеансов, связанных с ними, базе данных нужно сравнить строки таблицы «фильмы» со столбцом названия всех строк таблицы «сеансы» и выбрать те пары, в которых значения совпадают.

```
SELECT * FROM films
JOIN sessions ON films.id = sessions.id_film;
```

Этот запрос вернет все строки из таблицы фильмов, объединенные соответствующими строками из таблицы сеансов. Однако в выходном наборе будут два столбца, содержащих идентификаторы фильма, на практике это нежелательно, поэтому перечислим выходные столбцы явно:

```
SELECT name, age_restriction, date_session, price FROM films
JOIN sessions ON films.id = sessions.id_film;
```

Поскольку все столбцы имеют разные имена, парсер автоматически определяет, к какой таблице они относятся. Если бы имена столбцов в двух таблицах дублировались, пришлось бы уточнить имена столбцов, например:

```
SELECT films.name, films.age_restriction, sessions.date_session, sessions.price FROM films
JOIN sessions ON films.id = sessions.id_film;
```

Существует возможность, чтобы запрос включил в результат даже те фильмы, у которых нет сеансов, для этого необходимо использовать внешнее соединение. До этого момента рассматривалось внутреннее соединение. Запрос выглядит следующим образом:

```
SELECT * FROM films
LEFT OUTER JOIN sessions ON films.id = sessions.id_film;
```

Запрос использует левое внешнее соединение, которое вернёт все строки из таблицы фильмов и соответствующие строки из таблицы сеансов. Если у фильма нет сеансов, то значения для полей, связанных с таблицей сеансов, будут NULL, если нужно определить все сеансы, включая те, для которых нет соответствующих фильмов, используют правое внешнее соединение – RIGHT OUTER JOIN.

Слова INNER и OUTER не являются обязательными в любых видах SQL JOIN операций. По умолчанию используется INNER JOIN, а для LEFT, RIGHT и FULL JOIN подразумевается использование внешнего соединения. Условие соединения может быть указано в предложении ON, USING или неявно с помощью ключевого слова NATURAL.

Предложение ON представляет собой наиболее универсальный способ задания условия соединения: принимая выражение с булевым результатом, аналогичное тому, которое используется в предложении WHERE, строки из таблицы фильмов и таблицы сеансов соединяются если условие, заданное в выражении ON, истинно.

Предложение USING – сокращение, позволяет воспользоваться специфической ситуацией, когда обе стороны соединения используют одно и то же имя объединяемого столбца. USING принимает список имён общих столбцов, разделённых запятыми, и формирует условие объединения, включающее равенство для каждого из них.

4.9. Перекрестное соединение

Для любой возможной пары строк из таблицы фильмов и таблицы сеансов, то есть для декартового произведения этих таблиц, результатом будет таблица, в которой каждая строка состоит из всех столбцов таблицы фильмов, за ними последуют все столбцы таблицы сеансов. Если в таблице фильмов есть N строк, а в таблице сеансов – M строк, то объединённая таблица будет содержать N * M строк.

```
SELECT * FROM films
CROSS JOIN sessions;
```

Перекрестные соединения могут привести к большому количеству строк в результирующем наборе, особенно если в таблицах много данных. В большинстве случаев для связывания данных предпочтительно использовать соединения с условиями соответствия, чтобы получить более конкретные и осмысленные результаты.

4.10. Типы квалифицированного соединения

Условие соединения определяет пары строк из двух таблиц, которые считаются «совпадающими». Существуют различные типы квалифицированных соединений:

1. INNER JOIN создает объединенную таблицу таким образом, чтобы для каждой строки из таблицы фильмов в ней находилась соответствующая строка из таблицы сеансов, удовлетворяющая условию соединения.

2. LEFT OUTER JOIN начинается с выполнения внутреннего соединения. После этого для каждой строки из таблицы фильмов, не имеющей соответствия в таблице сеансов согласно условию соединения, добавляется строка, где поля таблицы сеансов заполнены нулевыми значениями. Так, в объединенной таблице гарантированно будет, по крайней мере, одна строка для каждой строки из таблицы фильмов.

3. RIGHT OUTER JOIN начинается с выполнения внутреннего соединения. После этого для каждой строки из таблицы сеансов, не

нашедшей соответствия в таблице фильмов согласно условию соединения, добавляется строка с нулевыми значениями для столбцов таблицы фильмов. Это действие является обратным к действию LEFT OUTER JOIN, обеспечивая в итоговой таблице запись для каждой строки из таблицы сеансов.

4. FULL OUTER JOIN сначала выполняет внутреннее соединение, затем для строк из таблицы фильмов, которые не соответствуют условиям соединения с любой строкой из таблицы сеансов, включаются строки с нулевыми значениями для столбцов таблицы сеансов. Аналогично для строк таблицы сеансов, не удовлетворяющих условиям соединения с любой строкой таблицы фильмов, добавляются строки с нулевыми значениями для столбцов таблицы фильмов.

Контрольные вопросы и задания

1. Опишите принципы, связанные с ограничениями таблицы.
2. Зачем нужны ограничения, в чем их преимущество?
3. Как осуществляется объединение таблиц, перечислите основные виды объединений.
4. Создайте в своей базе данных связанные таблицы с ограничениями первичного и внешнего ключей; ограничения проверки, а также задайте значение по умолчанию для столбца, если оно не указано.
5. Какие операции можно проводить с помощью SQL-запросов?
6. Опишите принципы работы со связанными строками. С помощью чего обеспечивается ссылочная целостность?
7. Поработайте с SQL-операторами и функциями для различных запросов. Создайте от 3 до 5 запросов для выборки различных данных. Например, необходимо посчитать сколько и каких экземпляров книг нужно заказать поставщикам, чтобы на складе было одинаковое количество экземпляров каждой книги.

Глава 5. ИНДЕКСЫ И РАБОТА С ЗАПРОСАМИ. ТРАНЗАКЦИИ И ПРИВИЛЕГИИ ПОЛЬЗОВАТЕЛЕЙ

Индексы – это структуры данных, которые применяются для ускорения операций, т. е. производительности баз данных, позволяют серверу быстро находить и извлекать конкретные строки из таблицы, основываясь на значениях одного или нескольких столбцов. Однако индексы требуют ресурсов на своё обновление, занимают дополнительное пространство, поэтому к их созданию следует подходить рационально.

Когда создают индекс, сервер создает дополнительную структуру, которая содержит отображение значений столбца к физическим местоположениям записей в таблице.

Предположим, у нас есть таблица с фильмами:

```
CREATE TABLE films (
id SERIAL,
name varchar,
);
```

и приложение выдает множество запросов такого вида:

```
SELECT name FROM films WHERE id = 1234;
```

Без оптимизации системе придётся выполнять полное сканирование таблицы фильмов, проверяя каждую строку на соответствие условиям поиска. Это может быть крайне неэффективным, особенно если в таблице содержится большое количество строк, а результатом запроса являются лишь немногие из них. Однако, если системе дана команда поддерживать индекс по столбцу id, она сможет использовать более эффективный метод поиска, пройдя всего несколько уровней дерева индекса для нахождения нужных записей. В этом случае можно использовать команду для создания индекса по столбцу id таким образом:

```
CREATE INDEX films_id_index ON films (id);
```

Выбор имени для индекса – важный шаг, так как подходящее имя поможет в будущем быстро понять назначение индекса. Используйте имена, которые ясно отражают связь индекса с его функцией или столбцами, к которым он применяется.

Для удаления индекса используется команда DROP INDEX. Индексы могут быть добавлены или удалены в любой момент, делая

управление структурой базы данных гибким. После создания индекс автоматически обновляется системой при изменении данных в таблице, и будет использоваться в запросах, когда система определит, что это улучшит производительность запросов.

Индексы можно применять в командах обновления и удаления с условием поиска, а также в объединенном поиске. Индекс, определенный для столбца, который является частью условия объединения, может значительно ускорить выполнение запросов с объединениями.

Транзакции представляют собой ключевую концепцию во всех системах управления базами данных, обеспечивая целостность данных через механизм, который можно описать как операцию «всё или ничего». Они позволяют группировать несколько операций с базой данных в одну единую логическую единицу работы, гарантируя, что либо все операции в транзакции успешно выполнены и зафиксированы как единое целое, либо, в случае возникновения ошибки, все изменения откатываются, не оставляя никакого влияния на состояние базы данных.

5.1. Виды индексов

PostgreSQL предлагает разнообразие типов индексов, каждый из которых оптимизирован под конкретные сценарии использования и типы данных благодаря различным алгоритмам, лежащим в их основе. Среди доступных типов индексов находятся: B-tree, Hash, GiST, SP-GiST, GIN, BRIN, а также дополнительный модуль bloom.

Команда CREATE INDEX по умолчанию создаёт индексы типа B-tree, которые хорошо подходят для обширного спектра задач в виду своей универсальности. Если необходим другой тип индекса, его можно указать, используя ключевое слово USING, за которым следует название желаемого типа индекса. Например, для создания индекса типа Hash используется синтаксис:

```
CREATE INDEX name ON table USING HASH (column);
```

5.2. Индексы B-дерева

B-дерево эффективно работает с запросами, требующими проверки на равенство и диапазонных запросов по данным, которые организованы в определённом порядке. Планировщик запросов PostgreSQL рассматривает возможность использования индекса B-tree, когда индексируемый столбец задействован в операциях сравнения с использованием операторов: <, <=, >=, =, >. Это означает, что индекс B-tree может быть выбран для оптимизации таких запросов.

Кроме того, B-дерево может применяться при выполнении запросов с операторами LIKE и ILIKE, которые используются для поиска по шаблону. Однако оптимизация с их помощью возможна только в случаях, когда шаблон начинается с фиксированного префикса, не подверженного изменениям из-за преобразования регистра букв.

Также, индексы B-tree могут использоваться для извлечения данных в сортированном порядке. Хотя это не всегда быстрее, чем выполнение полного сканирования с последующей сортировкой, во многих случаях использование индексов может значительно повысить производительность запросов за счёт уменьшения количества необходимых операций сравнения и перемещения данных.

5.3. HASH-индексы

HASH-индексы (хеш-индексы) в PostgreSQL создаются путём преобразования значений индексируемого столбца в 32-битные хеш-коды. Эта особенность делает их эффективными для обработки запросов, которые включают только проверку на равенство. Когда индексируемый столбец участвует в сравнении, использующем оператор равенства (=), планировщик запросов может определить хеш-индекс как подходящий для использования.

Ограничение хеш-индексов состоит в их невозможности обрабатывать диапазонные запросы или сравнения на больше/меньше, что делает их менее универсальными по сравнению с B-деревьями. Тем не менее в случаях, когда требуются исключительно проверки равенства, хеш-индексы могут предложить высокую производительность за счет более быстрого доступа к данным.

5.4. Индексы типа GiST и SP-GiST

Два вида индексов предлагают инфраструктуру, поддерживающую различные виды поиска. В качестве примера можно привести стандартный дистрибутив PostgreSQL, включающий классы операторов GiST для нескольких двумерных геометрических данных, которые поддерживают индексированные запросы с использованием таких операторов: <<, &<, &>, >>, <<|, &<|, |&>, |>>, @>, <@, ~=, &&

Индексы GiST также способны оптимизировать поиск «ближайших соседей», например:

```
SELECT * FROM places
ORDER BY location <-> point '(101,456)' LIMIT  10;
```

который находит десять мест, ближайших к заданной целевой точке. Возможность выполнения этой задачи зависит и от конкретного класса используемых операторов.

SP-GiST позволяет реализовать широкий спектр различных несбалансированных дисковых структур данных, таких как квадтри, k-d-деревья и радиксные деревья (radix tree). В качестве примера можно привести стандартный дистрибутив операторов SP-GiST для двумерных точек, которые поддерживают индексированные запросы с использованием этих операторов: <<, >>, ~=, <@, <<|, |>>, как и GiST, SP-GiST поддерживают поиск «ближайших соседей».

5.5. Индексы GIN и BRIN

Индексы GIN (Generalized Inverted Indexes) являются инвертированными индексами, идеально подходящими для работы с данными, которые включают множественные значения в одном поле, такие как массивы или JSON-объекты. Инвертированный индекс создаёт уникальную запись для каждого элемента данных, позволяя быстро находить строки по содержащимся в них значениям. Это делает индексы GIN полезными для выполнения запросов, которые проверяют наличие определённых элементов в данных.

Индексы BRIN (Block Range Indexes) предназначены для эффективного хранения информации о значении данных в больших диапазонах физических блоков таблицы. Они лучше всего подходят для столбцов, значения в которых изменяются предсказуемо в соответствии с физическим расположением строк в таблице, например, временные метки или последовательные идентификаторы. BRIN-индексы значительно уменьшают объём хранимых данных за счёт суммирования информации по блокам, что делает их наиболее подходящими для таблиц больших размеров.

Как и другие типы индексов, GiST, SP-GiST и GIN, BRIN поддерживают различные стратегии индексации, адаптируясь к специфике хранимых данных. В случаях, когда данные обладают линейным порядком, BRIN-индексы эффективно отражают минимальные и максимальные значения каждого диапазона блоков, ускоряя поиск по соответствующим столбцам.

5.6. Многостолбцовые и уникальные индексы

Индекс может быть определен более чем для одного столбца таблицы. Например, возьмём таблицу с фильмами:

```
CREATE TABLE films (
id SERIAL,
year_relese INT,
age_restrict INT,
name VARCHAR
);
```

Если вы часто выдаете запросы типа:

```
SELECT name FROM films
ON year_relese =123 AND age_restrict=12;
```

тогда уместно определить индекс на этих столбцах вместе, например:

```
CREATE INDEX films_rr_index
ON films (year_relese, age_restrict);
```

Индексы также используются для обеспечения уникальности значения столбца или уникальности объединенных значений более чем одного столбца.

```
CRETE UNIGUE INDEX name
ON table (column [, …]) [NULLS [NOT] DISTINCT];
```

Когда индекс объявлен уникальным, это означает, что в таблице не могут существовать две строки с идентичными значениями в индексированных столбцах. В контексте уникальных индексов PostgreSQL по умолчанию не считает NULL-значения равными между собой, что позволяет иметь несколько строк с NULL в уникальном столбце.

Использование параметра NULLS NOT DISTINCT в определении уникального индекса изменяет эту логику, обязывая индекс рассматривать NULL-значения как равные друг другу. Это ограничивает возможность иметь более одного NULL-значения в уникальном индексе.

В случае многостолбцового уникального индекса ограничение на уникальность применяется ко всему набору индексируемых столбцов. Таким образом, добавление новой строки будет отклонено только в случае, если значения во всех индексируемых столбцах идентичны значениям в уже существующей строке таблицы.

5.7. Создание представления

В базах данных представления являются виртуальными таблицами, созданными на основе запроса к одной или нескольким реальным таблицам. Они обеспечивают абстракцию данных, предоставляя удобный способ для выполнения запросов и сокрытия сложной структуры данных.

Предположим, что комбинированный список записей о фильмах и сеансах представляет особый интерес для приложения, но каждый раз набирать запрос мы не хотим. В таком случае необходимо создать представление на запрос, которое даст запросу имя, на него можно будет ссылаться как на обычную таблицу:

```
CREATE VIEW film_session_view AS
SELECT name, age_restriction, date_session, price
FROM films, sessions
WHERE films.id = sessions.id_film;
SELECT * FROM film_session_view;
```

Свободное использование представлений является ключевым аспектом правильного проектирования базы данных. Представления позволяют инкапсулировать (скрывать) детали структуры таблицы, которые могут меняться по мере развития приложения.

Представления можно использовать практически везде, где необходимо создать реальную таблицу. Построение представлений на основе других представлений не редкость.

Простое представление основано на одной таблице или другом представлении. Оно может быть использовано для фильтрации или переименования столбцов.

```
CREATE VIEW simple_view AS
SELECT column1, column2
FROM table
WHERE condition;
```

Составное представление основано на нескольких таблицах или представлениях, объединенных в один запрос:

```
CREATE VIEW complex_view AS
SELECT t1.column1, t2.column2
FROM table1 t1
JOIN table2 t2 ON t1.id = t2.id;
```

5.8. Сочетание запросов

Для комбинирования результатов двух или более запросов SQL предлагает операции объединения (UNION), пересечения (INTERSECT) и разности множеств (EXCEPT), каждая из которых имеет свои особенности:

UNION объединяет результаты двух запросов в один набор, автоматически исключая дубликаты, если только не указано UNION ALL.

Это значит, что результат будет содержать уникальные строки, присутствующие в обоих запросах, без гарантии порядка их следования.

```
SELECT column1 FROM table1
UNION
SELECT column1 FROM table2;
```

INTERSECT возвращает только те строки, которые присутствуют в результатах обоих запросов, также исключая дубликаты, если не использовано INTERSECT ALL. Это позволяет найти общие данные для двух запросов.

```
SELECT column1 FROM table1
INTERSECT
SELECT column1 FROM table2;
```

EXCEPT возвращает строки из первого запроса, которые не найдены в результате второго запроса, эффективно вычисляя разницу между двумя наборами данных. Как и в предыдущих случаях, дубликаты будут исключены, если не применён EXCEPT ALL.

```
SELECT column1 FROM table1
EXCEPT
SELECT column1 FROM table2.
```

Для того чтобы эти операции могли быть применены, запросы должны быть совместимы с объединением. Это значит, что каждый запрос должен возвращать одинаковое количество столбцов, а типы данных соответствующих столбцов должны быть совместимы, что позволяет корректно объединять, пересекать или вычитать результаты.

5.9. Создание транзакций

Например, рассмотрим базу данных кинотеатра. Предположим, что мы хотим добавить посетителя и зарезервировать для него билет. Нам нужна гарантия того, если что-то пойдёт не так, ни один из шагов, выполненных до этого, не будет выполнен.

В PostgreSQL транзакция создается путем окружения команд BEGIN и COMMIT. Таким образом, наша транзакция будет выглядеть так:

```
BEGIN;
--Добавление нового пользователя
INSERT INTO visitors (name, surname, b_date, email, phone_num)
VALUES ('Мария', 'Иванова', '1999-01-01', 'mary@example.com',
'123456789');
--Получение ID добавленного пользователя
DECLARE new_visitor_id INTEGER;
SELECT id INTO new_visitor_id FROM visitors
WHERE email = 'mary@example.com';
--Резервирование билета для нового пользователя
INSERT INTO reservations (id_visitors, date_reservation, timr_reser-
vation)
VALUES (new_visitor_id, CURRENT_DATE, CURRENT_TIME);
COMMIT;
```

Если в какой-то момент во время транзакции вы решите, что не хотите применять произведённые изменения, можно использовать команду ROLLBACK вместо COMMIT. Это приведёт к отмене всех операций, выполненных в рамках транзакции, и возврату базы данных к состоянию, которое было до начала транзакции.

5.10. Преимущества транзакций

Группировка действий в транзакции дает гарантию, если что-то пойдет не так, то ни один из шагов не выполнится. Считается, что транзакция является атомарной, когда она либо выполняется полностью, либо не выполняется вообще.

Транзакция гарантирует, что все сделанные ей обновления будут записаны в постоянное хранилище до того, как транзакция завершится.

Это свойство транзакции известно как изоляция и является одним из четырёх основных принципов транзакций в базах данных, обозначаемых аббревиатурой ACID (Атомарность, Согласованность, Изоляция, Долговечность). Изоляция гарантирует, что параллельно выполняющиеся транзакции не влияют друг на друга таким образом, чтобы одна транзакция видела незавершённые изменения, сделанные другой транзакцией.

Изоляция обеспечивает, что каждая транзакция работает в изолированном окружении и видит состояние базы данных либо до начала других параллельных транзакций, либо после их завершения, но не в промежуточном состоянии. Это предотвращает множество потенциальных проблем, включая потерянные обновления, «грязное чтение», неповторяющееся чтение и фантомное чтение.

5.11. Точки сохранения

Точки сохранения предоставляют возможность детального контроля над операциями в рамках транзакции, давая шанс отменить части транзакции при сохранении остальных изменений. При использовании команды SAVEPOINT можно задать маркер в транзакции, к которому возможен возврат с помощью ROLLBACK TO, если это станет необходимым.

Изменения, внесённые в базу данных после создания точки сохранения и до выполнения отката к ней, будут отменены, в то время как все действия, произошедшие до этого момента, останутся неизменными. Интересный момент заключается в том, что после отката к точке сохранения она не исчезает и может быть использована повторно, если потребуется вернуться к этому же состоянию снова.

Если вы определите, что возврат к конкретной точке сохранения больше не потребуется, можно освободить её с помощью соответствующей команды, что позволит системе освободить занятые ею ресурсы. Важно помнить, что освобождение или откат к точке сохранения автоматически приведёт к освобождению всех точек сохранения, установленных после неё, что помогает поддерживать порядок и оптимизировать использование ресурсов в рамках транзакции.

Рассмотрим точки сохранения на примере нашей транзакции:

```
BEGIN;
INSERT INTO visitors (name, surname, b_date, email, phone_num)
VALUES ('Мария', 'Иванова', '1999-01-01', 'mary@example.com',
'123456789');
SAVEPOINT step1;
DECLARE new_visitor_id INTEGER;
```

```
SELECT id INTO new_visitor_id FROM visitors
WHERE email = 'mary@example.com';
SAVEPOINT step2;
INSERT INTO reservations (id_visitors, date_reservation, timr_reservation)
VALUES (new_visitor_id, CURRENT_DATE, CURRENT_TIME);
COMMIT;
```

Для отката к первой точке сохранения используется команда:

```
ROLLBACK TO step1;
```

5.12. Привилегии пользователей и создание пользователей

В системах управления базами данных, когда создаётся объект, например, таблица или представление, за ним закрепляется владелец – обычно это роль, которая инициировала создание объекта. В большинстве случаев, сразу после создания объект настроен таким образом, что взаимодействовать с ним могут только его владелец или суперпользователь. Это ограничивает доступ к объекту для остальных пользователей и их ролей в системе.

Для расширения доступа к объекту другим ролям необходимо явно предоставить соответствующие привилегии.

Для создания роли, которая может использоваться для входа в систему базы данных, т.е. обладает атрибутом LOGIN, применяют SQL-команду CREATE ROLE. В PostgreSQL и некоторых других системах управления базами данных такая роль эквивалентна понятию «пользователь базы данных». Чтобы создать роль с привилегией входа в систему, используйте:

```
CREATE ROLE name LOGIN;
CREATE USER name;
```

CREATE USER эквивалентен CREATE ROLE, за исключением того, что CREATE USER включает LOGIN по умолчанию, а CREATE ROLE – нет.

Создадим пользователя с логином и паролем:

```
CREATE USER administrator WITH PASSWORD 'qwerty123';
```

Можно создать пользователя, который будет действителен до какого-то определенного срока, например, по истечении одной секунды после окончания 2026 года пароль перестанет действовать:

```
CREATE ROLE user WITH LOGIN PASSWORD 'password'
VALID UNTIL '2025-01-01';
```

5.13. Виды привилегий

Привилегии в системах управления базами данных определяют, какие действия могут выполнять пользователи над различными объектами, такими как таблицы, функции, представления и др. Рассмотрим подробнее привилегии (табл. 4).

SELECT: эта привилегия позволяет пользователю читать данные из таблиц, представлений и других объектов, являющихся коллекциями данных. Если привилегия SELECT выдана на таблицу, пользователь может использовать операцию SELECT для извлечения данных из всех или указанных столбцов таблицы. Это также позволяет выполнение команды COPY TO для экспорта данных. Для операций обновления (UPDATE), удаления (DELETE) или слияния (MERGE), где необходимо ссылаться на существующие значения столбцов, также требуется привилегия SELECT. Для последовательностей (sequences) эта привилегия дает доступ к функции currval, а для больших объектов – возможность их чтения.

INSERT дает возможность вставки новых строк в таблицу или представление. Если привилегия INSERT предоставлена для конкретного столбца или группы столбцов, то в эти столбцы можно вставлять данные, в то время как остальные столбцы будут заполнены значениями по умолчанию или значениями, определенными в триггерах или правилах. Эта привилегия также разрешает использование команды COPY FROM для импорта данных в таблицу.

DELETE позволяет удалить строку из таблицы, представления и т. д.

CREATE позволяет создавать новые схемы и публикации в базе данных, а также устанавливать в БД доверенные расширения.

Таблица 4. Виды привилегий

Привилегия	Применимые типы объектов
SELECT	Большие объекты, таблица и таблицеподобные объекты, столбец таблицы
INSERT	Таблица, столбец таблицы
UPDATE	Большие объекты, таблица и таблицеподобные объекты, столбец таблицы
DELETE	Таблица
TRUNCATE	Таблица
REFERENCES	Таблица, столбец таблицы
TRIGGER	Таблица
CREATE	База данных, схема, пространство таблиц
CONNECT	База данных
TEMPORARY	База данных
EXECUTE	Функция, процедура
USAGE	Домен, иностранный сервер, язык, схема, тип
SET	Параметр
ALTER SYSTEM	Параметр

5.14. Создание привилегий для пользователя

В PostgreSQL для управления правами доступа к объектам базы данных, таким как таблицы, представления и схемы, используются привилегии. Привилегии предоставляют или ограничивают доступ к определенным операциям.

Предоставим созданному пользователю SELECT-привилегии на все таблицы:

```
GRANT SELECT ON ALL TABLES IN SCHEMA cinema
TO administrator;
```

Для отмены раннее предоставленных прав используется команда REVOKE. Например, отменим все привилегии на схему базы данных:

```
REVOKE USAGE ON CHEMA cinema FROM administrator;
```

Если будет необходимо изменить уже существующие привилегии, можно сделать это следующим образом:

```
REVOKE SELECT ON TABLE films FROM administrator;
GRANT INSERT ON TABLE sessions FROM administrator;
```

Разрешается предоставлять привилегии суперпользователя, однако необходимо это делать с осторожностью, так как будет предоставлен полный контроль над базой данных:

```
ALTER USER administrator WITH SUPERUSER;
```

Суперпользователя можно переключить на обычного пользователя. Заметьте, что для выполнения этих операций вам могут потребоваться соответствующие привилегии администратора. Будьте внимательны при изменении или отзыве привилегий, чтобы не нарушить корректное функционирование базы данных:

```
ALTER USER administrator WITH NOSUPERUSER;
```

Контрольные вопросы и задания

1. Чем создание пользователей отличается от создания ролей в PostgreSQL? Опишите, как происходит управление привилегиями пользователей.

2. Создайте несколько пользователей и присвойте им разные привилегии на SELECT, DELETE, INSERT.

3. Создайте пользователя и передайте ему привилегии суперпользователя, а затем верните его до уровня обычного пользователя.

4. Объясните, что такое индексы. Как они работают и какие проблемы помогают решить?

5. Почему так важны транзакции, какие свойства они имеют?

6. Создайте индексы для таблиц вашей базы данных и сравните время выполнения запросов до и после их создания.

7. Создайте представление, которое сочетает данные из нескольких таблиц.

8. Напишите пример использования транзакций в PostgreSQL.

Глава 6. ФУНКЦИИ, ПРОЦЕДУРЫ, ТРИГГЕРЫ И КУРСОРЫ

В PostgreSQL функции и процедуры предоставляют мощные инструменты для создания пользовательских операций и логики, которые могут быть повторно использованы в различных частях приложения или запросов к базе данных.

PostgreSQL позволяет вызывать функции, имеющие именованные параметры, используя позиционную или именованную нотацию, а также поддерживает смешанную нотацию, которая сочетает в себе позиционную и именованную нотации. В этом случае позиционные параметры записывают первыми, а именованные параметры – после них.

Процедура – это объект базы данных, аналогичный функции. Основные отличия – процедура определяется с помощью команды CREATE PROCEDURE, а не CREATE FUNCTION.

Процедуры не возвращают значение функции, поэтому в CREATE PROCEDURE отсутствует предложение RETURNS. Однако процедуры могут возвращать данные вызывающим их пользователям через выходные параметры. В то время как функцию вызывают как часть запроса или команды DML, процедуру – изолированно с помощью команды CALL.

В совокупности функции и процедуры также называются рутинами. Существуют такие команды, как ALTER ROUTINE и DROP ROUTINE, которые могут работать с функциями и процедурами без уточнения, к какому типу они относятся.

Функции и процедуры в PostgreSQL могут быть написаны на различных языках, таких как PL/pgSQL, SQL, PL/Tcl, PL/Perl и др. В зависимости от сложности и требований к коду можно выбрать наиболее подходящий язык для написания функций и процедур.

Триггеры в SQL – это специальные объекты, которые автоматически выполняются при определенных событиях в базе данных, таких как вставка, обновление или удаление данных в определенной таблице. Триггеры позволяют вам автоматизировать выполнение определенных задач или применять бизнес-правила к данным при их изменении.

Вместо того чтобы выполнять весь запрос за один раз, можно установить курсор, который инкапсулирует запрос, а затем считывать результат запроса по несколько строк за раз. Это необходимо для того, чтобы избежать перерасхода памяти, когда результат содержит большое количество строк. Более интересный вариант – возврат ссылки на курсор, который создала функция, позволяя вызывающей стороне читать строки. Это обеспечивает эффективный способ возврата больших наборов строк из функций.

6.1. Создание функции

CREATE FUNCTION определяет новую функцию. CREATE OR REPLACE FUNCTION либо создает новую функцию, либо заменяет существующее определение.

Если указано имя схемы, то функция создается в указанной схеме, в противном случае – в текущей схеме. Имя новой функции не должно совпадать ни с одной существующей функцией или процедурой с теми же типами входных аргументов в той же схеме. Однако функции и процедуры с разными типами аргументов могут иметь общее имя (это называется перегрузкой).

Посчитаем общее количество клиентов в базе данных:

```
CREATE OR REPLACE FUNCTION get_visitors_count()
RETURNS INTEGER AS $$
DECLARE visitor_count INTEGER;
BEGIN
SELECT COUNT(*) INTO visitor_count FROM visitors;
RETURN visitor_count;
END;
$$ LANGUAGE PLPGSQL;
```

В этом примере создается функция get_visitors_count, которая возвращает общее количество посетителей в таблице зрителей.

Пример вызова функции:

```
SELECT get_visitors_count();
```

6.2. Создание процедуры

Например, создадим процедуру update_visitors_phone, которая принимает идентификатор посетителя и новый телефон, затем обновляет запись о посетителе в таблице зрителей:

```
CREATE OR REPLACE PROCEDURE
update_visitors_phone (visitors_id INT, new_phone VARCHAR)
LANGUAGE PLPGSQL AS $$
BEGIN
UPDATE visitors SET phone_namber = new_phone
WHERE id = visitor_id;
END;
$$;
```

Пример вызова процедуры:

```
CALL update_visitors_phone (1,’123456789’);
```

6.3. Создание триггеров

Пример создания триггера в SQL:

```
CREATE TRIGGER VisitorUpdate
ON visitors
AFTER UPDATE
AS
BEGIN
INSERT INTO Visitorlog (TableName, RecordID, Action)
SELECT 'visitors', visitorsID, 'Update'
FROM inserted
END
```

Код создает триггер под названием VisitorUpdate, который срабатывает после обновления записей в таблице зрителей, вставляет соответствующие данные в таблицу Visitorlog для отслеживания изменений.

6.4. Объявление переменных курсора

Вместо того чтобы выполнять весь запрос за один раз, можно установить курсор, который инкапсулирует запрос, а затем считывать результат запроса по несколько строк за раз. Это необходимо для устранения перерасхода памяти, когда результат содержит большое количество строк.

Весь доступ к курсорам осуществляется через переменные курсора, которые всегда имеют специальный тип данных refcursor. Один из способов создать курсорную переменную – просто объявить ее как переменную типа refcursor, другой способ – использовать синтаксис объявления курсора, который в общем случае выглядит следующим образом:

```
name [ [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

Если указано SCROLL, курсор будет способен прокручивать курсор назад; если указано NO SCROLL, поиск в обратном направлении будет отклонен; если не указаны ни одно из этих условий, ни одна из спецификаций, разрешение на обратную прокрутку будет зависеть от запроса.

Примеры:

```
DECLARE
curs1 refcursor;
curs2 CURSOR FOR SELECT * FROM tenk1;
curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WERE
unique1 = key;
```

Все три переменные имеют тип данных refcursor, но первая может быть использована с любым запросом, в то время как ко второй уже привязан полностью определенный запрос, а к последней – параметризованный запрос.

6.5. Открытие курсоров

Прежде чем курсор можно будет использовать для получения строк, его необходимо открыть. Это действие эквивалентно SQL-команде DECLARE CURSOR.

Связанные переменные курсора можно использовать и без явного открытия курсора – с помощью оператора FOR. Цикл FOR открывает курсор, а затем снова закрывает его, когда цикл завершится. Открытие курсора включает создание внутренней структуры данных сервера, называемой порталом, который содержит состояние выполнения запроса курсора. Портал имеет имя, оно должно быть уникальным в пределах сессии в течение всего времени существования.

Пример:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

6.6. Курсоры для чтения

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

FETCH извлекает данные для цели, которая может быть переменной строки, переменной записи или списком простых переменных, разделенных запятыми, как и в SELECT INTO.

Пример использования курсора:

```
FETCH NEXT FROM your_cursor INTO variable1, variable2;
-- Продолжение чтения до конца
WHILE FOUND LOOP
-- Ваш код для обработки переменных
-- Получение следующей строки
FETCH NEXT FROM your_cursor INTO variable1, variable2;
END LOOP;
```

В этом примере FETCH NEXT используется для получения следующей строки из курсора. WHILE FOUND обеспечивает выполнение цикла до тех пор, пока есть данные для чтения.

После обработки всех данных курсор следует закрыть:

```
CLOSE your_cursor;
```

Контрольные вопросы и задания

1. Объясните в чем заключаются различия между функциями, процедурами и триггерами.
2. Напишите примеры реализаций функций, процедур, триггеров.
3. Какова роль курсоров и чем они отличаются от обычных запросов?
4. Создайте 2 − 3 курсора для вашей базы данных.

Глава 7. ПЕРЕЧИСЛЯЕМЫЕ ТИПЫ И JSON

Перечисляемые типы (enum) – это типы данных, состоящие из статического, упорядоченного набора значений. Перечисления эквивалентны типам enum, поддерживаемым в ряде языков программирования. Примером перечислительного могут быть дни недели или набор значений состояния для части данных [7].

Типы данных JSON предназначены для хранения данных в формате JSON (JavaScript Object Notation). Такие данные можно хранить и в виде текста, но преимущество типов данных JSON заключается в том, что каждое хранимое значение является действительным в соответствии с правилами JSON.

7.1. Объявление перечисляемых типов

Перечисляемые типы создаются с помощью команды CREATE TYPE, например:

```
CREATE TYPE genre AS ENUM ('драма', 'комедия', 'фантастика');
```

После создания тип enum можно использовать в определениях таблиц и функций, как и любой другой тип:

```
CREATE TABLE films (
 name VARCHAR,
 genres GANRE
);
INSERT INTO films VALUES ('Тренер', 'драма');
SELECT * FROM films WHERE genres = 'драма';
```

7.2. Упорядочивание

Порядок значений в типе перечисления – это порядок, в котором значения были перечислены при создании типа. Для перечислений поддерживаются все стандартные операторы сравнения и связанные с ними агрегатные функции. Например:

```
INSERT INTO films VALUES ('Один дома', 'комедия');
INSERT INTO films VALUES ('Аватар', 'фантастика');
SELECT * FROM films WHERE genres > 'комедия';
```

7.3. Детали реализации

Метки Enum чувствительны к регистру, поэтому 'happy' не то же самое, что 'HAPPY'. Пробелы в метках также имеют значение, хотя типы перечислений в основном предназначены для статических наборов значений. Существуют поддержка добавления новых значений к типу перечисления, а также переименование значений.

Значения не могут быть удалены из перечисляемого типа, равно как и не может быть изменен порядок сортировки таких значений, если только они не будут удалены и повторно созданы.

Значение перечисления занимает четыре байта на диске. Длина текстовой метки значения перечисления ограничивается параметром NAMEDATALEN, в стандартных сборках это означает не более 63 байт.

Переводы внутренних значений перечислений в текстовые метки хранятся в системном каталоге pg_enum.

7.4. Работа с JSON

Существуют различные функции и операторы, специфичные для JSON. PostgreSQL предлагает два типа для хранения данных JSON: json и jsonb. Чтобы реализовать эффективные механизмы запросов к типам данных PostgreSQL также предоставляет тип данных jsonpath.

Типы данных json и jsonb принимают на вход практически одинаковые наборы значений, отличие заключается в эффективности. Тип данных json хранит точную копию входного текста, которую функции обработки должны повторно анализировать при каждом выполнении; в то время как данные jsonb хранятся в разложенном двоичном формате, что замедляет его работу при вводе данных из-за дополнительных расходов на их преобразование. Скорость действия jsonb значительно быстрее при обработке данных, поскольку не требуется повторная обработка; jsonb также поддерживает индексирование, что является значительным преимуществом.

7.5. Генерация столбцов из JSON

Предположим, есть столбец с JSON-данными в таблице и необходимо извлечь значения из этого JSON в виде отдельных столбцов. В PostgreSQL можно использовать оператор `->>` для извлечения значения из JSON. Пример:

```
CREATE TABLE my_table (
id serial PRIMARY KEY,
json_data json
);
-- Вставка данных с JSON
INSERT INTO my_table (json_data) VALUES
('{"name": "John", "age": 25, "city": "New York"}'),
('{"name": "Alice", "age": 30, "city": "San Francisco"}');
-- Извлечение значений из JSON
SELECT
id,
json_data ->> 'name' AS name,
(json_data ->> 'age')::int AS age,
json_data ->> 'city' AS city
FROM
my_table;
```

В примере:

- `json_data ->> 'name'` извлекает значение по ключу "name" из JSON.
- `(json_data ->> 'age')::int` извлекает значение по ключу "age" и преобразует его в целое число.
- `json_data ->> 'city'` извлекает значение по ключу "city".

7.6. Генерация JSON из столбцов

Можно сгенерировать JSON из отдельных столбцов с использованием функций `ROW_TO_JSON` или `JSON_BUILD_OBJECT`.

Пример:

```
SELECT
id,
ROW_TO_JSON((id, name, age, city)::my_table) AS json_data
FROM my_table;
```

В примере `ROW_TO_JSON` используется для генерации JSON из столбцов, также можно применять `JSON_BUILD_OBJECT` для аналогичной задачи.

Примеры демонстрируют основные концепции работы с JSON в PostgreSQL. Работа с JSON может быть более сложной, если данные содержат вложенные объекты или массивы, и в таких случаях могут потребоваться более сложные запросы или функции.

Контрольные вопросы и задания

1. Почему стоит использовать JSON в базах данных? Перечислите особенности использования данных в таком формате.

2. Создайте пример хранения и выборки данных в формате JSON для вашей базы данных.

3. Каковы преимущества использования перечисляемых типов в базах данных?

4. Создайте перечисляемый тип и напишите пример с его использованием.

ЗАКЛЮЧЕНИЕ

В учебном пособии были рассмотрены основные вопросы, связанные с управлением данными в СУБД PostgreSQL. В результате изучения материала у обучающегося вырабатываются единые представления об информационных системах, которые ориентированы на обработку и хранение данных. При этом многие из рассмотренных вопросов позволяют абстрагироваться от уровня выбранной в пособии СУБД.

Впоследствии изучения и практики накопленные знания позволят достаточно грамотно подходить к вопросам выбора структур хранения данных, и, что более важно, научиться пользоваться языком структурированных запросов SQL.

Практические навыки, приобретенные при работе с системой управления базами данных PostgreSQL, можно закрепить при решении задач, представленных в пособии.

Полученные знания могут быть использованы и в других специальных дисциплинах, где возникает необходимость организовать БД и построить работу с ней.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Агальцов, В. П. Базы данных. Распределенные и удаленные базы данных : учебник. В 2 кн. Кн. 2 / В. П. Агальцов. – М. : ФОРУМ : Инфра, 2024. – 271 с. – ISBN 978-5-8199-0959-1.

2. Новиков, Б. Основы технологий баз данных : учеб. пособие / Б. Новиков, Е. Горшкова, Н. Графеева. – 2-е изд. – М. : ДМК-Пресс, 2020. – 582 с. – ISBN 978-5-97060-841-8.

3. Рогов, Е. PostgreSQL изнутри / Е. Рогов. – М. : ДМК-Пресс, 2024. – 664 с. – ISBN 978-5-93700-305-8.

4. Кузнецов, С. Д. Базы данных. Модели и языки / С. Д. Кузнецов. – М. : Бином, 2008. – 720 с. – ISBN 978-5-9518-0132-6.

5. Лузанов, П. Postgres: первое знакомство / П. Лузанов, Е. Рогов, И. Лёвшин. – 9-е изд. – М. : ППГ, 2023. – 179 с. – ISBN 978-5-6045970-1-9.

6. Джуба, С. Изучаем PostgreSQL 10 / С. Джуба, А. Волков. – 2-е изд. – М. : ДМК-Пресс, 2019. – 400 с. – ISBN 978-5-97060-643-8.

7. Моргунов, Е. П. PostgreSQL. Основы языка SQL : учеб. пособие / Е. П. Моргунов ; под ред. Е. В. Рогова, П. В. Лузанова. – СПб. : БХВ-Петербург, 2018. – 336 с. – ISBN 978-5-9775-4022-3.

РЕКОМЕНДАТЕЛЬНЫЙ БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Домбровская, Г. Оптимизация запросов PostgreSQL / Г. Домбровская, Б. Новиков, А. Бейликова. – М. : ДМК-Пресс, 2021. – 278 с. − ISBN 978-5-97060-963-7.

2. Дейт, К. Дж. Введение в системы баз данных. В 2 т. Т. 1 / К. Дж. Дейт. – 8-е изд. – М. : Вильямс, 2012. – 768 с. − ISBN 978-5-8459-1697-1.

3. Момиджанов, О. PostgreSQL 13. Подробное руководство / О. Момиджанов, Г. Момиджанов. – М. : ДМК-Пресс, 2021. – 672 с. − ISBN 978-5-97060-823-4.

4. Риггс, Саймон. PostgreSQL 12. Выжми максимум / Саймон Риггс, Ханс-Юрген Шёнйич. – М. : ДМК-Пресс, 2020. – 720 с. − ISBN 978-5-97060-713-8.

5. Керниган, Б. У. Язык программирования SQL / Б. У. Керниган, Р. Ф. Пайк – 2-е изд. – М. : Вильямс, 2018. – 240 с. − ISBN 978-5-8459-2011-4.

6. Гарсиа-Молина, Э. Системы баз данных. Полный курс / Э. Гарсиа-Молина, Дж. Ульман, Дж. Уидом. – 2-е изд. – М. : Вильямс, 2016. – 1152 с. − ISBN 978-5-8459-1888-3.

Учебное электронное издание

ОСНОВЫ ПРОЕКТИРОВАНИЯ БАЗЫ ДАННЫХ НА POSTGRESQL

Учебное пособие

Автор-составитель
КУРЬЕРОВА Софья Алексеевна

Редактор А. А. Амирсейидова
Верстка Л. В. Макаровой
Рецензент А. И. Петрова
Выпускающий редактор Е. А. Лебедева

Системные требования: Intel от 1,3 ГГц; Windows XP/7/8/10;
Adobe Reader; дисковод CD-ROM.

Тираж 9 экз.

Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.