POSTGRES

П. Лузанов, Е. Рогов, И. Лёвшин

ПЕРВОЕ ЗНАКОМСТВО

15

Предисловие

Эту небольшую книгу мы написали для тех, кто только начинает знакомиться с PostgreSQL. Из нее вы узнаете:

Мы надеемся, что наша книга сделает ваш первый опыт работы с PostgreSQL приятным и поможет влиться в сообщество пользователей этой СУБД. Желаем удачи!

I О PostgreSQL

PostgreSQL — наиболее полнофункциональная свободно распространяемая СУБД с открытым кодом. Разработанная в академической среде, за долгую историю сплотившая вокруг себя широкое сообщество разработчиков, эта СУБД обладает всеми возможностями, необходимыми большинству заказчиков. PostgreSQL активно применяется по всему миру для создания критичных бизнес-систем, работающих под большой нагрузкой.

Немного истории

Современный PostgreSQL ведет происхождение от проекта POSTGRES, который разрабатывался под руководством Майкла Стоунбрейкера (Michael Stonebraker), профессора Калифорнийского университета в Беркли. До этого Стоунбрейкер возглавлял разработку INGRES — одной из первых реляционных СУБД, — и POSTGRES возник как результат осмысления предыдущей работы и желания преодолеть ограниченность жесткой системы типов.

Работа над проектом началась в 1985 году, и до 1988 года был опубликован ряд научных статей, описывающих модель данных, язык запросов POSTQUEL (в то время SQL еще не был общепризнанным стандартом) и устройство хранилища данных.

POSTGRES иногда относят к так называемым постреляционным СУБД. Ограниченность реляционной модели всегда была предметом критики, хотя и являлась обратной стороной ее простоты и строгости. Однако проникновение компьютерных технологий во все сферы жизни привело к появлению новых классов приложений и потребовало от баз данных поддержки нестандартных типов данных и таких возможностей, как наследование, создание сложных объектов и управление ими.

Первая версия СУБД была выпущена в 1989 году. База данных совершенствовалась на протяжении нескольких лет, а в 1993 году, когда вышла версия 4.2, проект был закрыт. Но, несмотря на официальное прекращение, открытый код и BSD-лицензия позволили выпускникам Беркли Эндрю Ю и Джоли Чену в 1994 году взяться за его дальнейшее развитие. Они заменили язык запросов POSTQUEL на ставший к тому времени общепринятым SQL, а проект нарекли Postgres95.

К 1996 году стало ясно, что название Postgres95 не выдержит испытания временем, и было выбрано новое имя — PostgreSQL, которое отражает связь и с оригинальным проектом POSTGRES, и с переходом на SQL. Надо признать, что название получилось сложновыговариваемым, но тем не менее: PostgreSQL следует произносить как «постгрес-куэль» или просто «постгрес», но только не «постгре».

Новая версия стартовала как 6.0, продолжая исходную нумерацию. Проект вырос, и управление им взяла на себя поначалу небольшая группа инициативных пользователей и разработчиков, которая получила название Глобальной группы разработки PostgreSQL (PostgreSQL Global Development Group).

Развитие

Все основные решения о планах развития и выпусках новых версий принимаются Управляющим комитетом (Core team), состоящим сейчас из семи человек.

Помимо обычных разработчиков, вносящих посильную лепту в развитие системы, выделяется группа основных разработчиков (major contributors), сделавших существенный вклад в развитие PostgreSQL, а также группа разработчиков, имеющих право записи в репозиторий исходного кода (committers). Состав групп со временем меняется, появляются новые члены, кто-то отходит от проекта. Актуальный список разработчиков публикуется на официальном сайте: postgresql.org/community/contributors.

Вклад российских разработчиков в PostgreSQL весьма значителен. Это, пожалуй, крупнейший глобальный проект с открытым исходным кодом из всех, в которых настолько широко представлена Россия.

Большую роль в становлении и развитии PostgreSQL сыграл программист из Красноярска Вадим Михеев, входивший в Управляющий комитет. Он является автором таких важнейших частей системы, как многоверсионное управление одновременным доступом (MVCC), система очистки (vacuum), журнал транзакций (WAL), вложенные запросы, триггеры. Сейчас Вадим уже не занимается проектом.

В 2015 году Олег Бартунов, астроном и научный сотрудник ГАИШ МГУ, совместно с Федором Сигаевым и Александром Коротковым основал компанию Postgres Professional как кузницу квалифицированных кадров в области разработки систем баз данных и место создания отечественной СУБД.

Среди направлений выполненных ими работ можно выделить локализацию PostgreSQL (поддержка национальных кодировок и Unicode), систему полнотекстового поиска, работу с массивами и слабоструктурированными данными (hstore, json, jsonb), новые методы индексации (GiST, SP-GiST, GIN и RUM, Bloom). Также они являются авторами большого числа популярных расширений.

Цикл работы над очередной версией PostgreSQL обычно занимает около года. За это время от всех желающих принимаются на рассмотрение патчи с исправлениями, изменениями и новым функционалом. Для обсуждения патчей по традиции используется список рассылки pgsql-hackers. Если сообщество признает идею полезной, ее реализацию — правильной, а код проходит обязательную проверку другими разработчиками, то патч включается в релиз.

В некоторый момент (обычно весной, примерно за полгода до релиза) объявляется этап стабилизации кода — новый функционал откладывается до следующей версии, а продолжают приниматься только исправления или улучшения уже включенных в релиз патчей. Несколько раз в течение релизного цикла выпускаются бета-версии, ближе к концу цикла появляется релиз-кандидат, а вскоре выходит и новая основная (major) версия PostgreSQL.

Раньше номер основной версии состоял из двух чисел, но начиная с 2017 года было решено оставить только одно. Таким образом, за 9.6 последовала 10, а последней актуальной версией PostgreSQL является версия 15, вышедшая в середине октября 2022 года.

При работе над новой версией СУБД могут обнаруживаться ошибки. Наиболее критические из них исправляются не только в текущей, но и в предыдущих версиях. Обычно раз в квартал выпускаются дополнительные (minor) версии,

включающие накопленные исправления. Например, версия 12.5 содержит только исправления ошибок, найденных в 12.4, а 15.1 — в версии 15.0.

Поддержка

Глобальная группа разработки PostgreSQL выполняет поддержку основных версий системы в течение пяти лет с момента выпуска. Эта поддержка, как и координация разработки, осуществляется через списки рассылки. Корректно оформленное сообщение об ошибке имеет все шансы на скорейшее решение: нередки случаи, когда исправления ошибок выпускаются в течение суток.

Помимо поддержки сообществом разработчиков, ряд компаний по всему миру осуществляет коммерческую поддержку PostgreSQL. В России такой компанией является Postgres Professional (postgrespro.ru), предоставляющая услуги поддержки в режиме 24x7.

Современное состояние

PostgreSQL — одна из самых популярных в мире систем баз данных. По итогам свыше двадцати лет развития на прочном академическом фундаменте она выросла в полноценную СУБД, пригодную для корпоративного использования, и составляет реальную альтернативу коммерческим системам. Чтобы убедиться в этом, достаточно посмотреть на важнейшие характеристики новейшей на сегодняшний день версии PostgreSQL 15.

Надежность и устойчивость

При работе с критически важными данными в корпоративных приложениях особенно важно обеспечить надежность. С этой целью PostgreSQL позволяет настраивать горячее резервирование, восстановление на заданный момент времени в прошлом, различные виды репликации (синхронную, асинхронную, каскадную).

Безопасность

PostgreSQL позволяет пользователям подключаться по защищенному SSL-соединению. Возможна аутентификация по паролю (включая SCRAM), использование клиентских сертификатов и аутентификация с помощью внешних сервисов (LDAP, RADIUS, PAM, Kerberos).

Для управления доступом к объектам баз данных предоставляются следующие возможности:

- создание и управление учетными записями пользователей и групповыми ролями;
- разграничение доступа к объектам БД на уровне как отдельных пользователей, так и групп;
- детальное управление доступом на уровне отдельных столбцов и строк;
- поддержка SELinux через встроенную функциональность SE-PostgreSQL (мандатное управление доступом).

Специальная версия PostgreSQL компании Postgres Professional, Postgres Pro Certified, сертифицирована ФСТЭК для использования в системах обработки конфиденциальной информации и персональных данных.

Соответствие стандартам

PostgreSQL обеспечивает соответствие новым требованиям стандарта ANSI SQL по мере их появления. Это относится ко всем версиям стандарта от SQL-92 до самой последней SQL:2016, стандартизировавшей поддержку работы с форматом JSON. Существенная часть этого функционала уже реализована в PostgreSQL 15.

В целом PostgreSQL обеспечивает высокий уровень соответствия стандарту и поддерживает 170 из 177 обязательных возможностей, а также большое количество необязательных.

Поддержка транзакционности

PostgreSQL обеспечивает полную поддержку свойств ACID и обеспечивает эффективную изоляцию транзакций. Для этого используется механизм многоверсионного управления одновременным доступом (MVCC), который позволяет обходиться без блокировок строк во всех случаях, кроме одновременного изменения одной и той же строки данных в нескольких процессах: чтение никогда не блокирует запись, а запись — чтение.

Это же относится и к самому строгому уровню изоляции serializable, который, используя инновационную систему Serializable Snapshot Isolation, обеспечивает полное отсутствие аномалий сериализации и гарантирует совпадение результатов параллельного и последовательного выполнения.

Для разработчиков приложений

Разработчики приложений получают в свое распоряжение богатый инструментарий, позволяющий реализовать приложения любого типа:

- всевозможные языки серверного программирования: встроенный PL/pgSQL (удобный своей тесной интеграцией с SQL), C для критичных по производительности задач, Perl, Python, Tcl, а также JavaScript, Java и другие;
- программные интерфейсы для обращения к СУБД из приложений на любом языке, включая стандартные интерфейсы ODBC и JDBC;
- набор объектов баз данных, позволяющий эффективно реализовать логику любой сложности на стороне сервера: таблицы и индексы, последовательности, ограничения целостности, представления и материализованные представления, секционирование, подзапросы и with-запросы (в том числе рекурсивные), агрегатные и оконные функции, хранимые функции, триггеры и т. д.;
- гибкая система полнотекстового поиска с поддержкой русского и всех европейских языков, дополненная эффективным индексным доступом;
- слабоструктурированные данные, характерные для NoSQL: hstore (хранилище пар «ключ-значение»), xml, json (как в текстовом, так и в более эффективном двоичном представлении jsonb);
- подключение источников данных, включая все основные СУБД, в качестве внешних таблиц по стандарту SQL/MED с возможностью их полноценного использования, в том числе для записи и распределенного выполнения запросов (Foreign Data Wrappers).

Масштабируемость и производительность

PostgreSQL эффективно использует современную архитектуру многоядерных процессоров — производительность СУБД растет практически линейно с увеличением количества ядер.

Предусмотрен параллельный режим выполнения запросов и некоторых служебных команд (таких как создание индексов и очистка). В таком режиме операции чтения данных и соединения выполняются несколькими одновременно работающими процессами. JIT-компиляция запросов повышает возможности ускорения операций аппаратными средствами. Новые возможности распараллеливания появляются в каждой новой версии PostgreSQL.

Для горизонтального масштабирования PostgreSQL предоставляет возможности репликации, как физической, так и логической. Это позволяет строить на базе PostgreSQL кластеры для обеспечения отказоустойчивости, высокой производительности и географической распределенности. Примерами таких систем могут служить Citus (Citusdata), Postgres-BDR (2ndQuadrant), Multimaster (Postgres Professional), Patroni (Zalando).

Планировщик запросов

В PostgreSQL используется стоимостной планировщик запросов. Сбор статистических данных и учет ресурсоемкости как дисковых, так и процессорных операций позволяет оптимизировать даже самые сложные запросы. В распоряжении планировщика находятся все методы доступа к данным и способы выполнения соединений, имеющиеся у передовых коммерческих СУБД.

Возможности индексирования

В PostgreSQL реализованы различные способы индексирования. Помимо традиционных B-деревьев, имеется ряд других методов доступа.

- Hash — индекс, основанный на хешировании. В отличие от B-деревьев, он работает только при проверке на равенство, но в ряде случаев оказывается компактнее и эффективнее.
- GiST — обобщенное сбалансированное дерево поиска, которое применяется для данных, не допускающих упорядочения. Примерами могут служить R-деревья для индексирования точек на плоскости с возможностью быстрого поиска ближайших соседей (k-NN search) и индексирование операции пересечения интервалов.
- SP-GiST — обобщенное несбалансированное дерево, основанное на разбиении области значений на непересекающиеся вложенные области. Примерами могут служить дерево квадрантов для пространственных данных и префиксное дерево для текстовых строк.
- GIN — обобщенный инвертированный индекс, который используется для сложных значений, состоящих из элементов. Основная область его применения — полнотекстовый поиск, то есть поиск документов, в которых встречаются указанные в поисковом запросе слова. Другим примером использования является поиск значений в массивах данных.
- RUM — дальнейшее развитие метода GIN для полнотекстового поиска. Этот индекс, доступный в виде расширения, ускоряет фразовый поиск и сортирует выдачу по релевантности без дополнительных вычислений.

- BRIN — компактная структура, позволяющая найти компромисс между размером индекса и скоростью поиска. Такой индекс эффективен на больших кластеризованных таблицах.
- Bloom — индекс, основанный на фильтре Блума. Благодаря очень компактному представлению позволяет быстро отсечь заведомо ненужные строки, но требует перепроверки оставшихся.

Многие типы индексов могут создаваться не только по одному, но и по нескольким столбцам таблицы. Независимо от типа можно строить индексы как по столбцам, так и по произвольным выражениям, а также создавать частичные индексы только для определенных строк. Покрывающие индексы позволяют ускорить выполнение запросов за счет того, что все необходимые данные извлекаются из самого индекса без обращения к таблице.

В арсенале планировщика имеется сканирование по битовой карте, которое позволяет объединять сразу несколько индексов для ускорения доступа.

Кроссплатформенность

PostgreSQL работает на операционных системах семейства Unix, включая серверные и клиентские разновидности Linux, FreeBSD, Solaris и macOS, а также на Windows.

За счет открытого и переносимого кода на языке C PostgreSQL можно собирать на самых разных платформах, даже если для них отсутствует поддерживаемая сообществом сборка.

Расширяемость

Расширяемость — одно из фундаментальных преимуществ системы, лежащее в основе архитектуры PostgreSQL. Пользователи могут самостоятельно, не меняя базовый код системы, добавлять:

- типы данных;
- функции и операторы для работы с новыми типами;
- индексные и табличные методы доступа;
- языки серверного программирования;
- подключения к внешним источникам данных (Foreign Data Wrappers);
- загружаемые расширения.

Полноценная поддержка расширений позволяет реализовать функционал любой сложности не внося изменений в ядро PostgreSQL и допуская подключение по мере необходимости. Например, именно в виде расширений построены такие сложные системы, как:

- CitusDB — возможность распределения данных по разным экземплярам PostgreSQL (шардинг) и массивно-параллельного выполнения запросов;
- PostGIS — одна из наиболее известных и мощных систем обработки геоинформационных данных;
- TimescaleDB — работа с временны́ми рядами, включая специальное секционирование и шардирование.

Только стандартный комплект, входящий в сборку PostgreSQL 15, содержит около полусотни расширений, доказавших свою надежность и полезность.

Доступность

Либеральная лицензия PostgreSQL, сходная с лицензиями BSD и MIT, разрешает неограниченное использование СУБД, модификацию кода, а также включение в состав других продуктов, в том числе закрытых и коммерческих.

Независимость

PostgreSQL не принадлежит ни одной компании и развивается международным сообществом, в том числе и российскими разработчиками. Это означает, что системы, использующие PostgreSQL, не зависят от какого-либо конкретного производителя, благодаря чему вложенные в них средства сохранятся в любой ситуации.

II Новое в PostgreSQL 15

Если вы знакомы с предыдущими версиями PostgreSQL, эта глава даст вам представление о том, что успело поменяться за прошедший год. Здесь перечислена только часть изменений; полный список, как обычно, смотрите в замечаниях к выпуску: postgrespro.ru/docs/postgresql/15/release-15.

Команды SQL

Наконец-то **реализована команда MERGE** после неудачной попытки, предпринятой еще в 11-й версии. Эта команда определена стандартом SQL. Она более гибкая и в ряде случаев более эффективная, чем имеющаяся `INSERT ... ON CONFLICT`.

Считать ли значения **NULL уникальными** в ограничениях целостности? Раньше был только один правильный ответ — отрицательный, — но теперь можно выбирать с помощью предложения `NULLS [NOT] DISTINCT`.

Допускается **список столбцов в `ON DELETE SET NULL`** для составных внешних ключей: при удалении родительской записи можно сбрасывать не все поля, а только часть.

Команду `COPY` научили принимать и выводить **заголовок в первой строке** данных.

Функции

Добавлены функции **`unnest` и `range_agg` для мультидиапазонных типов**, введенных в 14-й версии.

Появились **новые функции для регулярных выражений** для упрощения миграции: `regexp_like`, `regexp_count`, `regexp_instr`, `regexp_substr`.

Функции **`pg_size_pretty` и `pg_size_bytes` обучили петабайтам**. Приставки для других кратных единиц приберегли на будущее.

Секционирование

Переименование триггера на секционированной таблице автоматически **переименует триггеры на таблицах-секциях**.

Для секционированных таблиц **поддерживается команда `CLUSTER`**.

Запросы планируются быстрее, когда бóльшая часть секций отсекается на этапе планирования.

Журнал предзаписи

Добавлены алгоритмы сжатия **LZ4 и ZStd для полных образов страниц**. По сравнению со стандартным PGLZ алгоритм LZ4 обычно менее ресурсоемок при той же эффективности, а ZStd сильнее нагружает процессор, но и лучше сжимает. Алгоритму LZ4 обучена и утилита pg_receivewal.

Возможна **предвыборка WAL при восстановлении**, что может ускорить запуск сервера после сбоя, сократить время развертывания из резервной копии и увеличить скорость применения журнальных записей при репликации.

Непрерывная архивация может использовать **загружаемые модули** вместо запуска команды ОС. Эту возможность должны подхватить создатели систем резервного копирования.

Другая недоступная пользователю напрямую новинка — возможность создавать **собственные менеджеры ресурсов**. Это важно для разработчиков новых методов доступа, как индексных, так и табличных. Кстати, теперь **табличный метод доступа можно поменять** командой `ALTER TABLE ... SET METHOD` – но пока не на что.

Появилось **расширение pg_walinspect** для просмотра записей WAL SQL-запросом в дополнение к имеющейся утилите pg_waldump.

Логическая репликация

В этом выпуске логическая репликация была значительно доработана. Есть надежда, что скоро будут реализованы и такие долгожданные возможности, так репликация последовательностей и команд DDL.

При создании публикации можно **фильтровать строки и столбцы**, а также **реплицировать все таблицы схемы** (`FOR ALL TABLES IN SCHEMA`).

Добавлена поддержка **репликации подготовленных транзакций**.

Процесс на стороне сервера-подписчика выполняется **с правами владельца подписки**; суперпользовательские права больше не нужны.

Можно **остановить подписку при возникновении конфликта**; раньше процесс приема журнальных записей перезапускался каждую секунду. А для разрешения конфликта теперь можно **пропустить конфликтующую транзакцию** (`ALTER SUBSCRIPTION ... SKIP`).

Резервное копирование

Старая добрая утилита pg_basebackup существенно переработана.

Появилась возможность указать, **где создавать резервную копию**: на клиенте или на сервере. В соответствии с идеей расширяемости можно создавать собственные цели; например, новое расширение `basebackup_to_shell` позволяет передать резервную копию команде ОС.

Поддерживается **сжатие данных на стороне сервера** любым из алгоритмов Gzip, LZ4 и ZStd, что может быть полезно при низкой пропускной способности сети.

Безопасность

Отозван доступ на запись в схему `public`, который по умолчанию имелся у всех пользователей и доставался им от псевдороли `public`. Схемой `public` теперь владеет **новая псевдороль `pg_database_owner`**, неявно включающая владельца текущей базы данных.

Системный администратор имеет все больше возможностей передать свои задачи непривилегированным пользователям. **Право выполнения команды `CHECKPOINT`** выдано новой роли `pg_checkpointer`. **Доступ к информации о выделении серверной памяти** (представления `pg_backend_memory_contexts` и `pg_shmem_allocations`) выдан роли `pg_read_all_stats`. Появилась возможность **разграничить доступ к конфигурационным параметрам** (`GRANT SET` и `GRANT ALTER SYSTEM`).

Появилась возможность создавать **представления с правами вызывающего**: в этом случае у пользователя должны быть права на используемые в представлении объекты.

Обычные пользователи больше **не могут управлять членством в своей роли** (`ADMIN OPTION`), то есть включать в свою роль и исключать из своей роли другие роли.

Мониторинг

Кумулятивная статистика перенесена в общую память сервера, а процесс-коллектор теперь не используется. Больше не надо монтировать tmpfs для файлов статистики.

Появились **события ожидания для команд файлового архива**: ArchiveCommand, ArchiveCleanupCommand, RestoreCommand, RecoveryEndCommand.

Поддерживается **формат JSON для журнала сообщений сервера**. Такой формат программно гораздо удобнее разбирать, чем обычный текстовый вывод.

Функция `pg_log_backend_memory_contexts` стала выводить **распределение памяти служебных процессов**, а не только обслуживающих.

Postgres_fdw

Обертку сторонних данных `postgres_fdw`, через которую лежит путь к встроенному шардированию, научили **передавать выражения CASE** на внешний сервер и **устанавливать параметру *application_name*** произвольное значение.

Транзакции, начатые на внешних серверах, **могут фиксироваться параллельно**, что повышает производительность. К сожалению, о настоящих распределенных транзакциях речи пока не идет.

Очистка и заморозка

Проделана большая работа по рефакторингу очистки и заморозки. Теперь VACUUM не пропускает страницы, которые не может заблокировать, а выполняет для них хотя бы часть работы. Как следствие, **граница `relfrozenxid` может продвигаться и в неагрессивном режиме**, а, значит, агрессивная очистка может выполняться реже.

Команда **`VACUUM VERBOSE` выводит больше полезной информации**, которую раньше можно было получить только в журнале сообщений.

Оптимизации

Сортировка эффективнее использует память, что увеличивает шансы обойтись без обращения к диску, а **сортировка одного столбца выполняется быстрее**.

Проверка строк в кодировке UTF-8 ускорена за счет одновременной обработки нескольких байт.

В параллельных запросах **ускорена пересылка данных** от рабочих процессов к ведущему. До изменения значения *parallel_tuple_cost* дело, правда, не дошло.

Параллельная агрегация работает давно, а теперь и команду **`SELECT DISTINCT` научили выполняться параллельно**.

Лучше оценивается **кардинальность условий с функцией `starts_with`** или оператором `^@` благодаря добавленным вспомогательным функциям для планировщика.

Значение параметра ***hash_mem_multiplier*** **увеличено до 2.0** (было 1.0). Это означает, что по умолчанию для хеш-таблиц будет выделяться в два раза больше памяти, чем *work_mem*: запросам это пойдет на пользу, но за расходом памяти надо следить.

Можно **управлять размером рабочей таблицы в рекурсивных запросах** с помощью параметра *recursive_worktable_factor*.

Разное

Провайдер локалей ICU, добавленный еще в 10-й версии, можно использовать для баз данных и при инициализации кластера. Версия провайдера отслеживается в системном каталоге.

Python 2 больше не поддерживается, языки `plpython2u` и `plpythonu` удалены.

Документация

Документация обзавелась новой главой о хеш-индексах: postgrespro.ru/docs/postgresql/15/hash-index.

III Установка и начало работы

Что нужно для начала работы с PostgreSQL? В этой главе мы объясним, как установить службу PostgreSQL и управлять ею, а в следующей создадим простую базу данных и на ее примере изложим основы языка SQL, на котором формулируются запросы.

Мы возьмем обычный («ванильный») дистрибутив PostgreSQL 15. Сервер PostgreSQL устанавливается и запускается по-разному в зависимости от того, какая у вас операционная система:

- если Windows, читайте дальше;
- если Linux семейства Debian или Ubuntu — переходите к с. 30.

Инструкции по установке для других операционных систем есть здесь: postgresql.org/download.

С тем же успехом вы можете воспользоваться и дистрибутивом Postgres Pro Standard 15: он полностью совместим с обычной СУБД PostgreSQL, включает некоторые разработки, выполненные в нашей компании Postgres Professional, и бесплатен при использовании в ознакомительных и образовательных целях. В этом случае инструкции по установке ищите на сайте postgrespro.ru/products/download.

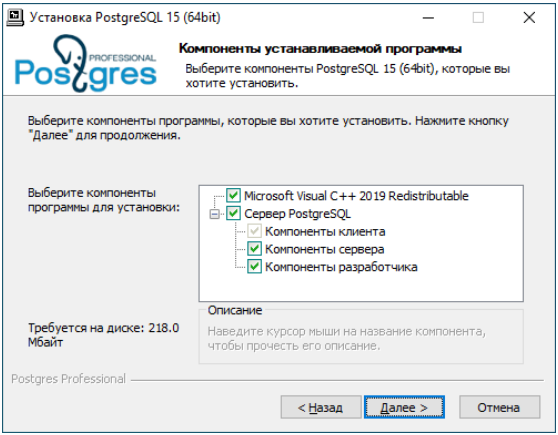
Windows

Установка

Скачайте установщик (postgrespro.ru/windows), запустите его и выберите язык установки.

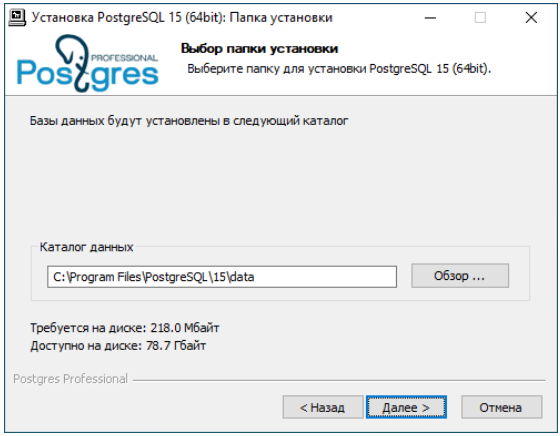
Установщик построен в традиционном стиле «мастера»: если вас все устраивает, просто нажимайте «Далее». Рассмотрим основные этапы установки.

Компоненты устанавливаемой программы (если сомневаетесь в выборе, ничего не меняйте):



Далее идет выбор каталога для установки PostgreSQL. По умолчанию установка выполняется в папку `C:\Program Files\PostgreSQL\15`.

Отдельно можно выбрать расположение каталога для баз данных:

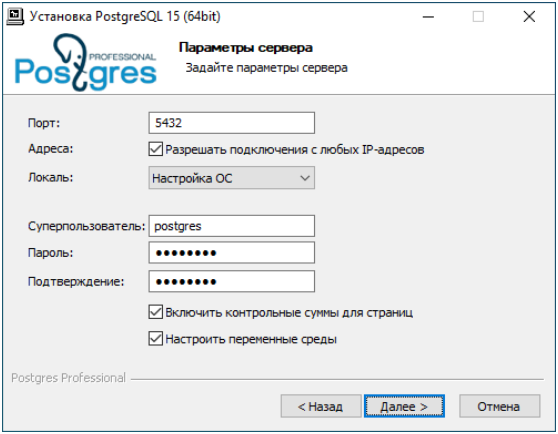


Именно здесь будет находиться хранимая в СУБД информация, так что если вы планируете хранить очень много данных — убедитесь, что на диске достаточно места.

Чтобы работать с данными на языке, отличном от английского, выберите подходящую локаль (например, «Russian, Russia») или оставьте вариант «Настройка ОС», если нужная локаль установлена в Windows.

Введите (и подтвердите повторным вводом) пароль пользователя СУБД `postgres`. Также отметьте флажок «Настроить переменные среды», чтобы подключаться к серверу PostgreSQL под текущей учетной записью ОС.

Остальные параметры можно оставить без изменений:

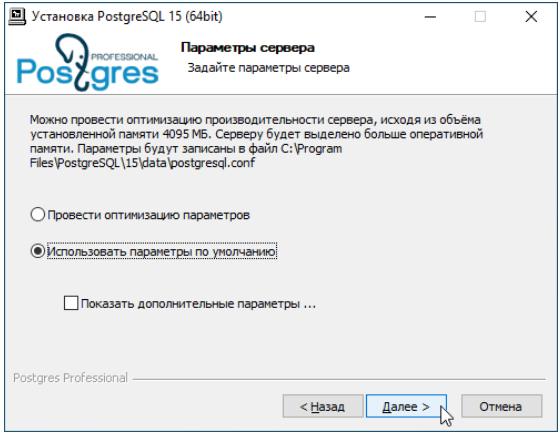


Если вы планируете установить PostgreSQL только в ознакомительных целях, на следующем экране отметьте вариант «Использовать параметры по умолчанию», чтобы СУБД не занимала много оперативной памяти.

Управление службой и основные файлы

При установке PostgreSQL в системе регистрируется служба «postgresql-15». Она запускается автоматически при старте компьютера под учетной записью Network Service (Сетевая служба). При необходимости параметры службы можно изменить штатными средствами Windows.

Для временной остановки службы сервера баз данных запустите программу Stop Server из папки в меню «Пуск», указанной при установке.



Запускается служба программой Start Server, находящейся там же.

Если при запуске службы произошла ошибка, об этом делается отметка в журнале сообщений сервера. Журнал находится в подкаталоге `log` каталога, выбранного при установке для баз данных (обычно `C:\Program Files\PostgreSQL\15\data\log`). Журнал настроен так, чтобы запись периодически переключалась в новый файл. Найти актуальный файл можно по дате последнего изменения или по имени, которое содержит дату и время переключения.

Есть несколько важных конфигурационных файлов, которые определяют настройки сервера. Они располагаются в каталоге баз данных. При начальном ознакомлении с PostgreSQL изменять их не нужно, но в реальной работе

они непременно потребуются. Обязательно загляните в эти файлы — они прекрасно документированы:

- `postgresql.conf` — это основной конфигурационный файл, содержащий значения параметров сервера;
- `pg_hba.conf` — файл, определяющий настройки доступа. В целях безопасности по умолчанию доступ должен быть подтвержден паролем и допускается только с локального компьютера.

Теперь мы готовы подключиться к базе данных и пробовать давать команды и выполнять запросы. Переходите к разделу «Пробуем SQL» на с. 35.

Debian и Ubuntu

Установка

Если вы используете Linux, то для установки необходимо подключить пакетный репозиторий PGDG (PostgreSQL Global Development Group). В настоящее время для системы Debian поддерживаются версии 10 «Buster», 11 «Bullseye» и 12 «Bookworm», а для Ubuntu — 18.04 «Bionic», 20.04 «Focal», 22.04 «Jammy» и 22.10 «Kinetic».

Выполните в терминале следующие команды:

```
$ sudo apt-get install lsb-release

$ sudo sh -c 'echo "deb \
  http://apt.postgresql.org/pub/repos/apt/ \
  $(lsb_release -cs)-pgdg main" \
  > /etc/apt/sources.list.d/pgdg.list'
```

```
$ wget --quiet -O - \
  https://postgresql.org/media/keys/ACCC4CF8.asc \
  | sudo apt-key add -
```

Репозиторий подключен, обновим список пакетов:

```
$ sudo apt-get update
```

Перед установкой проверьте настройки локализации:

```
$ locale
```

Чтобы работать с данными на языке, отличном от английского, может потребоваться изменить значение переменных LC_CTYPE и LC_COLLATE. Для русского языка подходит и локаль «en_US.UTF8», но все-таки лучше ее сменить:

```
$ export LC_CTYPE=ru_RU.UTF8
$ export LC_COLLATE=ru_RU.UTF8
```

Также убедитесь, что в операционной системе установлена соответствующая локаль:

```
$ locale -a | grep ru_RU
ru_RU.utf8
```

Если это не так, сгенерируйте ее:

```
$ sudo locale-gen ru_RU.utf8
```

Теперь можно приступить к установке:

```
$ sudo apt-get install postgresql-15
```

Это был последний этап; теперь СУБД PostgreSQL установлена, запущена и готова к работе. Чтобы проверить это, выполните команду:

```
$ sudo -u postgres psql -c 'select now()'
```

Если все проделано успешно, в ответ вы должны получить текущее время.

Управление службой и основные файлы

При установке PostgreSQL создает специальную учетную запись `postgres`, от имени которой работают процессы, обслуживающие сервер, и которой принадлежат все файлы, относящиеся к СУБД. PostgreSQL будет автоматически запускаться при перезагрузке операционной системы. С настройками по умолчанию это не проблема, так как при отсутствии обращений к серверу ресурсов системы тратится совсем немного. Если вы все-таки захотите отключить автозапуск, выполните:

```
$ sudo systemctl disable postgresql
```

Чтобы временно остановить службу сервера баз данных, выполните команду:

```
$ sudo systemctl stop postgresql
```

Запустить службу сервера можно командой:

```
$ sudo systemctl start postgresql
```

Можно также проверить текущее состояние:

```
$ sudo systemctl status postgresql
```

Если служба не запускается, найти причину поможет журнал сообщений сервера. Внимательно прочитайте самые последние записи из журнала, который находится в файле `/var/log/postgresql/postgresql-15-main.log`.

Вся информация, которая содержится в базе данных, располагается в файловой системе в специальном каталоге `/var/lib/postgresql/15/main/`. Если вы собираетесь хранить очень много данных — убедитесь, что для них хватит места.

Есть несколько важных конфигурационных файлов, которые определяют настройки сервера. При начале работы изменять эти файлы не нужно, но лучше ознакомиться с ними заранее — в дальнейшем они непременно понадобятся:

- `/etc/postgresql/15/main/postgresql.conf` — основной конфигурационный файл, содержащий значения параметров сервера;
- `/etc/postgresql/15/main/pg_hba.conf` — файл, определяющий настройки доступа. В целях безопасности по умолчанию доступ разрешен только с локального компьютера и только от лица пользователя базы данных, имя которого совпадает с именем учетной записи операционной системы.

Самое время подключиться к базе данных и попробовать SQL в деле.

IV Пробуем SQL

Подключение с помощью psql

Чтобы подключиться к серверу СУБД и выполнить какие-либо команды, требуется программа-клиент. В главе «PostgreSQL для приложения» мы будем говорить о том, как посылать запросы из программ на разных языках программирования, а сейчас речь пойдет о терминальном клиенте `psql`, работа с которым происходит интерактивно в режиме командной строки.

К сожалению, в наше время многие недолюбливают командную строку. Почему имеет смысл научиться с ней работать?

Во-первых, `psql` — стандартный клиент, он входит в любую сборку PostgreSQL и поэтому всегда под рукой. Иметь настроенную под себя среду — это, конечно, хорошо, но оказаться беспомощным в среде незнакомой просто нелогично.

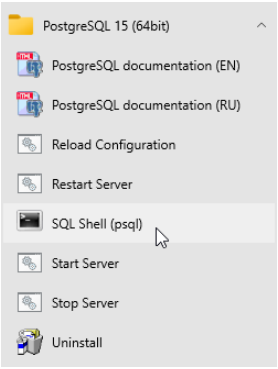
Во-вторых, `psql` действительно удобен для решения повседневных задач по администрированию баз данных, для написания небольших запросов и для автоматизации процессов, например, периодической установки обновлений программного кода на сервер СУБД. Он имеет собственные команды, позволяющие сориентироваться в объектах,

хранящихся в базе данных, и представить информацию из таблиц в наглядном виде.

Но если вы привыкли работать с графическими пользовательскими интерфейсами, попробуйте pgAdmin — мы еще упомянем эту программу ниже — или другие аналогичные продукты: wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools

Чтобы запустить `psql` в операционной системе Linux, выполните команду:

```
$ sudo -u postgres psql
```



В ОС Windows запустите программу «SQL Shell (psql)» из меню «Пуск». В ответ на запрос введите пароль пользователя `postgres`, указанный при установке PostgreSQL.

Пользователи Windows могут столкнуться с проблемой неправильного отображения символов кириллицы в терминале. В этом случае убедитесь, что свойствах окна терминала установлен TrueType-шрифт (обычно «Lucida Console» или «Consolas»).

Итак, приглашение выглядит одинаково в обеих операционных системах: `postgres=#` («postgres» здесь — это имя

базы данных, к которой вы сейчас подключены). Один сервер может обслуживать несколько БД, но работать в каждый момент времени можно только с одной.

Теперь изучим первые команды. Вводите только то, что выделено жирным; приглашение и ответ системы на команду приведены исключительно для удобства.

База данных

Создадим новую базу данных с именем test. Выполните:

```
postgres=# CREATE DATABASE test;
CREATE DATABASE
```

Не забудьте про точку с запятой в конце команды — пока PostgreSQL не увидит этот символ, он будет считать, что вы продолжаете ввод (то есть команда может быть разбита на несколько строк).

Теперь переключимся на созданную базу:

```
postgres=# \c test
You are now connected to database "test" as user
"postgres".
test=#
```

Как видите, приглашение сменилось на `test=#`.

Команда, которую мы только что ввели, не похожа на SQL — она начинается с обратной косой черты. Так выглядят специальные команды, которые понимает только `psql` (поэтому, если у вас открыт pgAdmin или другое графическое средство, пропускайте все, что начинается с косой черты, или поищите замену).

Команд `psql` довольно много; с некоторыми из них мы познакомимся чуть позже, а полный список с краткими описаниями можно получить прямо сейчас:

```
test=# \?
```

Поскольку справочная информация довольна объемна, она будет показана с помощью настроенной в операционной системе команды-пейджера (обычно `more` или `less`).

Таблицы

В реляционных СУБД данные представляются в виде **таблиц**. Структура таблицы определяется ее **столбцами**. Собственно данные располагаются в **строках;** они хранятся неупорядоченными и даже не обязательно располагаются в порядке их добавления в таблицу.

Для каждого столбца устанавливается **тип данных**; значения полей в строках должны соответствовать этим типам. PostgreSQL располагает большим числом встроенных типов (postgrespro.ru/doc/datatype) и возможностями для создания новых, но мы ограничимся самыми основными:

- integer — целые числа;
- text — текстовые строки;
- boolean — логический тип, принимающий значения `true` («истинно») или `false` («ложно»).

Помимо обычных значений, определяемых типом данных, поле может иметь **неопределенное значение** NULL — его можно рассматривать как «значение неизвестно» или «значение не задано».

Давайте создадим таблицу дисциплин, читаемых в вузе:

```
test=# CREATE TABLE courses(
test(#   c_no text PRIMARY KEY,
test(#   title text,
test(#   hours integer
test(# );
CREATE TABLE
```

Обратите внимание, как меняется приглашение `psql`: это подсказка, что ввод команды продолжается на новой строке. В дальнейшем для удобства мы не будем дублировать приглашение на каждой строке.

Этой командой мы определили, что таблица с именем `courses` будет состоять из трех столбцов: `c_no` — текстовый номер курса, `title` — название курса, и `hours` — целое число лекционных часов.

Кроме столбцов и типов данных, можно ввести ограничения целостности, которые будут проверяться автоматически, — СУБД не допустит появления в базе некорректных данных. В нашем примере добавлено ограничение `PRIMARY KEY` для столбца `c_no`; теперь в нем не допускаются повторяющиеся, а также неопределенные значения. С помощью такого столбца можно отличать строки друг от друга. Полный список ограничений целостности есть на странице postgrespro.ru/doc/ddl-constraints.

Точный синтаксис команды `CREATE TABLE` можно посмотреть в документации, а можно прямо в `psql`:

```
test=# \help CREATE TABLE
```

Такая справка есть по каждой команде SQL, а полный список команд покажет `\help` без параметров.

Наполнение таблиц

Добавим в созданную таблицу несколько строк:

```
test=# INSERT INTO courses(c_no, title, hours)
VALUES ('CS301', 'Базы данных', 30),
       ('CS305', 'Сети ЭВМ', 60);
```

```
INSERT 0 2
```

Для массовой загрузки данных из внешнего источника команда INSERT подходит плохо, зато есть специально предназначенная для этого команда COPY: postgrespro.ru/doc/sql-copy.

Создадим в базе еще две таблицы: «Студенты» и «Экзамены». Пусть по каждому студенту хранится его имя и год поступления, а идентифицироваться он будет номером студенческого билета.

```
test=# CREATE TABLE students(
  s_id integer PRIMARY KEY,
  name text,
  start_year integer
);
```

```
CREATE TABLE
```

```
test=# INSERT INTO students(s_id, name, start_year)
VALUES (1451, 'Анна', 2014),
       (1432, 'Виктор', 2014),
       (1556, 'Нина', 2015);
```

```
INSERT 0 3
```

Таблица экзаменов содержит данные об оценках, полученных студентами по различным дисциплинам. Таким образом, студенты и дисциплины связаны друг с другом отношением «многие ко многим»: один студент может сдавать

экзамены по многим дисциплинам, а экзамен по одной дисциплине могут сдавать много студентов.

Запись в таблице экзаменов идентифицируется совокупностью номера студбилета и номера курса. Такое ограничение целостности, относящее сразу к нескольким столбцам, определяется с помощью предложения CONSTRAINT:

```
test=# CREATE TABLE exams(
  s_id integer REFERENCES students(s_id),
  c_no text REFERENCES courses(c_no),
  score integer,
  CONSTRAINT pk PRIMARY KEY(s_id, c_no)
);
CREATE TABLE
```

Кроме того, с помощью предложения REFERENCES мы добавили два ограничения ссылочной целостности, называемые **внешними ключами**. Такие ограничения показывают, что значения в одной таблице **ссылаются** на строки в другой таблице.

Теперь при любых действиях СУБД будет проверять соответствие всех идентификаторов s_id, указанных в таблице экзаменов, реальным студентам (то есть записям в таблице студентов), а также номера c_no — реальным курсам. Таким образом, будет исключена возможность поставить оценку несуществующему студенту или же по несуществующей дисциплине — независимо от действий пользователя или возможных ошибок в приложении.

Поставим нашим студентам несколько оценок:

```
test=# INSERT INTO exams(s_id, c_no, score)
VALUES (1451, 'CS301', 5),
       (1556, 'CS301', 5),
       (1451, 'CS305', 5),
       (1432, 'CS305', 4);
INSERT 0 4
```

Выборка данных

Простые запросы

Чтение данных из таблиц выполняется оператором SQL SELECT. Для примера выведем только два столбца из таблицы courses. Конструкция AS позволяет переименовать столбец, если это необходимо:

```
test=# SELECT title AS course_title, hours
FROM courses;
 course_title | hours
--------------+-------
 Базы данных  |    30
 Сети ЭВМ     |    60
(2 rows)
```

Чтобы вывести все столбцы, достаточно указать символ звездочки:

```
test=# SELECT * FROM courses;
 c_no  |    title    | hours
-------+-------------+-------
 CS301 | Базы данных |    30
 CS305 | Сети ЭВМ    |    60
(2 rows)
```

В промышленном коде лучше явно перечислять только необходимые столбцы, чтобы запрос выполнялся эффективнее, а результат не зависел от появления новых столбцов. Но для интерактивных запросов «звездочка» очень удобна.

Выдача по запросу может содержать одинаковые строки. Когда выводятся не все столбцы — дубликаты могут появиться даже если в исходной таблице их не было:

```
test=# SELECT start_year FROM students;
 start_year
------------
       2014
       2014
       2015
(3 rows)
```

Чтобы выбрать все **различные** года поступления, после SELECT надо добавить слово DISTINCT:

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

Подробнее смотрите в документации: postgrespro.ru/doc/sql-select#SQL-DISTINCT

Вообще после слова SELECT можно указывать любые выражения. А без предложения FROM запрос вернет одну строку. Например:

```
test=# SELECT 2+2 AS result;
 result
--------
      4
(1 row)
```

Обычно при выборке данных требуется получить не все строки, а только те, которые удовлетворят какому-либо условию. Такое условие фильтрации записывается в предложении WHERE:

```
test=# SELECT * FROM courses WHERE hours > 45;
```

```
 c_no  |  title   | hours
-------+----------+-------
 CS305 | Сети ЭВМ |    60
(1 row)
```

Условие должно иметь логический тип. Например, оно может содержать операторы =, <> (или !=), >, >=, <, <=, а также может объединять более простые условия с помощью логических операций AND, OR, NOT и круглых скобок — как в обычных языках программирования.

Тонкий момент представляет собой неопределенное значение NULL. В выборку попадают только те строки, для которых условие фильтрации истинно; если же значение ложно **или не определено,** строка отбрасывается.

Учтите:

- результат сравнения чего-либо с неопределенным значением не определен;
- результат логических операций с неопределенным значением, как правило, не определен (исключения: `true OR NULL = true`, `false AND NULL = false`);
- для проверки определенности значения используются специальные операторы `IS NULL` (`IS NOT NULL`) и `IS DISTINCT FROM` (`IS NOT DISTINCT FROM`).

При работе с неопределенными значениями часто используют выражение `coalesce` (читается «коуэлес») для замены NULL на что-нибудь другое, например, на пустую строку для текстовых типов или на ноль для числовых.

Подробнее смотрите в документации: postgrespro.ru/doc/functions-comparison.

Соединения

Грамотно спроектированная реляционная база данных не содержит избыточной информации. Например, таблица экзаменов не должна содержать имя студента, потому что его можно найти в другой таблице по номеру студенческого билета.

Поэтому для получения всех необходимых значений в запросе часто приходится соединять данные из нескольких таблиц, перечисляя их имена в предложении FROM:

```
test=# SELECT * FROM courses, exams;
 c_no  |    title    | hours | s_id | c_no  | score
-------+-------------+-------+------+-------+-------
 CS301 | Базы данных |    30 | 1451 | CS301 |     5
 CS305 | Сети ЭВМ    |    60 | 1451 | CS301 |     5
 CS301 | Базы данных |    30 | 1556 | CS301 |     5
 CS305 | Сети ЭВМ    |    60 | 1556 | CS301 |     5
 CS301 | Базы данных |    30 | 1451 | CS305 |     5
 CS305 | Сети ЭВМ    |    60 | 1451 | CS305 |     5
 CS301 | Базы данных |    30 | 1432 | CS305 |     4
 CS305 | Сети ЭВМ    |    60 | 1432 | CS305 |     4
(8 rows)
```

То, что у нас получилось, называется прямым или декартовым произведением таблиц — к каждой строке одной таблицы добавляется каждая строка другой.

Как правило, более полезный и содержательный результат можно получить, указав в предложении WHERE условие соединения. Запросим оценки по всем дисциплинам, сопоставляя курсы с теми экзаменами, которые проводились именно по данному курсу:

```
test=# SELECT courses.title, exams.s_id, exams.score
FROM courses, exams
WHERE courses.c_no = exams.c_no;
```

```
    title    | s_id | score
-------------+------+-------
 Базы данных | 1451 |     5
 Базы данных | 1556 |     5
 Сети ЭВМ    | 1451 |     5
 Сети ЭВМ    | 1432 |     4
(4 rows)
```

Запросы можно формулировать и в другом виде, указывая соединения с помощью ключевого слова JOIN. Выведем студентов и их оценки по курсу «Сети ЭВМ»:

```
test=# SELECT students.name, exams.score
FROM students
JOIN exams
  ON students.s_id = exams.s_id
  AND exams.c_no = 'CS305';
  name  | score
--------+-------
 Анна   |     5
 Виктор |     4
(2 rows)
```

С точки зрения СУБД эти формы эквивалентны друг другу, так что можно использовать тот способ, который представляется более наглядным.

Как видно, в выдаче нет тех строк таблицы, указанной слева от слова JOIN, для которых не нашлось пары в таблице, указанной справа от этого слова: условие наложено на дисциплины, но исключаются и студенты, не сдававшие по ней экзамен. Чтобы в выборку попали все студенты, надо использовать внешнее соединение:

```
test=# SELECT students.name, exams.score
FROM students
LEFT JOIN exams
  ON students.s_id = exams.s_id
  AND exams.c_no = 'CS305';
```

```
  name   | score
---------+-------
 Анна    |     5
 Виктор  |     4
 Нина    |
(3 rows)
```

Теперь в выдаче есть и те строки из левой таблицы (поэтому операция называется LEFT JOIN), для которых не нашлось пары в правой. При этом для столбцов правой таблицы возвращаются неопределенные значения.

Условия после WHERE применяются к уже соединенным строкам, поэтому, если вынести ограничение на дисциплины из условия соединения в предложение WHERE, Нина не попадет в выборку — ведь для нее значение exams.c_no не определено:

```
test=# SELECT students.name, exams.score
FROM students
LEFT JOIN exams ON students.s_id = exams.s_id
WHERE exams.c_no = 'CS305';

  name   | score
---------+-------
 Анна    |     5
 Виктор  |     4
(2 rows)
```

Не стоит опасаться соединений. Это обычная и естественная для реляционных СУБД операция, и у PostgreSQL имеется целый арсенал эффективных механизмов для ее выполнения. Не соединяйте данные в приложении, доверьте эту работу серверу баз данных — он прекрасно с ней справляется.

Подробнее смотрите в документации: postgrespro.ru/doc/sql-select#SQL-FROM.

Подзапросы

Оператор SELECT формирует таблицу, которая (как мы уже видели) может быть выведена в качестве результата, а может быть использована в другой конструкции языка SQL в любом месте, где по смыслу может находиться таблица. Такая вложенная команда SELECT, заключенная в круглые скобки, называется **подзапросом.**

Если подзапрос возвращает ровно одну строку и ровно один столбец, его можно использовать как обычное скалярное выражение:

```
test=# SELECT name,
  (SELECT score
   FROM exams
   WHERE exams.s_id = students.s_id
   AND exams.c_no = 'CS305')
FROM students;

  name   | score
---------+-------
 Анна    |     5
 Виктор  |     4
 Нина    |
(3 rows)
```

Если скалярный подзапрос, использованный в списке выражений SELECT, не содержит ни одной строки, возвращается неопределенное значение (как в последней строке выдачи по нашему примеру). Поэтому скалярные подзапросы можно раскрыть, заменив их на соединение, но обязательно внешнее.

Скалярные подзапросы можно также использовать в условиях фильтрации. Получим все экзамены, которые сдавали студенты, поступившие после 2014 года:

```
test=# SELECT *
FROM exams
WHERE (SELECT start_year
       FROM students
       WHERE students.s_id = exams.s_id) > 2014;
```

```
 s_id | c_no  | score
------+-------+-------
 1556 | CS301 |     5
(1 row)
```

В SQL можно формулировать условия и на подзапросы, возвращающие произвольное количество строк. Для этого существует несколько конструкций, одна из которых — отношение IN — проверяет, содержится ли значение в таблице, возвращаемой подзапросом.

Выведем студентов, получивших какие-либо оценки по указанному курсу:

```
test=# SELECT name, start_year
FROM students
WHERE s_id IN (SELECT s_id
               FROM exams
               WHERE c_no = 'CS305');
```

```
  name  | start_year
--------+------------
 Анна   |       2014
 Виктор |       2014
(2 rows)
```

Оператор NOT IN возвращает противоположный результат. Например, список студентов, не получивших ни одной отличной оценки:

```
test=# SELECT name, start_year
FROM students
WHERE s_id NOT IN
      (SELECT s_id FROM exams WHERE score = 5);
```

```
  name  | start_year
--------+------------
 Виктор |       2014
(1 row)
```

Обратите внимание, что в выборку попадут и студенты, не получившие ни одной оценки.

Еще одна возможность — использовать предикат EXISTS, проверяющий, возвратил ли подзапрос хотя бы одну строку. С его помощью можно записать предыдущий запрос в другом виде:

```
test=# SELECT name, start_year
FROM students
WHERE NOT EXISTS (SELECT s_id
                  FROM exams
                  WHERE exams.s_id = students.s_id
                  AND score = 5);
```

```
  name  | start_year
--------+------------
 Виктор |       2014
(1 row)
```

Подробнее смотрите в документации: postgrespro.ru/doc/functions-subquery.

Выше мы дополняли имена столбцов именами таблиц, чтобы избежать неоднозначности, но иногда этого недостаточно. Например, одна и та же таблица может фигурировать в запросе дважды, или на месте таблицы в предложении FROM может стоять безымянный подзапрос. В таких случаях после подзапроса можно указывать его псевдоним (alias). Обычным таблицам тоже можно присваивать псевдонимы.

Выведем имена студентов и их оценки по предмету «Базы данных»:

```
test=# SELECT s.name, ce.score
FROM students s
JOIN (SELECT exams.*
      FROM courses, exams
      WHERE courses.c_no = exams.c_no
      AND courses.title = 'Базы данных') ce
  ON s.s_id = ce.s_id;
 name | score
------+-------
 Анна |     5
 Нина |     5
(2 rows)
```

Здесь s — псевдоним таблицы, а ce — псевдоним подзапроса. Псевдонимы лучше делать короткими, но понятными.

Тот же запрос можно записать и без подзапросов:

```
test=# SELECT s.name, e.score
FROM students s, courses c, exams e
WHERE c.c_no = e.c_no
AND c.title = 'Базы данных'
AND s.s_id = e.s_id;
```

Сортировка

Как уже говорилось, данные в таблицах не упорядочены. Для вывода строк результата в определенном порядке используется предложение ORDER BY со списком выражений, по которым надо выполнить сортировку. После каждого выражения (ключа сортировки) можно указать направление: ASC — по возрастанию (этот порядок используется по умолчанию) — или DESC — по убыванию.

```
test=# SELECT *
FROM exams
ORDER BY score, s_id, c_no DESC;
```

```
 s_id | c_no  | score
------+-------+-------
 1432 | CS305 |     4
 1451 | CS305 |     5
 1451 | CS301 |     5
 1556 | CS301 |     5
(4 rows)
```

Здесь строки упорядочены по возрастанию оценки; если оценки совпадают — по возрастанию номера студенческого билета; если же совпадают оба первых ключа — по убыванию номера курса.

Операцию сортировки имеет смысл выполнять в конце запроса непосредственно перед получением результата; в подзапросах она обычно бессмысленна.

Подробнее смотрите в документации: postgrespro.ru/doc/sql-select#SQL-ORDERBY.

Группировка

При группировке в одной строке результата размещается значение, вычисленное на основании данных нескольких строк исходных таблиц. Вместе с группировкой используют **агрегатные функции**. Например, выведем общее количество проведенных экзаменов, количество сдававших их студентов и средний балл:

```
test=# SELECT count(*), count(DISTINCT s_id),
avg(score)
FROM exams;
 count | count |        avg
-------+-------+--------------------
     4 |     3 | 4.7500000000000000
(1 row)
```

Ту же информацию можно получить в разбивке по номерам курсов с помощью предложения GROUP BY, в котором указываются ключи группировки:

```
test=# SELECT c_no, count(*),
count(DISTINCT s_id), avg(score)
FROM exams
GROUP BY c_no;
 c_no  | count | count |        avg
-------+-------+-------+--------------------
 CS301 |     2 |     2 | 5.0000000000000000
 CS305 |     2 |     2 | 4.5000000000000000
(2 rows)
```

Полный список агрегатных функций: postgrespro.ru/doc/functions-aggregate.

При использовании группировки может возникнуть необходимость отфильтровать строки на основании результатов агрегирования. Такие условия можно задать в предложении HAVING. Отличие от WHERE состоит в том, что условия WHERE применяются до группировки (в них можно использовать столбцы исходных таблиц), а условия HAVING — после группировки (и в них также можно использовать столбцы таблицы-результата).

Выберем имена студентов, получивших более одной пятерки по любому предмету:

```
test=# SELECT students.name
FROM students, exams
WHERE students.s_id = exams.s_id AND exams.score = 5
GROUP BY students.name
HAVING count(*) > 1;
 name
------
 Анна
(1 row)
```

Подробнее смотрите в документации: postgrespro.ru/doc/sql-select#SQL-GROUPBY.

Изменение и удаление данных

Изменение данных в таблице выполняет оператор UPDATE, в котором указываются новые значения полей для строк, определяемых предложением WHERE (таким же, как в операторе SELECT).

Например, увеличим число лекционных часов для курса «Базы данных» в два раза:

```
test=# UPDATE courses
SET hours = hours * 2
WHERE c_no = 'CS301';

UPDATE 1
```

Подробнее смотрите в документации: postgrespro.ru/doc/sql-update.

Оператор DELETE удаляет из указанной таблицы строки, определяемые все тем же предложением WHERE:

```
test=# DELETE FROM exams WHERE score < 5;

DELETE 1
```

Транзакции

Теперь усложним схему данных и распределим студентов по группам, причем у каждой группы должен быть староста. Для этого создадим таблицу групп:

```
test=# CREATE TABLE groups(
  g_no text PRIMARY KEY,
  monitor integer NOT NULL REFERENCES students(s_id)
);
CREATE TABLE
```

Здесь мы использовали ограничение целостности NOT NULL, которое запрещает неопределенные значения.

Теперь нам нужен еще один столбец — номер группы. К счастью, в уже существующую таблицу можно добавить новый столбец:

```
test=# ALTER TABLE students
ADD g_no text REFERENCES groups(g_no);
ALTER TABLE
```

С помощью команды psql всегда можно посмотреть, какие столбцы определены в таблице:

```
test=# \d students
      Table "public.students"
   Column   |  Type   | Modifiers
------------+---------+----------
 s_id       | integer | not null
 name       | text    |
 start_year | integer |
 g_no       | text    |
 ...
```

Также можно вспомнить, какие вообще таблицы присутствуют в базе данных:

```
test=# \d
          List of relations
 Schema |   Name   | Type  | Owner
--------+----------+-------+----------
 public | courses  | table | postgres
```

```
 public | exams    | table | postgres
 public | groups   | table | postgres
 public | students | table | postgres
(4 rows)
```

Создадим теперь группу «А-101» и поместим в нее всех студентов, а старостой сделаем Анну.

Здесь есть нетривиальный момент. Нельзя создать группу, не указав старосты, но нельзя и назначить студента старостой группы, в которую он не входит, — это привело бы к появлению в базе данных логически некорректных, несогласованных данных. Эти две операции не имеют смысла по отдельности: их надо совершить одновременно. Группа операций, составляющих логически неделимую единицу работы, называется **транзакцией**.

Начнем транзакцию:

```
test=# BEGIN;

BEGIN
```

Затем добавим группу вместе со старостой. Помнить наизусть номера студенческих мы не обязаны, так что выполним запрос прямо в команде добавления строк:

```
test=*# INSERT INTO groups(g_no, monitor)
SELECT 'A-101', s_id
FROM students
WHERE name = 'Анна';

INSERT 0 1
```

«Звездочка» в приглашении напоминает о незавершенной транзакции.

Теперь откройте новое окно терминала и запустите еще один процесс `psql`: это будет сеанс, работающий параллельно с первым. Чтобы не запутаться, команды второго сеанса мы будем показывать с отступом.

Увидит ли второй сеанс сделанные изменения?

```
    postgres=# \c test
    You are now connected to database "test" as user
    "postgres".
    test=# SELECT * FROM groups;
     g_no | monitor
    ------+---------
    (0 rows)
```

Нет, не увидит, ведь транзакция еще не завершена.

Теперь переведем всех студентов в созданную группу:

```
test=*# UPDATE students SET g_no = 'A-101';
UPDATE 3
```

И снова второй сеанс видит согласованные данные, актуальные на начало еще не оконченной транзакции:

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

А теперь завершим транзакцию, зафиксировав все сделанные изменения:

```
test=*# COMMIT;
COMMIT
```

И только в этот момент второму сеансу становятся доступны все изменения, сделанные в транзакции, как будто они появились одномоментно:

```
test=# SELECT * FROM groups;
  g_no | monitor
-------+---------
 A-101 |    1451
(1 row)

test=# SELECT * FROM students;
 s_id |  name  | start_year | g_no
------+--------+------------+-------
 1451 | Анна   |       2014 | A-101
 1432 | Виктор |       2014 | A-101
 1556 | Нина   |       2015 | A-101
(3 rows)
```

СУБД дает несколько очень важных гарантий.

Во-первых, любая транзакция либо выполняется целиком (как в нашем примере), либо не выполняется никак. Если бы при выполнении одной из команд произошла ошибка, или мы сами прервали бы транзакцию командой ROLLBACK, то база данных осталась бы в том состоянии, в котором она была до команды BEGIN. Это свойство называется **атомарностью**.

Во-вторых, при фиксации изменений должны выполняться все ограничения целостности, иначе транзакция обрывается. Как в начале работы транзакции, так и в конце ее данные обязательно находятся в согласованном состоянии; это свойство так и называется — **согласованность**.

В-третьих, как мы убедились на примере, другие пользователи никогда не увидят данные, еще не зафиксированные транзакцией. Это свойство называется **изоляцией**; за счет его соблюдения СУБД способна параллельно обслуживать много сеансов, не жертвуя корректностью данных. Особенностью PostgreSQL является очень эффективная реализация изоляции: несколько сеансов могут одновременно читать и изменять данные, не блокируя друг друга. Блокировка возникает только при попытке одновременного изменения одной и той же строки несколькими разными процессами.

И в-четвертых, гарантируется **долговечность**: зафиксированные данные не пропадут даже в случае сбоя (конечно, при правильных настройках и регулярном выполнении резервного копирования).

Это крайне полезные свойства, без которых невозможно представить себе реляционную систему баз данных.

Подробнее о транзакциях см. postgrespro.ru/doc/tutorial-transactions (и еще более подробно — postgrespro.ru/doc/mvcc).

Полезные команды psql

\?	Справка по командам `psql`.
\h	Справка по SQL: список доступных команд или синтаксис конкретной команды.
\x	Переключение табличного вывода (столбцы и строки) на расширенный (каждый столбец на отдельной строке) и обратно. Удобно для просмотра нескольких «широких» строк.

\l	Список баз данных.
\du	Список пользователей.
\dt	Список таблиц.
\di	Список индексов.
\dv	Список представлений.
\df	Список функций.
\dn	Список схем.
\dx	Список установленных расширений.
\dp	Список привилегий.
\d имя	Подробная информация по конкретному объекту базы данных.
\d+ имя	Еще более подробная информация по конкретному объекту.
\timing on	Вывод времени выполнения операторов.

Заключение

Конечно, мы успели осветить только малую толику того, что необходимо знать о СУБД, но надеемся, что вы убедились: начать использовать PostgreSQL совсем нетрудно. Язык SQL позволяет формулировать запросы самой разной сложности, а PostgreSQL предоставляет качественную поддержку стандарта и эффективную реализацию. Пробуйте, экспериментируйте!

И еще одна важная команда `psql`: чтобы завершить сеанс работы, наберите

```
test=# \q
```

V Демонстрационная база данных

Описание

Общая информация

Для того чтобы научиться работать с более сложными запросами, нам понадобится более серьезная база данных — не три таблицы, а целых восемь, — наполненная правдоподобными данными.

В качестве предметной области мы выбрали авиаперевозки. Наша база данных содержит информацию о рейсах, совершенных воображаемой авиакомпанией за некоторый промежуток времени. Тем, кто хотя бы раз летал на самолете, все должно быть понятно, но в любом случае мы все объясним.

Схема нашей базы приведена на рисунке на с. 63. Мы старались сделать схему данных не слишком сложной, но и не слишком простой, чтобы обойтись без лишних деталей и в то же время отработать на ее примере интересные и осмысленные запросы.

Основной сущностью является **бронирование** (`bookings`) на одного или нескольких пассажиров, каждому из которых

выписывается отдельный **билет** (`tickets`). Пассажира как человека (возможно, летавшего нашей авиакомпанией не единожды) мы не выделяем в самостоятельную сущность, поскольку для него нет надежного уникального идентификатора. Будем считать, что уникален каждый пассажиро-рейс.

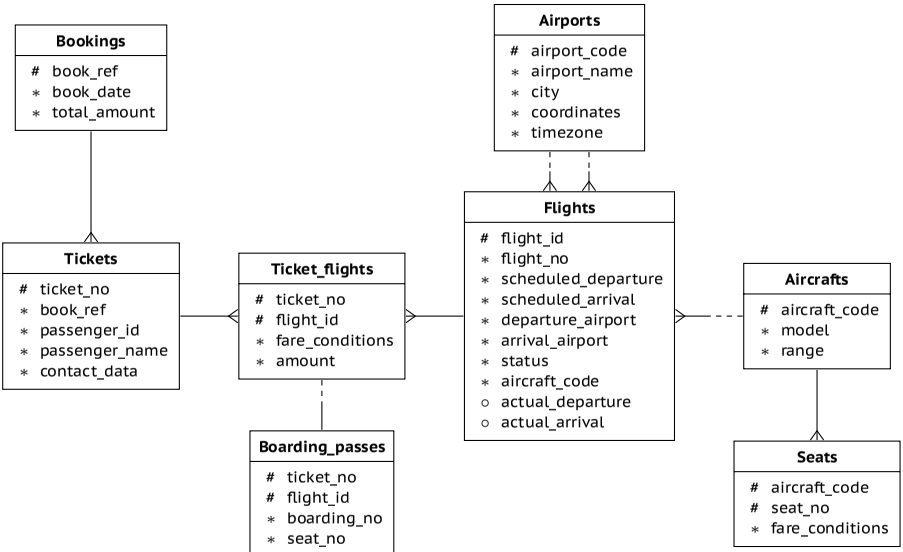
Каждому билету соответствует один или несколько **перелетов** (`ticket_flights`). Более одного перелета в билет включается тогда, когда между пунктами отправления и назначения нет прямого рейса (то есть взят билет с пересадкой) и когда они совпадают (то есть взят билет туда и обратно).

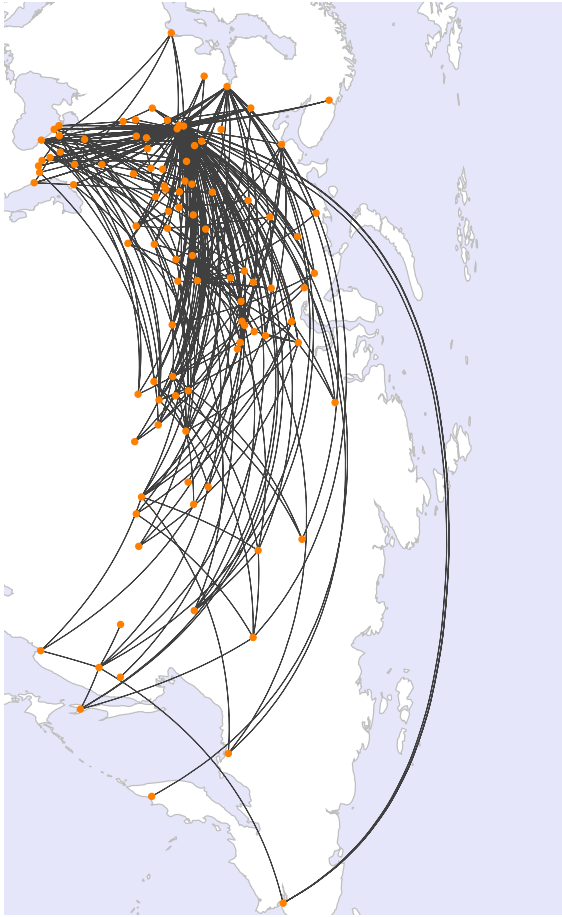
Предполагается, что все билеты в одном бронировании имеют одинаковый набор перелетов, хотя жесткого ограничения на этот счет схемой данных не предусмотрено.

Каждый **рейс** (`flights`) следует из одного **аэропорта** (`airports`) в другой. У рейсов с одинаковыми номерами совпадают пункты вылета и прилета, а отличаются даты.

При регистрации на рейс пассажиру выдается **посадочный талон** (`boarding_passes`), в котором указано место в самолете. Зарегистрироваться на рейс можно только имея на него билет. Сочетание рейса и места должно быть уникальным, чтобы исключить выдачу разных посадочных талонов на одно место.

Количество **мест** (`seats`) в самолете и их распределение по классам обслуживания зависит от модели **самолета** (`aircrafts`), выполняющего рейс. Предполагается, что у самолетов одной модели компоновка салона одинаковая. Проверка наличия в салоне места, указанного в посадочном талоне, схемой данных не предусмотрена.


Bookings
book_ref
∗ book_date
∗ total_amount
Airports
airport_code
∗ airport_name
∗ city
∗ coordinates
∗ timezone
Tickets
ticket_no
∗ book_ref
∗ passenger_id
∗ passenger_name
∗ contact_data
Ticket_flights
ticket_no
flight_id
∗ fare_conditions
∗ amount
Flights
flight_id
∗ flight_no
∗ scheduled_departure
∗ scheduled_arrival
∗ departure_airport
∗ arrival_airport
∗ status
∗ aircraft_code
○ actual_departure
○ actual_arrival
Aircrafts
aircraft_code
∗ model
∗ range
Boarding_passes
ticket_no
flight_id
∗ boarding_no
∗ seat_no
Seats
aircraft_code
seat_no
∗ fare_conditions



Далее мы подробно опишем каждую таблицу, а также дополнительные представления и функции. Точное определение любой таблицы, включая типы данных и описание столбцов, всегда можно получить командой `\d+`.

Бронирование

Намереваясь воспользоваться услугами нашей авиакомпании, пассажир заранее (`book_date`, максимум за месяц до рейса) бронирует необходимые билеты. Бронирование идентифицируется номером `book_ref` (шестизначная комбинация букв и цифр).

Поле `total_amount` хранит общую стоимость включенных в бронирование перелетов всех пассажиров.

Билет

Билет имеет уникальный номер `ticket_no`, состоящий из 13 цифр.

Билет содержит номер документа, который удостоверяет личность пассажира `passenger_id`, а также его фамилию и имя `passenger_name` и контактную информацию `contact_data`.

Заметим, что ни идентификатор пассажира, ни его имя не являются постоянными (можно поменять паспорт, можно сменить фамилию). Поэтому однозначно найти все билеты одного и того же пассажира невозможно. Для простоты можно считать, что все пассажиры уникальны.

Перелет

Перелет ассоциирует билет с рейсом, а определяется двумя их номерами.

Для каждого перелета указываются его стоимость `amount` и класс обслуживания `fare_conditions`.

Рейс

Уникальный идентификатор может быть естественным (если в нем есть признаки объектов реального мира) или суррогатным (если генерируется системой, обычно из возрастающей последовательности чисел).

Естественный ключ таблицы рейсов состоит из номера рейса `flight_no` и даты отправления `scheduled_departure`. Чтобы сделать внешние ключи на эту таблицу компактнее, в качестве первичного используется суррогатный ключ `flight_id`.

Рейс всегда соединяет две точки — аэропорты вылета `departure_airport` и прибытия `arrival_airport`.

Такое понятие, как «рейс с пересадками», отсутствует: если между аэропортами нет прямого рейса, в билет просто включаются все нужные рейсы.

У каждого рейса есть запланированные дата и время вылета `scheduled_departure` и прибытия `scheduled_arrival`. Реальное время вылета `actual_departure` и реальное время прибытия `actual_arrival` могут отличаться; чаще не сильно, но иногда задержка рейса может составить несколько часов.

Статус рейса `status` может принимать следующие значения:

- Scheduled
  Рейс доступен для бронирования. Значение устанавливается за месяц до плановой даты вылета одновременно с появлением в базе данных записи о рейсе.
- On Time
  Рейс доступен для регистрации (за сутки до плановой даты вылета) и не задержан.
- Delayed
  Рейс доступен для регистрации (за сутки до плановой даты вылета), но задержан.
- Departed
  Самолет уже вылетел и находится в воздухе.
- Arrived
  Самолет прибыл в пункт назначения.
- Cancelled
  Рейс отменен.

Аэропорт

Каждый аэропорт идентифицируется трехбуквенным кодом `airport_code` и имеет название `airport_name`.

Название города `city` указывается как атрибут аэропорта; отдельной сущности для него не предусмотрено. Нужно оно в том числе для определения аэропортов одного города. Также указываются координаты (долгота и широта) `coordinates` и часовой пояс `timezone`.

Посадочный талон

При регистрации на рейс, которая возможна за сутки до плановой даты отправления, пассажиру выдается посадочный талон. Он идентифицируется так же, как и перелет, — номером билета и номером рейса.

Посадочным талонам присваиваются последовательные номера `boarding_no` в порядке регистрации пассажиров на рейс (этот номер будет уникальным только в пределах данного рейса). В посадочном талоне указывается номер места `seat_no`.

Самолет

Каждая модель воздушного судна идентифицируется своим трехзначным кодом `aircraft_code`. Указывается также название модели `model` и максимальная дальность полета в километрах `range`.

Место

Набор мест определяет схему салона каждой модели. Каждое место определяется номером `seat_no` и имеет закрепленный за ним класс обслуживания `fare_conditions` — Economy, Comfort или Business.

Представление для рейсов

Над таблицей `flights` создано представление `flights_v`. Представления можно использовать в запросах так же, как

таблицы, но они не хранят данные, а выполняют некоторый запрос. Определение представления и запомненный им запрос можно посмотреть следующей командой psql:

```
postgres=# \d+ flights_v
```

Это представление добавляет следующую информацию:

- расшифровку данных об аэропорте вылета `departure_airport`, `departure_airport_name`, `departure_city`;
- расшифровку данных об аэропорте прибытия `arrival_airport`, `arrival_airport_name`, `arrival_city`;
- местное время вылета `scheduled_departure_local`, `actual_departure_local`;
- местное время прибытия `scheduled_arrival_local`, `actual_arrival_local`;
- продолжительность полета `scheduled_duration`, `actual_duration`.

Представление для маршрутов

Таблица рейсов содержит избыточность: из нее можно было бы выделить информацию о маршруте (номер рейса, аэропорты отправления и назначения, модель самолета), не зависящую от конкретных дат рейсов.

Именно такая информация и составляет представление `routes`. Кроме того, это представление показывает массив дней недели `days_of_week`, по которым совершаются полеты, и плановую продолжительность рейса `duration`.

Функция now

Демонстрационная база содержит временной «срез» данных — так, как будто в некоторый момент была сделана резервная копия реальной системы. Например, если некоторый рейс имеет статус Departed, это означает, что в момент резервного копирования самолет находился в воздухе.

Позиция «среза» сохранена в функции `bookings.now`. Ею можно пользоваться в запросах там, где в обычных условиях использовалась бы функция `now`.

Кроме того, значение этой функции определяет версию демонстрационной базы данных. Актуальная версия на момент подготовки этого выпуска книги — от 15.08.2017.

Установка

Установка с сайта

База данных доступна в трех версиях, которые отличаются только объемом:

- edu.postgrespro.ru/demo-small.zip — небольшая, данные по полетам за один месяц (файл 21 МБ, размер БД 280 МБ),
- edu.postgrespro.ru/demo-medium.zip — средняя, данные по полетам за три месяца (файл 62 МБ, размер БД 702 МБ),
- edu.postgrespro.ru/demo-big.zip — большая, данные по полетам за один год (файл 232 МБ, размер БД 2638 МБ).

Тренироваться писать запросы можно и на небольшой базе, а для погружения в вопросы оптимизации лучше подойдет большая — тогда сразу будет видно, как запросы ведут себя на серьезных объемах данных.

Файлы содержат логическую резервную копию базы `demo`, созданную утилитой `pg_dump`. Имейте в виду, что если у вас уже есть база данных с именем `demo`, она будет удалена и создана заново при восстановлении из резервной копии. Владельцем базы `demo` станет тот пользователь СУБД, который выполнял восстановление.

Чтобы установить демонстрационную базу данных в операционной системе Linux, скачайте один из файлов, предварительно переключившись на пользователя `postgres`. Например, небольшая база скачивается так:

```
$ sudo su - postgres
```

```
$ wget https://edu.postgrespro.ru/demo-small.zip
```

Затем выполните команду:

```
$ zcat demo-small.zip | psql
```

В операционной системе Windows любым веб-браузером скачайте с сайта файл edu.postgrespro.ru/demo-small.zip, после чего дважды кликните на нем, чтобы открыть архив, и затем скопируйте файл `demo-small-20170815.sql` в каталог `C:\Program Files\PostgreSQL\14`.

Программа pgAdmin (о которой пойдет речь на с. 111) не позволяет восстановить базу данных из такой резервной копии. Поэтому запустите psql (ярлык «SQL Shell (psql)») и выполните команду:

```
postgres# \i demo-small-20170815.sql
```

Если файл не будет найден, проверьте у ярлыка свойство «Start in» («Рабочая папка») — файл должен находиться именно в этом каталоге.

Примеры запросов

Пара слов о схеме

Установка завершена. Запустите psql и подключитесь к демонстрационной базе:

```
postgres=# \c demo
You are now connected to database "demo" as user
"postgres".
```

Все интересующие нас объекты находятся в схеме bookings. При подключении к базе данных эта схема используется автоматически, так что явно ее указывать не нужно:

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

Но при вызове функции bookings.now — нужно, так как существует стандартная функция now:

```
demo=# SELECT bookings.now();

          now
------------------------
 2017-08-15 18:00:00+03
(1 row)
```

Как вы уже заметили, названия моделей самолетов, аэропортов и городов выводятся по-русски:

```
demo=# SELECT airport_code, city
FROM airports LIMIT 5;

 airport_code |           city
--------------+--------------------------
 YKS          | Якутск
 MJZ          | Мирный
 KHV          | Хабаровск
 PKC          | Петропавловск-Камчатский
 UUS          | Южно-Сахалинск
(5 rows)
```

А при значении en параметра bookings.lang, определяющего язык, — по-английски:

```
demo=# ALTER DATABASE demo SET bookings.lang = en;

ALTER DATABASE
```

Мы сменили язык на уровне базы данных; теперь нужно подключиться заново.

```
demo=# \c

You are now connected to database "demo" as user
"postgres".
```

```
demo=# SELECT airport_code, city
FROM airports
LIMIT 5;

 airport_code |       city
--------------+-------------------
 YKS          | Yakutsk
 MJZ          | Mirnyj
 KHV          | Khabarovsk
 PKC          | Petropavlovsk
 UUS          | Yuzhno-sakhalinsk
(5 rows)
```

Чтобы изучить механизм смены языка, посмотрите определение `aircrafts` или `airports` командой psql `\d+`.

Подробнее про управление схемами: postgrespro.ru/doc/ddl-schemas.

Про установку параметров сервера — postgrespro.ru/doc/config-setting.

Простые запросы

Рассмотрим несколько задач для этой схемы — от самых простых к довольно сложным. К большинству задач приводятся решения, но лучше сначала попробовать написать запрос, не подглядывая в ответ, ведь иначе научиться языку SQL не получится.

Задача. Кто летел позавчера рейсом Москва (SVO) — Новосибирск (OVB) на месте 1A и когда он забронировал себе билет?

Решение. «Позавчера» отсчитывается от `booking.now`, а не от текущей даты.

```
SELECT t.passenger_name,
       b.book_date
FROM   bookings b
       JOIN tickets t
         ON t.book_ref = b.book_ref
       JOIN boarding_passes bp
         ON bp.ticket_no = t.ticket_no
       JOIN flights f
         ON f.flight_id = bp.flight_id
WHERE  f.departure_airport = 'SVO'
AND    f.arrival_airport = 'OVB'
AND    f.scheduled_departure::date =
       bookings.now()::date - INTERVAL '2 day'
AND    bp.seat_no = '1A';
```

Задача. Сколько мест осталось незанятыми вчера на рейсе PG0404?

Решение. Задачу можно решить несколькими способами. Как вариант, при помощи конструкции NOT EXISTS найдем места без посадочных талонов:

```
SELECT count(*)
FROM   flights f
       JOIN seats s
         ON s.aircraft_code = f.aircraft_code
WHERE  f.flight_no = 'PG0404'
AND    f.scheduled_departure::date =
       bookings.now()::date - INTERVAL '1 day'
AND    NOT EXISTS (
         SELECT NULL
         FROM   boarding_passes bp
         WHERE  bp.flight_id = f.flight_id
         AND    bp.seat_no = s.seat_no
       );
```

В другом варианте прибегнем к вычитанию множеств. Разные решения дают одинаковый результат, но иногда отличаются по производительности, так что, если она важна, стоит это учитывать.

```
SELECT count(*)
FROM   (
  SELECT s.seat_no
  FROM   seats s
  WHERE  s.aircraft_code = (
    SELECT aircraft_code
    FROM   flights
    WHERE  flight_no = 'PG0404'
    AND    scheduled_departure::date =
           bookings.now()::date - INTERVAL '1 day'
  )
  EXCEPT
  SELECT bp.seat_no
  FROM   boarding_passes bp
  WHERE  bp.flight_id = (
    SELECT flight_id
    FROM   flights
    WHERE  flight_no = 'PG0404'
    AND    scheduled_departure::date =
           bookings.now()::date - INTERVAL '1 day'
  )
) t;
```

Задача. На каких рейсах происходили самые длительные задержки? Выведите список из десяти рейсов, задержанных на самые длительные сроки.

Решение. Учитываем только состоявшиеся вылеты.

```
SELECT   f.flight_no,
         f.scheduled_departure,
         f.actual_departure,
         f.actual_departure - f.scheduled_departure
         AS delay
FROM     flights f
WHERE    f.actual_departure IS NOT NULL
ORDER BY f.actual_departure - f.scheduled_departure
         DESC
LIMIT 10;
```

То же самое условие можно записать и на основе столбца `status`, перечислив все подходящие статусы. А можно

обойтись и вовсе без условия WHERE, указав порядок сортировки DESC NULLS LAST, чтобы неопределенные значения попали не в начало, а в конец выборки.

Агрегатные функции

Задача. Какова минимальная и максимальная продолжительность полета для каждого из возможных рейсов из Москвы в Санкт-Петербург, и сколько раз вылет рейса был задержан больше, чем на час?

Решение. Здесь удобно воспользоваться готовым представлением flights_v, чтобы не выписывать соединения необходимых таблиц. В запросе учитываем только уже выполненные рейсы.

```
SELECT   f.flight_no,
         f.scheduled_duration,
         min(f.actual_duration),
         max(f.actual_duration),
         sum(CASE WHEN f.actual_departure >
                       f.scheduled_departure +
                       INTERVAL '1 hour'
                  THEN 1 ELSE 0
             END) delays
FROM     flights_v f
WHERE    f.departure_city = 'Москва'
AND      f.arrival_city = 'Санкт-Петербург'
AND      f.status = 'Arrived'
GROUP BY f.flight_no,
         f.scheduled_duration;
```

Задача. Найдите самых дисциплинированных пассажиров, которые зарегистрировались на все рейсы первыми. Учтите только тех пассажиров, которые совершали минимум два рейса.

Решение. Используем тот факт, что номера посадочных талонов выдаются в порядке регистрации.

```
SELECT   t.passenger_name,
         t.ticket_no
FROM     tickets t
         JOIN boarding_passes bp
           ON bp.ticket_no = t.ticket_no
GROUP BY t.passenger_name,
         t.ticket_no
HAVING   max(bp.boarding_no) = 1
AND      count(*) > 1;
```

Задача. Сколько пассажиров приходится на одно бронирование?

Решение. Сосчитаем сначала количество пассажиров в каждом бронировании, а затем — количество бронирований с каждым вариантом количества пассажиров.

```
SELECT   tt.cnt,
         count(*)
FROM     (
           SELECT   t.book_ref,
                    count(*) cnt
           FROM     tickets t
           GROUP BY t.book_ref
         ) tt
GROUP BY tt.cnt
ORDER BY tt.cnt;
```

Оконные функции

Задача. Для каждого билета выведите входящие в него перелеты вместе с запасом времени на пересадку на следующий рейс. Ограничьте выборку теми билетами, которые были забронированы семью днями ранее.

Решение. Чтобы не обращаться к одним и тем же данным два раза, воспользуемся оконными функциями.

```
SELECT tf.ticket_no,
       f.departure_airport,
       f.arrival_airport,
       f.scheduled_arrival,
       lead(f.scheduled_departure) OVER w
       AS next_departure,
       lead(f.scheduled_departure) OVER w -
         f.scheduled_arrival
       AS gap
FROM   bookings b
       JOIN tickets t
         ON t.book_ref = b.book_ref
       JOIN ticket_flights tf
         ON tf.ticket_no = t.ticket_no
       JOIN flights f
         ON tf.flight_id = f.flight_id
WHERE  b.book_date =
       bookings.now()::date - INTERVAL '7 day'
WINDOW w AS (
         PARTITION BY tf.ticket_no
         ORDER BY f.scheduled_departure);
```

Как видно, в некоторых случаях выведенный запас времени на пересадку составляет до нескольких дней, так как билеты «туда и обратно» учитываются наравне с билетами в один конец, а время пребывания в пункте назначения — таким же образом, как и время пересадки. Используя решение одной из задач в разделе «Массивы», можно учесть этот факт в запросе.

Задача. Какие сочетания имен и фамилий встречаются чаще всего и какую долю от числа всех пассажиров они составляют?

Решение. Для подсчета общего числа пассажиров используется оконная функция.

```
SELECT   passenger_name,
         round( 100.0 * cnt / sum(cnt) OVER (), 2)
         AS percent
FROM     (
           SELECT   passenger_name,
                    count(*) cnt
           FROM     tickets
           GROUP BY passenger_name
         ) t
ORDER BY percent DESC;
```

Задача. Решите предыдущую задачу отдельно для имен и отдельно для фамилий.

Решение. Приведем вариант для подсчета имен. Решение для фамилий будет отличаться только подзапросом p.

```
WITH p AS (
  SELECT left(passenger_name,
              position(' ' IN passenger_name))
         AS passenger_name
  FROM   tickets
)
SELECT   passenger_name,
         round( 100.0 * cnt / sum(cnt) OVER (), 2)
         AS percent
FROM     (
           SELECT   passenger_name,
                    count(*) cnt
           FROM     p
           GROUP BY passenger_name
         ) t
ORDER BY percent DESC;
```

Вывод такой: не объединяйте в одном текстовом поле несколько значений, если собираетесь работать с ними по отдельности. По-научному это называется «первой нормальной формой».

Задача. В билете не указывается явно, в один он конец или туда и обратно. Но это можно определить, сравнив первый пункт отправления с последним пунктом назначения. Для каждого билета выведите аэропорты отправления и назначения без учета пересадок и с указанием того, взят он в один конец или туда и обратно.

Решение. Пожалуй, проще всего свернуть список аэропортов на пути следования в массив с помощью агрегатной функции `array_agg` и работать с ним.

В качестве аэропорта назначения для билетов «туда и обратно» выбираем средний элемент массива, предполагая, что пути «туда» и «обратно» имеют одинаковое число пересадок.

```
WITH t AS (
  SELECT ticket_no,
         a,
         a[1]                  departure,
         a[cardinality(a)]     last_arrival,
         a[cardinality(a)/2+1] middle
  FROM (
    SELECT   t.ticket_no,
             array_agg( f.departure_airport
               ORDER BY f.scheduled_departure) ||
            (array_agg( f.arrival_airport
               ORDER BY f.scheduled_departure DESC)
            )[1] AS a
    FROM     tickets t
             JOIN ticket_flights tf
               ON tf.ticket_no = t.ticket_no
             JOIN flights f
               ON f.flight_id = tf.flight_id
    GROUP BY t.ticket_no
  ) t
)
```

```
SELECT t.ticket_no,
       t.a,
       t.departure,
       CASE
         WHEN t.departure = t.last_arrival
           THEN t.middle
         ELSE t.last_arrival
       END arrival,
       (t.departure = t.last_arrival) return_ticket
FROM   t;
```

В таком варианте таблица билетов просматривается только один раз. Массив аэропортов выводится исключительно для наглядности; на большом объеме данных имеет смысл убрать его из запроса, поскольку лишние данные могут не лучшим образом сказаться на производительности.

Задача. Найдите билеты, взятые туда и обратно, в которых путь «туда» не совпадает с путем «обратно».

Задача. Найдите такие пары аэропортов, рейсы между которыми в одну и в другую стороны отправляются по разным дням недели.

Решение. Часть задачи по построению массива дней недели уже фактически решена в представлении `routes`. Остается только найти пересечение массивов с помощью оператора && и убедиться, что оно пусто:

```
SELECT r1.departure_airport,
       r1.arrival_airport,
       r1.days_of_week dow,
       r2.days_of_week dow_back
FROM   routes r1
       JOIN routes r2
         ON r1.arrival_airport = r2.departure_airport
        AND r1.departure_airport = r2.arrival_airport
WHERE  NOT (r1.days_of_week && r2.days_of_week);
```

Задача. Как добраться из Усть-Кута (UKX) в Нерюнгри (CNN) с минимальным числом пересадок и сколько времени придется провести в воздухе?

Решение. Здесь фактически нужно найти кратчайший путь в графе, что делается рекурсивным запросом.

```
WITH RECURSIVE p(
  last_arrival,
  destination,
  hops,
  flights,
  flight_time,
  found
) AS (
  SELECT a_from.airport_code,
         a_to.airport_code,
         array[a_from.airport_code],
         array[]::char(6)[],
         interval '0',
         a_from.airport_code = a_to.airport_code
  FROM   airports a_from,
         airports a_to
  WHERE  a_from.airport_code = 'UKX'
  AND    a_to.airport_code = 'CNN'
  UNION ALL
  SELECT r.arrival_airport,
         p.destination,
         (p.hops || r.arrival_airport)::char(3)[],
         (p.flights || r.flight_no)::char(6)[],
         p.flight_time + r.duration,
         bool_or(r.arrival_airport = p.destination)
           OVER ()
  FROM   p
         JOIN routes r
           ON r.departure_airport = p.last_arrival
  WHERE  NOT r.arrival_airport = ANY(p.hops)
  AND    NOT p.found
)
```

```
SELECT hops,
       flights,
       flight_time
FROM   p
WHERE  p.last_arrival = p.destination;
```

Этот запрос разбирается детально, шаг за шагом, в статье habr.com/company/postgrespro/blog/318398, так что здесь дадим только краткие комментарии.

Зацикливание предотвращается проверкой по массиву пересадок `hops`, который строится в процессе выполнения запроса.

Обратите внимание, что поиск происходит «в ширину», то есть первый же найденный путь и будет кратчайшим по числу пересадок. Чтобы не перебирать остальные пути (которых может быть очень много и которые заведомо длиннее уже найденного), используется признак «маршрут найден» (`found`). Он рассчитывается с помощью оконной функции `bool_or`.

Поучительно сравнить скорость выполнения этого запроса с более простым вариантом без флага.

Подробно про рекурсивные запросы можно посмотреть в документации: postgrespro.ru/doc/queries-with.

Задача. Какое максимальное число пересадок может потребоваться, чтобы добраться из одного любого аэропорта в любой другой?

Решение. В качестве основы решения можно взять предыдущий запрос. Но теперь начальная итерация должна содержать не одну пару аэропортов, а все возможные пары: каждый аэропорт соединяем с каждым. Для всех таких пар мы ищем кратчайший путь, а затем выбираем максимальный из них.

Конечно, так можно поступить только если граф маршрутов является связным, но в нашей демонстрационной базе это условие выполнено.

В этом запросе также используется признак «маршрут найден», но здесь его необходимо рассчитывать отдельно для каждой пары аэропортов.

```
WITH RECURSIVE p(
  departure,
  last_arrival,
  destination,
  hops,
  found
) AS (
  SELECT a_from.airport_code,
         a_from.airport_code,
         a_to.airport_code,
         array[a_from.airport_code],
         a_from.airport_code = a_to.airport_code
  FROM   airports a_from,
         airports a_to
  UNION ALL
  SELECT p.departure,
         r.arrival_airport,
         p.destination,
         (p.hops || r.arrival_airport)::char(3)[],
         bool_or(r.arrival_airport = p.destination)
           OVER (PARTITION BY p.departure,
                              p.destination)
  FROM   p
         JOIN routes r
           ON r.departure_airport = p.last_arrival
  WHERE  NOT r.arrival_airport = ANY(p.hops)
  AND    NOT p.found
)
SELECT max(cardinality(hops)-1)
FROM   p
WHERE  p.last_arrival = p.destination;
```

86 **Задача.** Найдите кратчайший путь, ведущий из Усть-Кута (UKX) в Нерюнгри (CNN), с точки зрения чистого времени перелетов (игнорируя время пересадок).

Подсказка: этот путь может оказаться не оптимальным по числу пересадок.

Решение. Для предотвращения зацикливания используется конструкция CYCLE, появившаяся в PostgreSQL 14.

```
WITH RECURSIVE p(
  last_arrival,
  destination,
  flights,
  flight_time,
  min_time
) AS (
  SELECT a_from.airport_code,
         a_to.airport_code,
         array[]::char(6)[],
         interval '0',
         NULL::interval
  FROM   airports a_from,
         airports a_to
  WHERE  a_from.airport_code = 'UKX'
  AND    a_to.airport_code = 'CNN'
  UNION ALL
  SELECT r.arrival_airport,
         p.destination,
         (p.flights || r.flight_no)::char(6)[],
         p.flight_time + r.duration,
         least(
           p.min_time,
           min(p.flight_time + r.duration)
           FILTER (
             WHERE r.arrival_airport = p.destination
           ) OVER ()
         )
  FROM   p
         JOIN routes r
           ON r.departure_airport = p.last_arrival
```

```
  WHERE  p.flight_time + r.duration
         < coalesce(
             p.min_time,
             INTERVAL '1 year'
           )
)
CYCLE last_arrival SET is_cycle USING hops
SELECT hops,
       flights,
       flight_time
FROM   (
         SELECT hops,
                flights,
                flight_time,
                min(min_time) OVER () min_time
         FROM   p
         WHERE  p.last_arrival = p.destination
       ) t
WHERE  flight_time = min_time;
```

Функции и расширения

Задача. Найдите расстояние между Калининградом (KGD) и Петропавловском-Камчатским (PKC).

Решение. В таблице `airports` имеются координаты аэропортов. Чтобы точно вычислить расстояние между сильно удаленными точками, нужно учесть непростую форму Земли. Лучше всего с этой задачей справляется расширение PostGIS, умеющее аппроксимировать земную поверхность геоидом.

Но для наших целей подойдет и простая сферическая модель. Воспользуемся расширением `earthdistance` и затем переведем результат из миль в километры.

```
CREATE EXTENSION IF NOT EXISTS cube;

CREATE EXTENSION IF NOT EXISTS earthdistance;
```

```
SELECT round(
         (a_from.coordinates <@> a_to.coordinates) *
         1.609344
       )
FROM   airports a_from,
       airports a_to
WHERE  a_from.airport_code = 'KGD'
AND    a_to.airport_code = 'PKC';
```

Задача. Нарисуйте граф рейсов между аэропортами.

VI PostgreSQL для приложения

Отдельный пользователь

В предыдущей главе мы подключались к серверу баз данных как пользователь `postgres`, единственный существующий сразу после установки СУБД. Но `postgres` обладает правами суперпользователя, а через приложение с этими правами подключаться к базе не следует. Лучше создать нового пользователя и сделать его владельцем отдельной базы данных — тогда его права будут ограничены этой базой.

```
postgres=# CREATE USER app PASSWORD 'p@ssw0rd';
CREATE ROLE
postgres=# CREATE DATABASE appdb OWNER app;
CREATE DATABASE
```

Подробнее про пользователей и разграничение доступа смотрите в документации: postgrespro.ru/doc/user-manag и postgrespro.ru/doc/ddl-priv.

Чтобы подключиться к новой базе данных и работать с ней от имени созданного пользователя, выполните:

```
postgres=# \c appdb app localhost 5432

Password for user app: ***
You are now connected to database "appdb" as user
"app" on host "127.0.0.1" at port "5432".

appdb=>
```

В команде указываются последовательно имя базы данных (appdb), имя пользователя (app), узел (localhost или 127.0.0.1) и номер порта (5432).

Обратите внимание, что в подсказке-приглашении изменилось не только имя базы данных: вместо «решетки» теперь отображается символ «больше». Решетка указывает на роль суперпользователя по аналогии с пользователем `root` в операционной системе Unix.

Со своей базой данных пользователь `app` работает без ограничений. Например, в ней можно создать таблицу:

```
appdb=> CREATE TABLE greeting(s text);

CREATE TABLE

appdb=> INSERT INTO greeting VALUES ('Привет, мир!');

INSERT 0 1
```

Удаленное подключение

В нашем примере клиент и СУБД находятся на одном и том же компьютере. Разумеется, можно установить PostgreSQL на выделенный сервер, а подключаться к нему с другой машины (например, с сервера приложений). В этом случае вместо localhost надо указать адрес вашего сервера

СУБД. Но этого недостаточно: по умолчанию из соображений безопасности PostgreSQL допускает только локальные подключения.

Чтобы подключиться к базе данных снаружи, необходимо отредактировать два файла.

Во-первых, `postgresql.conf` — **файл основных настроек** (обычно располагается в каталоге баз данных).

Найдите строку, определяющую, какие сетевые интерфейсы слушает PostgreSQL:

```
#listen_addresses = 'localhost'
```

и замените ее на:

```
listen_addresses = '*'
```

Во-вторых, `pg_hba.conf` — **файл с настройками аутентификации**.

Когда клиент подключается к серверу, в этом файле выбирается первая сверху строка, соответствующая соединению по четырем параметрам: типу соединения, имени базы данных, имени пользователя и IP-адресу клиента. В той же строке написано, как пользователь должен подтвердить, что он действительно тот, за кого себя выдает.

Например, в ОС Debian и Ubuntu этот файл содержит, среди прочих, такую настройку (верхняя строка, начинающаяся с «решетки», считается комментарием):

```
# TYPE  DATABASE  USER  ADDRESS       METHOD
local   all       all                 peer
```

Она говорит, что локальные подключения (`local`) к любой базе данных (`all`) от лица любого пользователя (`all`) должны проверяться методом `peer` (IP-адрес для локальных соединений, конечно, не указывается).

Метод `peer` означает, что PostgreSQL запрашивает имя текущего пользователя у операционной системы и считает, что ОС уже выполнила необходимую проверку (спросила у пользователя пароль). Поэтому в Linux-подобных операционных системах при подключении к локальному серверу ввод пароля обычно не требуется.

А вот в Windows локальные соединения не поддерживаются, и там настройка выглядит следующим образом:

```
# TYPE  DATABASE  USER  ADDRESS       METHOD
host    all       all   127.0.0.1/32  md5
```

Она означает, что сетевые подключения (`host`) к любой базе данных (`all`) от лица любого пользователя (`all`) с локального адреса (`127.0.0.1`) должны проверяться методом `md5`. Этот метод подразумевает ввод пароля пользователем.

Чтобы дать пользователю `app` доступ к базе данных `appdb` с любого адреса при указании верного пароля, допишите в конец `pg_hba.conf` следующую строку:

```
host    appdb     app   all           md5
```

После внесения изменений в конфигурационные файлы не забудьте попросить сервер перечитать настройки:

```
postgres=# SELECT pg_reload_conf();
```

Подробнее о настройках аутентификации: postgrespro.ru/doc/client-authentication.

Проверка связи

Чтобы подключиться к PostgreSQL из своей программы, найдите подходящую библиотеку и установите драйвер СУБД. Драйвер обычно представляет собой обертку для `libpq` — штатной реализации клиент-серверного протокола PostgreSQL, — хотя встречаются и другие варианты. Библиотека обеспечивает удобный для прикладной разработки доступ к низкоуровневым возможностям протокола.

Ниже приведены простые примеры кода на нескольких популярных языках программирования. Они помогут быстро проверить соединение с установленной и настроенной базой данных.

Приведенные программы содержат только минимально необходимый код для выполнения простого запроса к базе данных и вывода полученного результата; ничего сверх этого, включая обработку ошибок, в них не предусмотрено. Не следует считать эти фрагменты кода примером для подражания.

Для корректного отображения символов различных алфавитов под Windows может потребоваться в окне Command Prompt сменить шрифт на TrueType (например, «Lucida Console» или «Consolas») и поменять кодовую страницу. Так, для вывода на русском языке дайте следующие команды:

```
C:\> chcp 1251

Active code page: 1251

C:\> set PGCLIENTENCODING=WIN1251
```

PHP

В языке PHP работа с PostgreSQL организована с помощью специального расширения. Под Linux, кроме самого PHP, потребуется пакет с этим расширением:

```
$ sudo apt-get install php-cli php-pgsql
```

PHP для Windows доступен на сайте windows.php.net/download. Расширение для PostgreSQL уже входит в комплект, но в файле `php.ini` необходимо найти и раскомментировать (убрать точку с запятой) строку:

```
;extension=php_pgsql.dll
```

Пример программы (`test.php`):

```
<?php
  $conn = pg_connect('host=localhost port=5432 ' .
                     'dbname=appdb user=app ' .
                     'password=p@ssw0rd') or die;
  $query = pg_query('SELECT * FROM greeting') or die;
  while ($row = pg_fetch_array($query)) {
    echo $row[0].PHP_EOL;
  }
  pg_free_result($query);
  pg_close($conn);
?>
```

Выполняем:

```
$ php test.php
Привет, мир!
```

Расширение для PostgreSQL описано в документации: php.net/manual/ru/book.pgsql.php.

Perl

В языке Perl работа с базами данных организована с помощью интерфейса DBI. Сам Perl предустановлен в Debian и Ubuntu, так что дополнительно нужен только драйвер:

```
$ sudo apt-get install libdbd-pg-perl
```

Существует несколько сборок Perl для Windows, они перечислены на сайте perl.org/get.html. Популярные сборки ActiveState Perl и Strawberry Perl уже включают необходимый для PostgreSQL драйвер.

Пример программы (`test.pl`):

```
use DBI;
use open ':std', ':utf8';
my $conn = DBI->connect(
  'dbi:Pg:dbname=appdb;host=localhost;port=5432',
  'app',
  'p@ssw0rd') or die;
my $query = $conn->prepare('SELECT * FROM greeting');
$query->execute() or die;
while (my @row = $query->fetchrow_array()) {
  print @row[0]."\n";
}
$query->finish();
$conn->disconnect();
```

Выполняем:

```
$ perl test.pl
Привет, мир!
```

Интерфейс описан в документации:
metacpan.org/pod/DBD::Pg.

Python

В языке Python для работы с PostgreSQL обычно используется библиотека psycopg (название произносится как «сайко-пи-джи»).

В современных версиях Debian и Ubuntu язык Python версии 3 предустановлен, так что нужен только драйвер:

```
$ sudo apt-get install python3-psycopg2
```

Python для операционной системы Windows можно взять с сайта python.org. Библиотека psycopg доступна на сайте проекта initd.org/psycopg (выберите версию, соответствующую установленной версии Python). Там же находится необходимая документация.

Пример программы (`test.py`):

```
import psycopg2
conn = psycopg2.connect(
  host='localhost',
  port='5432',
  database='appdb',
  user='app',
  password='p@ssw0rd')
cur = conn.cursor()
cur.execute('SELECT * FROM greeting')
rows = cur.fetchall()
for row in rows:
  print(row[0])
conn.close()
```

Выполняем:

```
$ python3 test.py
Привет, мир!
```

Java

В языке Java работа с базами данных организована через интерфейс JDBC. Устанавливаем Java SE 11; дополнительно нам потребуется пакет с драйвером JDBC:

```
$ sudo apt-get install openjdk-11-jdk
$ sudo apt-get install libpostgresql-jdbc-java
```

JDK для ОС Windows можно скачать с сайта oracle.com/technetwork/java/javase/downloads. Драйвер JDBC доступен на сайте jdbc.postgresql.org (выберите версию, которая соответствует установленной версии JDK). Там же находится и документация.

Пример программы (`Test.java`):

```
import java.sql.*;
public class Test {
  public static void main(String[] args)
  throws SQLException {
    Connection conn = DriverManager.getConnection(
      "jdbc:postgresql://localhost:5432/appdb",
      "app", "p@ssw0rd");
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery(
      "SELECT * FROM greeting");
    while (rs.next()) {
      System.out.println(rs.getString(1));
    }
    rs.close();
    st.close();
    conn.close();
  }
}
```

Компилируем и выполняем программу, указывая в ключе путь к классу-драйверу JDBC (в Windows пути разделяются не двоеточием, а точкой с запятой):

```
$ javac Test.java
$ java -cp .:/usr/share/java/postgresql-jdbc4.jar \
Test
Привет, мир!
```

Резервное копирование

Хотя в созданной нами базе данных всего одна таблица, не помешает задуматься и о сохранности данных. Пока данных в приложении немного, сделать резервную копию проще всего утилитой pg_dump:

```
$ pg_dump appdb > appdb.dump
```

В получившемся файле appdb.dump будут содержаться обычные команды SQL, создающие и заполняющие данными все объекты базы appdb. Этот файл можно подать на вход psql и восстановить содержимое базы данных. Например, можно создать новую БД и загрузить данные туда:

```
$ createdb appdb2
$ psql -d appdb2 -f appdb.dump
```

Именно в таком виде распространяется демобаза, с которой мы познакомились в предыдущей главе.

У pg_dump много возможностей, с которыми стоит познакомиться: postgrespro.ru/doc/app-pgdump. Некоторые из них доступны только когда данные выгружаются в специальном внутреннем формате; в таком случае для восстановления нужно использовать не psql, а утилиту pg_restore.

В любом случае pg_dump выгружает содержимое только одной базы данных. Если требуется сделать резервную копию кластера, включая все базы данных, пользователей и табличные пространства, надо воспользоваться другой, хотя и похожей, командой pg_dumpall.

Для больших серьезных проектов требуется продуманная стратегия периодического резервного копирования. Для этого лучше подойдет физическая «двоичная» копия кластера, которую создает утилита pg_basebackup:

```
$ pg_basebackup -D backup
```

Такая команда создаст резервную копию всего кластера баз данных в каталоге backup. Чтобы восстановить систему из созданной копии, достаточно скопировать ее в каталог баз данных и запустить сервер.

Подробнее про все доступные средства резервного копирования и восстановления можно почитать в документации: postgrespro.ru/doc/backup, а также в учебном курсе DBA3 (с. 155).

Штатные средства PostgreSQL позволяют сделать практически все, что нужно, однако требуют выполнения многочисленных шагов, нуждающихся в автоматизации. Поэтому многие компании создают собственные инструменты для резервного копирования и восстановления. Такой инструмент — **pg_probackup** — есть и у нашей компании Postgres Professional. Он распространяется свободно и позволяет выполнять инкрементальное резервное копирование на уровне страниц, контролировать целостность данных, работать с большими объемами информации за счет параллелизма и сжатия, а также реализовывать различные стратегии резервного копирования. Полная документация доступна по адресу postgrespro.ru/doc/app-pgprobackup.

Что дальше?

Теперь вы готовы к разработке приложения. По отношению к базе данных оно всегда будет состоять из двух частей: серверной и клиентской. Серверная часть — это все, что относится к СУБД: таблицы, индексы, представления, триггеры, хранимые функции и процедуры. Клиентская — все, что работает вне СУБД и подключается к ней; с точки зрения базы данных не важно, будет это «толстый» клиент или сервер приложений.

Важный вопрос, на который нет однозначного правильного ответа: где должна быть сосредоточена бизнес-логика приложения?

Популярен подход, при котором вся логика находится на клиенте, вне базы данных. Выбирают его чаще тогда, когда разработчики не знают всех возможностей реляционной СУБД и работают с тем, с чем умеют, то есть с прикладным кодом.

В этом случае СУБД становится неким второстепенным элементом приложения и лишь обеспечивает «персистентность» данных, их надежное хранение. Часто от СУБД отгораживаются еще и дополнительным слоем абстракции, например, ORM-ом, который автоматически генерирует запросы к базе данных из конструкций на языке программирования, привычном разработчикам. Иногда такое решение мотивируют желанием обеспечить переносимость приложения на любую СУБД.

Подход имеет право на существование; если система, построенная таким образом, работает и выполняет возлагаемые на нее задачи — почему бы нет?

Но у этого решения есть и очевидные недостатки:

- **Согласованность данных поддерживает приложение.** Вместо того чтобы поручить СУБД следить за согласованностью данных (а это именно то, чем сильны реляционные системы), приложение выполняет необходимые проверки самостоятельно. Будьте уверены, что рано или поздно в базу попадут некорректные данные. Эти ошибки придется исправлять — или учить приложение работать с ними. А ведь бывают ситуации, когда над одной базой данных строятся несколько разных приложений. В этом случае обойтись без помощи СУБД просто невозможно.
- **Производительность оставляет желать лучшего.** ORM-системы позволяют в какой-то мере абстрагироваться от СУБД, но качество генерируемых ими SQL-запросов весьма сомнительно. Как правило, выполняется много небольших запросов, каждый из которых сам по себе работает достаточно быстро. Такая схема выдерживает только небольшие нагрузки на небольшом объеме данных и практически не поддается какой-либо оптимизации со стороны СУБД.
- **Усложняется прикладной код.** На прикладном языке программирования невозможно сформулировать по-настоящему сложный запрос, который бы автоматически и адекватно транслировался в SQL. Поэтому сложную обработку (если она нужна, разумеется) приходится программировать на уровне приложения, предварительно выбирая из базы все необходимые данные. Но, во-первых, при этом выполняется лишняя пересылка данных по сети, а во-вторых, такие алгоритмы, как сканирование, соединение, сортировка и агрегация в СУБД отлаживаются и оптимизируются

десятилетиями и справятся с задачей гарантированно лучше, чем прикладной код.

Конечно, использование СУБД на полную мощность, с реализацией всех ограничений целостности и логики работы с данными в хранимых функциях, требует вдумчивого изучения ее особенностей и возможностей. Потребуется освоить язык SQL для написания запросов и какой-либо язык серверного программирования (обычно PL/pgSQL) для написания функций и триггеров. Взамен вы овладеете надежным инструментом, одним из важных «кубиков» в архитектуре любой информационной системы.

Так или иначе, вопрос о том, где разместить бизнес-логику — на сервере или на клиенте, — вам придется для себя решить. Добавим только, что крайности нужны не всегда и часто истину стоит искать где-то посередине.

VII Настройка PostgreSQL

Основные настройки

PostgreSQL по умолчанию настроен так, чтобы сервер запускался даже на самом слабом «железе», однако для эффективной работы СУБД при настройке нужно учитывать как физические характеристики сервера, так и профиль нагрузки приложения.

Здесь мы рассмотрим только основные настройки, требующие особого внимания при промышленной эксплуатации СУБД. Тонкая настройка под конкретное приложение потребует дополнительных знаний, которые можно получить, например, пройдя курсы по администрированию PostgreSQL (см. с. 149).

Как изменять конфигурационные параметры

Чтобы изменить значение конфигурационного параметра, откройте файл `postgresql.conf` и либо найдите в нем нужный параметр и исправьте его значение, либо добавьте новую строку в конец файла — она будет иметь приоритет над значением, которое устанавливалось выше в том же файле.

Затем дайте серверу команду перечитать настройки:

```
postgres=# SELECT pg_reload_conf();
```

После этого проверьте текущее значение параметра командой SHOW. Если значение не изменилось — скорее всего при редактировании файла была допущена ошибка. Загляните в журнал сообщений сервера.

Вместо изменения файла в текстовом редакторе можно установить значение параметра с помощью SQL (после чего также необходимо перечитать настройки):

```
postgres=# ALTER SYSTEM SET work_mem='128MB';
```

Такие настройки попадают в файл `postgresql.auto.conf` и имеют приоритет над значениями, установленными в основном файле. Преимущество этого способа в том, что значения параметров сразу же проверяются на корректность.

Наиболее важные параметры

Одни из самых важных — это параметры, определяющие то, как PostgreSQL распоряжается оперативной памятью.

Параметр **shared_buffers** задает размер буферного кеша, нужного, чтобы работа с наиболее часто используемыми данными происходила в оперативной памяти и не требовала избыточных обращений к диску. Настройку можно начинать с 25 % от общего объема ОЗУ сервера. Изменение этого параметра вступает в силу только после перезагрузки сервера!

Значение параметра **effective_cache_size** не влияет на выделение памяти, но подсказывает PostgreSQL, на какой общий размер кеша рассчитывать, включая кеш операционной системы. Чем выше это значение, тем большее пред-

почтение отдается индексам. Начать можно с 50–75 % от объема ОЗУ.

Параметр **work_mem** определяет объем памяти, выделяемый для выполнения таких операций, как сортировка или построение хеш-таблиц при выполнении соединения. Признаком того, что памяти недостаточно, является активное использование временных файлов и, как следствие, уменьшение производительности. Значение по умолчанию 4 МБ в большинстве случаев стоит увеличить как минимум в несколько раз, но так, чтобы не выйти при этом за общий размер оперативной памяти сервера.

Параметр **maintenance_work_mem** определяет размер памяти, выделяемый служебным процессам. Его увеличение может ускорить построение индексов и работу процесса очистки (vacuum). Обычно устанавливается значение, в несколько раз превышающее значение work_mem.

Например, при ОЗУ 32 ГБ можно начать с настройки:

```
shared_buffers = '8GB'
effective_cache_size = '24GB'
work_mem = '128MB'
maintenance_work_mem = '512MB'
```

Отношение значений двух параметров random_page_cost и seq_page_cost должно соответствовать отношению скоростей произвольного и последовательного доступа к диску. По умолчанию предполагается, что произвольный доступ в 4 раза медленнее последовательного (расчет на обычные HDD-диски). Однако для дисковых массивов и SSD-дисков значение **random_page_cost** надо уменьшить (только никогда не изменяйте значение seq_page_cost, равное 1).

Например, для дисков SSD будет адекватна настройка:

```
random_page_cost = 1.2
```

Очень ответственной является настройка автоочистки (autovacuum). Этот процесс занимается «сборкой мусора» и выполняет ряд других важных для системы задач. Его настройка существенно зависит от конкретного приложения и нагрузки, которую оно создает.

В большинстве случаев можно начать со следующего:

- уменьшить значение **autovacuum_vacuum_scale_factor** до 0.01, чтобы очистка выполнялась чаще и меньшими порциями;
- увеличить значение **autovacuum_vacuum_cost_limit** (либо уменьшить **autovacuum_vacuum_cost_delay**) в 10 раз, чтобы очистка выполнялась быстрее (для версий до 12).

Не менее важна настройка процессов, связанных с обслуживанием буферного кеша и журнала предзаписи, но и она зависит от конкретного приложения. Начните с установки **checkpoint_completion_target** = 0.9 (чтобы сгладить нагрузку), увеличения **checkpoint_timeout** с 5 минут до 30 (чтобы уменьшить накладные расходы на выполнение контрольных точек) и пропорционального увеличения **max_wal_size** (с той же целью).

Тонкости настройки различных параметров подробно рассматриваются в учебном курсе DBA2 (с. 153).

Настройка подключения

Этот вопрос мы уже рассматривали в главе «PostgreSQL для приложения» на с. 89. Напомним, что обычно требуется установить параметр **listen_addresses** в значение '*'

и добавить разрешение на подключение в конфигурационный файл `pg_hba.conf`.

Вредные советы

Можно встретить советы по увеличению быстродействия, к которым ни в коем случае нельзя прислушиваться:

- Выключение автоочистки (autovacuum = off).

  Такая «экономия» ресурсов действительно даст кратковременный незначительный выигрыш в производительности, но приведет к накоплению «мусора» в данных и росту таблиц и индексов. Через некоторое время СУБД перестанет нормально функционировать. Автоочистку нужно не отключать, а правильно настраивать.

- Выключение синхронизации с диском (fsync = off).

  Отключение действительно приведет к существенному ускорению работы, но любой сбой сервера (будь то программный или аппаратный) приведет к полной потере баз данных. Восстановить систему можно будет только из резервной копии (если, конечно, она есть).

PostgreSQL и 1С

1С официально поддерживает работу с PostgreSQL. Это отличная возможность сэкономить на дорогих лицензиях на коммерческие СУБД.

Как и любое другое приложение, продукты 1С будут работать гораздо эффективнее, если PostgreSQL правильно

сконфигурирован. Кроме того, есть ряд параметров, специфических и обязательных для работы 1С.

Далее приведены рекомендации по установке и первоначальной настройке, которые помогут вам быстро приступить к работе.

Выбор версии и платформы

Для работы с 1С требуется версия PostgreSQL со специальными патчами. Такую версию можно взять на сайте 1C releases.1c.ru, а можно использовать СУБД Postgres Pro Standard или Postgres Pro Enterprise, которые тоже включают необходимые патчи.

PostgreSQL работает и на Windows, но если есть возможность выбора, то стоит отдать предпочтение ОС семейства Linux.

Перед установкой следует решить, необходим ли выделенный сервер для базы данных. Выделенный сервер более производителен за счет распределения нагрузки между сервером приложений и сервером базы данных.

Параметры конфигурации

Физические характеристики сервера должны соответствовать предполагаемой нагрузке. В качестве примерного ориентира можно привести следующие данные. Выделенный 8-ядерный сервер с ОЗУ 8 ГБ и дисковой подсистемой с RAID1 SSD должен справиться с объемом базы в пределах 100 ГБ, общим количеством пользователей до 50 человек, количеством документов до 2 000 в день. Если сервер не является

выделенным, для нужд PostgreSQL должно быть свободно соответствующее количество ресурсов общего сервера.

Исходя из общих рекомендаций, приведенных выше, и знания специфики приложений 1С, для такого сервера мы рекомендуем следующие начальные настройки:

```
# Обязательные для 1С настройки
standard_conforming_strings = off
escape_string_warning = off
shared_preload_libraries = 'online_analyze, plantuner'
plantuner.fix_empty_table = on
online_analyze.enable = on
online_analyze.table_type = 'temporary'
online_analyze.local_tracking = on
online_analyze.verbose = off

# Параметры, зависящие от объема оперативной памяти
shared_buffers = '2GB'          # 25% ОЗУ
effective_cache_size = '6GB'    # 75% ОЗУ
work_mem = '64MB'               # 64-128MB
maintenance_work_mem = '256MB' # 4*work_mem
# активная работа с временными таблицами
temp_buffers = '32MB'           # 32-128MB

# Требуется больше блокировок, чем 64 по умолчанию
max_locks_per_transaction = 256
```

Настройка подключения

Параметр listen_addresses в файле `postgresql.conf` должен быть установлен в значение `'*'`.

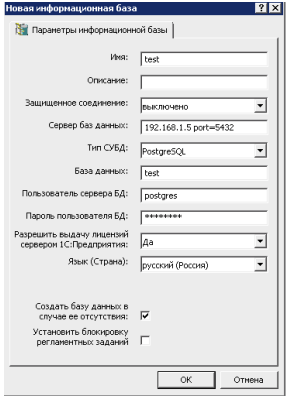
В начало конфигурационного файла `pg_hba.conf` необходимо добавить следующую строку, заменив «IP-адрес-сервера-1С» на конкретный адрес и маску подсети:

```
host    all    all    IP-адрес-сервера-1С   md5
```

После перезапуска PostgreSQL все изменения из файлов `postgresql.conf` и `pg_hba.conf` вступят в силу и сервер будет готов к подключению 1С.

Для подключения 1С использует суперпользовательскую роль, обычно это `postgres`. Установите для нее пароль:

```
postgres=# ALTER ROLE postgres PASSWORD 'p@ssw0rd';
ALTER ROLE
```



В настройках информационной базы 1С укажите в качестве сервера базы данных IP-адрес и порт сервера PostgreSQL и выберите тип СУБД «PostgreSQL». Укажите название базы данных, которая будет использоваться для 1С, и поставьте флажок «Создать базу данных в случае ее отсутствия» (создавать базу данных средствами PostgreSQL не нужно). Укажите имя и пароль суперпользовательской роли, которая будет использоваться для подключения.

Приведенных советов достаточно для быстрого начала работы, хотя, конечно, они не дают гарантии нужного уровня производительности.

Выражаем благодарность Антону Дорошкевичу из компании «Инфософт» за помощь в подготовке этого материала.

VIII pgAdmin

pgAdmin — популярное графическое средство для администрирования PostgreSQL. Программа упрощает основные задачи администрирования, отображает объекты баз данных, позволяет выполнять запросы SQL.

Долгое время стандартом де-факто являлся pgAdmin 3, однако разработчики из EnterpriseDB прекратили его поддержку и в 2016 году выпустили новую, четвертую версию, полностью переписав продукт с языка C++ на Python и веб-технологии. Из-за изменившегося интерфейса pgAdmin 4 поначалу был встречен достаточно прохладно, но продолжает разрабатываться и совершенствоваться.

Установка

Чтобы запустить pgAdmin 4 на Windows, воспользуйтесь установщиком на странице pgadmin.org/download. Процесс установки прост и очевиден, все предлагаемые значения можно оставить без изменений.

Для Debian и Ubuntu подключите репозиторий PostgreSQL (как описано на с. 30) и выполните команду

```
$ sudo apt-get install pgadmin4
```

В списке доступных программ появится «pgAdmin4».

Пользовательский интерфейс программы полностью переведен на русский язык нашей компанией. Чтобы сменить язык, нажмите значок **Настроить pgAdmin** (Configure pgAdmin) и в окне настроек выберите **Разное > Язык пользователя** (Miscellaneous > User language). Затем перезагрузите страницу в веб-браузере.

Подключение к серверу

В первую очередь настроим подключение к серверу. Нажмите на значок **Добавить новый сервер** (Add New Server) и в появившемся окне на вкладке **Общие** (General) введите произвольное **имя** (Name) для соединения.

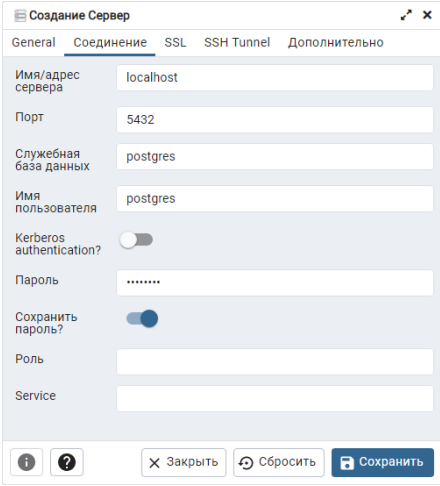
На вкладке **Соединение** (Connection) введите **имя сервера** (Host name/address), **порт** (Port), **имя пользователя** (Username) и **пароль** (Password).

Если не хотите вводить пароль каждый раз вручную, отметьте флажок **Сохранить пароль** (Save password). Пароли хранятся зашифрованными с помощью мастер-пароля, который pgAdmin попросит вас задать при первом запуске.

Обратите внимание, что у пользователя должен быть установлен пароль. Например, для `postgres` это можно сделать следующей командой:

```
postgres=# ALTER ROLE postgres PASSWORD 'p@ssw0rd';
```

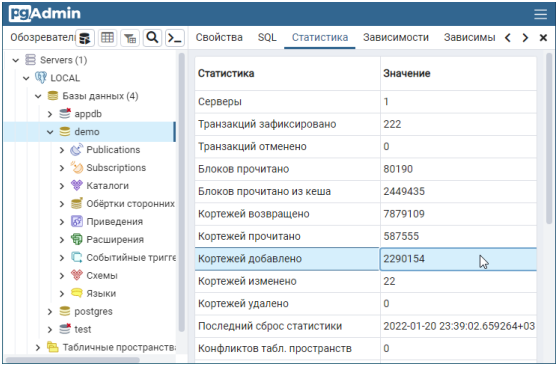
При нажатии на кнопку **Сохранить** (Save) программа проверит доступность сервера с указанными параметрами и запомнит новое подключение.



Навигатор

В левой части окна находится навигатор объектов. Разворачивая пункты списка, можно спуститься до сервера, который мы назвали LOCAL. Еще ниже будет перечень имеющихся в нем баз данных:

- `appdb` мы создали для проверки подключения к PostgreSQL из разных языков программирования;
- `demo` — демонстрационная база данных;
- `postgres` всегда создается при установке СУБД;
- `test` мы использовали, когда знакомились с SQL.



Развернув пункт **Схемы** (Schemas) для базы данных demo, можно обнаружить все таблицы, посмотреть их столбцы, ограничения целостности, индексы, триггеры и т. п.

Для каждого типа объекта в контекстном меню (по правой кнопке мыши) приведен список действий, которые с ним можно совершить. Например, выгрузить в файл или загрузить из файла, выдать привилегии, удалить.

В правой части окна на отдельных вкладках выводится справочная информация:

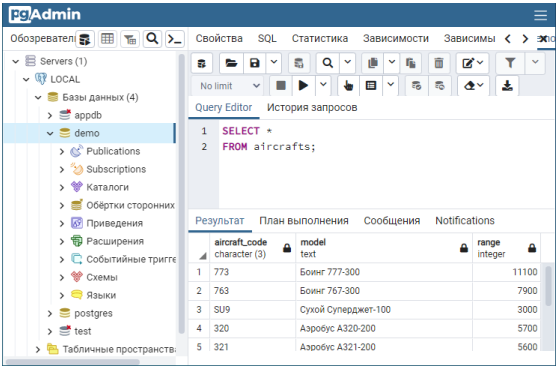
- **Панель информации** (Dashboard) — показывает графики, отражающие активность системы;
- **Свойства** (Properties) — свойства выбранного объекта (для столбца будет показан тип его данных и т. п.);
- **SQL** — команда SQL для создания выбранного в навигаторе объекта;

- **Статистика** (Statistics) — информация, которая используется оптимизатором для построения планов выполнения запросов и может рассматриваться администратором СУБД для анализа ситуации;
- **Зависимости**, **Зависимые** (Dependencies, Dependents) показывают зависимости между выбранным объектом и другими объектами в базе данных.

Выполнение запросов

Чтобы выполнить запрос, откройте новую вкладку с окном SQL, выбрав в меню **Инструменты — Запросник** (Tools — Query tool).

Введите запрос в верхней части окна и нажмите F5. В нижней части окна на вкладке **Результат** (Data Output) появится результат запроса.



Чтобы ввести новый запрос, необязательно стирать предыдущий: просто выделите нужный фрагмент кода и нажмите F5. Тогда история ваших действий всегда будет на виду — обычно это удобнее, чем искать нужный запрос в истории команд на вкладке **История запросов** (Query History).

Другое

Программа pgAdmin предоставляет графический интерфейс для стандартных утилит PostgreSQL, информации системного каталога, административных функций и команд SQL. Особо отметим встроенный отладчик PL/pgSQL-кода. Со всеми возможностями этой программы можно познакомиться на сайте продукта pgadmin.org либо в справочной системе самой программы.

IX Дополнительные возможности

Полнотекстовый поиск

Полнотекстовым поиском называют поиск документов на естественных языках, обычно с сортировкой по релевантности. В самом простом и типичном случае запросом становится набор слов, а критерием соответствия — их частотность в документе. Примерно таким поиском по нашим запросам занимаются те же Google и Яндекс. Впрочем, при всей мощи языка запросов SQL — для эффективной работы с такими данными не хватает и ее. Особенно заметно это стало в последнее время, когда хранилища информации заполняются лавинообразным потоком Больших Данных, обильных, зачастую плохо структурированных и плохо поддающихся структурированию.

Существует большое количество поисковиков, как платных, так и бесплатных, которые позволяют индексировать все имеющиеся документы и организовать вполне качественный поиск. В этих случаях индекс — важнейший инструмент и ускоритель поиска — не является частью базы данных. А это значит, что недоступными становятся такие ценимые пользователями СУБД особенности, как синхронизация содержимого БД, транзакционность, доступ к метаданным,

ограничение с их помощью области поиска, организация политики доступа к документам и многое другое.

Недостатки у все более популярных документо-ориентированных СУБД обычно такие же: развитые средства полнотекстового поиска есть, но безопасность и синхронизация не в приоритете. К тому же обычно они (MongoDB, например) принадлежат классу NoSQL СУБД, а значит, по определению лишены всей десятилетиями накопленной мощи SQL.

С другой стороны, традиционные SQL-СУБД имеют встроенные средства текстового поиска. Оператор `LIKE` входит в стандартный синтаксис SQL, но гибкость его явно недостаточна, так что производителям приходилось добавлять к стандарту SQL собственные расширения. У PostgreSQL это операторы сравнения `ILIKE`, `~`, `~*`, но и они не решают всех проблем, так как не учитывают словоизменение, не приспособлены для ранжирования и не слишком быстро работают.

Если говорить об инструментах собственно полнотекстового поиска, то важно понимать, что до их стандартизации пока далеко — в каждой реализации СУБД свой синтаксис и свои подходы. В этом плане российскому пользователю PostgreSQL очень удобно: полнотекстовый поиск реализован российскими разработчиками, а при необходимости можно разобраться в нем подробнее, напрямую связавшись со специалистами или даже посетив их лекции. Мы же ограничимся простыми примерами.

Для изучения возможностей полнотекстового поиска создадим еще одну таблицу в демонстрационной базе данных. Пусть это будут наброски конспекта, разбитые на главы-лекции:

```
test=# CREATE TABLE course_chapters(
  c_no text REFERENCES courses(c_no),
  ch_no text,
  ch_title text,
  txt text,
  CONSTRAINT pkt_ch PRIMARY KEY(ch_no, c_no)
);
CREATE TABLE
```

Введем в таблицу тексты первых лекций по знакомым нам специальностям CS301 и CS305:

```
test=# INSERT INTO course_chapters(
  c_no, ch_no,ch_title, txt)
VALUES
('CS301', 'I', 'Базы данных',
 'С этой главы начинается наше знакомство ' ||
 'с увлекательным миром баз данных'),
('CS301', 'II', 'Первые шаги',
 'Продолжаем знакомство с миром баз данных. ' ||
 'Создадим нашу первую текстовую базу данных'),
('CS305', 'I', 'Локальные сети',
 'Здесь начнется наше полное приключений ' ||
 'путешествие в интригующий мир сетей');
INSERT 0 3
```

Проверим результат:

```
test=# SELECT ch_no AS no, ch_title, txt
FROM course_chapters \gx
-[ RECORD 1 ]-----------------------------------------
no       | I
ch_title | Базы данных
txt      | С этой главы начинается наше знакомство с
           увлекательным миром баз данных
-[ RECORD 2 ]-----------------------------------------
no       | II
ch_title | Первые шаги
txt      | Продолжаем знакомство с миром баз данных.
           Создадим нашу первую текстовую базу данных
```

```
-[ RECORD 3 ]-------------------------------------
no       | I
ch_title | Локальные сети
txt      | Здесь начнется наше полное приключений
           путешествие в интригующий мир сетей
```

Найдем в таблице информацию по базам данных традиционными средствами SQL, используя оператор LIKE:

```
test=# SELECT txt
FROM course_chapters
WHERE txt LIKE '%базы данных%' \gx
```

Легко догадаться, каким будет результат: 0 строк. Ведь оператор LIKE не опознает слова «базы» в формах родительного и винительного падежей («баз», «базу»).

А по запросу

```
test=# SELECT txt
FROM course_chapters
WHERE txt LIKE '%базу данных%' \gx
```

— будет выдана строка из главы II, но не из главы I, где это слово стоит в другом падеже:

```
-[ RECORD 1 ]-------------------------------------
txt | Продолжаем знакомство с миром баз данных.
      Создадим нашу первую текстовую базу данных
```

В Postgres есть оператор ILIKE, позволяющий хотя бы не думать о различии регистров (т. е. заглавных и строчных букв). Конечно, есть и регулярные выражения (шаблоны поиска), придумывание которых сродни искусству, но иногда все же хочется иметь инструмент, думающий за тебя. Поэтому добавим к таблице глав еще один столбец со специальным типом данных — tsvector:

```
test=# ALTER TABLE course_chapters
  ADD txtvector tsvector;
test=# UPDATE course_chapters
  SET txtvector = to_tsvector('russian',txt);
test=# SELECT txtvector
FROM course_chapters \gx
-[ RECORD 1 ]----------------------------------------
txtvector | 'баз':10 'глав':3 'дан':11 'знакомств':6
            'мир':9 'начина':4 'наш':5 'увлекательн':8
-[ RECORD 2 ]----------------------------------------
txtvector | 'баз':5,11 'дан':6,12 'знакомств':2
            'мир':4 'наш':8 'перв':9 'продолжа':1
            'создад':7 'текстов':10
-[ RECORD 3 ]----------------------------------------
txtvector | 'интриг':8 'мир':9 'начнет':2 'наш':3
            'полн':4 'приключен':5 'путешеств':6
            'сет':10
```

Мы видим, что в строках:

1) слова сократились до своих неизменяемых частей (лексем),
2) появились цифры, означающие позицию вхождения слова в текст (видно, что некоторые слова вошли два раза),
3) в строку не вошли предлоги (а также не вошли бы союзы и прочие незначимые для поиска единицы предложения — так называемые стоп-слова).

Поиск будет работать еще лучше, если включить в его область названия глав, а заодно придать им бóльшую весомость, чем у остального текста (это делается функцией `setweight`). Поправим таблицу:

```
test=# UPDATE course_chapters
  SET txtvector =
      setweight(to_tsvector('russian',ch_title),'B')
      || ' ' ||
      setweight(to_tsvector('russian',txt),'D');
UPDATE 3
```

```
test=# SELECT txtvector
FROM course_chapters \gx

-[ RECORD 1 ]-----------------------------------------
txtvector | 'баз':1B,12 'глав':5 'дан':2B,13
            'знакомств':8 'мир':11 'начина':6 'наш':7
            'увлекательн':10
-[ RECORD 2 ]-----------------------------------------
txtvector | 'баз':7,13 'дан':8,14 'знакомств':4
            'мир':6 'наш':10 'перв':1B,11 'продолжа':3
            'создад':9 'текстов':12 'шаг':2B
-[ RECORD 3 ]-----------------------------------------
txtvector | 'интриг':10 'локальн':1B 'мир':11
            'начнет':4 'наш':5 'полн':6 'приключен':7
            'путешеств':8 'сет':2B,12
```

У лексем появился относительный вес — B и D (из четырех возможных — A, B, C, D). Реальный вес мы будем задавать при составлении запросов. Это придаст им дополнительную гибкость.

Во всеоружии вернемся к поиску. Функции `to_tsvector` симметрична функция `to_tsquery`, приводящая символьное выражение к типу данных `tsquery`, который используют в запросах.

```
test=# SELECT ch_title
FROM course_chapters
WHERE txtvector @@
      to_tsquery('russian','базы & данные');

  ch_title
-------------
 Базы данных
 Первые шаги
(2 rows)
```

Можно убедиться, что поисковый запрос с другими грамматическими формами тех же слов (`'база & данных'`) даст

тот же результат. Здесь мы применили оператор сравнения @@, выполняющий для полнотекстового поиска ту же роль, что оператор LIKE — для обычного. Оператор @@ не допускает выражений естественного языка с пробелами, поэтому слова в запросе соединены логическим оператором «И».

Аргумент russian указывает на конфигурацию, используемую СУБД и определяющую то, какие словари надо подключать и каким парсером разбивать фразу на лексемы.

Словари, несмотря на свое название, позволяют преобразовывать лексемы самым различным образом. Например, используемый по умолчанию простой словарь-стеммер snowball оставляет от слова только неизменяемую часть, благодаря чему поиск игнорирует окончания слов в запросе. Можно подключать и другие, например:

- «обычные» словари, такие как ispell, myspell или hunspell, для более точного учета морфологии;
- словари синонимов;
- тезаурус;
- unaccent, чтобы превратить букву «ё» в «е».

Благодаря присвоенным весам записи выводятся в порядке убывания рейтинга:

```
test=# SELECT ch_title,
    ts_rank_cd('{0.1, 0.0, 1.0, 0.0}', txtvector, q)
FROM course_chapters,
    to_tsquery('russian','базы & данных') q
WHERE txtvector @@ q
ORDER BY ts_rank_cd DESC;
  ch_title   | ts_rank_cd
-------------+------------
 Базы данных |    1.11818
 Первые шаги |       0.22
(2 rows)
```

Массив {0.1, 0.0, 1.0, 0.0} задает веса. Это не обязательный аргумент функции `ts_rank_cd`, по умолчанию массив {0.1, 0.2, 0.4, 1.0} соответствует D, C, B, A. Вес слова влияет на значимость найденной строки.

В заключительном эксперименте модифицируем выдачу. Будем считать, что найденные слова мы хотим выделить жирным шрифтом в странице html. Функция `ts_headline` задает наборы символов, обрамляющих слово, а также минимальное и максимальное количество слов в строке:

```
test=# SELECT ts_headline(
  'russian',
  txt,
  to_tsquery('russian', 'мир'),
  'StartSel=<b>, StopSel=</b>, MaxWords=50, MinWords=5'
)
FROM course_chapters
WHERE to_tsvector('russian', txt) @@
      to_tsquery('russian', 'мир');

-[ RECORD 1 ]----------------------------------------
ts_headline | знакомство с увлекательным <b>миром</b>
              баз данных
-[ RECORD 2 ]----------------------------------------
ts_headline | <b>миром</b> баз данных. Создадим нашу
-[ RECORD 3 ]----------------------------------------
ts_headline | путешествие в интригующий <b>мир</b>
              сетей
```

Для ускорения полнотекстового поиска используются специальные индексы GiST, GIN и RUM, отличные от обычных индексов в базах данных. Но они, как и многие другие полезные знания о полнотекстовом поиске, останутся вне рамок этого краткого руководства.

Более подробно о полнотекстовом поиске можно прочитать в документации PostgreSQL по адресу postgrespro.ru/doc/textsearch.

Работа с JSON и JSONB

Реляционные базы данных, использующие SQL, создавались с большим запасом прочности: первой заботой их потребителей была целостность и безопасность данных, а объемы информации были несравнимы с современными. Когда появилось новое поколение СУБД — NoSQL, — сообщество призадумалось: отказ от поддержки строгой согласованности и значительно упрощенная модель данных (поначалу это были преимущественно хранилища пар ключ—значение) позволяли значительно ускорить поиск. Базы NoSQL могли обрабатывать небывалые объемы информации и легко масштабировались, вовсю используя параллельные вычисления.

Когда прошел первый шок, стало понятно, что для большинства реальных задач простой структурой не обойтись. Стали появляться сложные ключи, потом группы ключей. Создатели реляционных СУБД не желали отставать от жизни и начали добавлять возможности, типичные для NoSQL.

Поскольку в реляционных СУБД изменение схемы данных связано с большими издержками, как нельзя кстати оказался новый тип данных — JSON. Напоминающий XML своей иерархической структурой, он предназначался для программирования на JavaScript (откуда название), в том числе для разработки AJAX-приложений. Гибкость JSON позволила разработчикам приложений добавлять разнородные данные с непредсказуемой структурой, не переделывая каждый раз схему базы данных.

Допустим, в нашу демобазу студентов теперь можно вносить личные данные: запустили анкету, расспросили преподавателей. Не все пункты анкеты обязательны к заполнению, а некоторые допускают такие ответы, как «другое»

и «добавьте данные на свое усмотрение». При традиционном подходе новая информация, не укладывающаяся в текущую структуру, потребовала бы добавления множества таблиц или столбцов с большим количеством пустых полей, а появление все новой информации приводило бы к постоянному перекраиванию всей базы данных.

Эта проблема решается использованием типа `json` и появившегося позже `jsonb`, хранящего данные в экономичном бинарном виде и, в отличие от `json`, приспособленного для построения индексов, что ускоряет поиск иногда на порядки.

Создадим таблицу с объектами JSON:

```
test=# CREATE TABLE student_details(
  de_id int,
  s_id int REFERENCES students(s_id),
  details json,
  CONSTRAINT pk_d PRIMARY KEY(s_id, de_id)
);
```

```
test=# INSERT INTO student_details
  (de_id, s_id, details)
VALUES
(1, 1451,
'{ "достоинства": "отсутствуют",
   "недостатки":
   "неумеренное употребление мороженого"
}'
),
(2, 1432,
'{ "хобби":
     { "гитарист":
         { "группа": "Постгрессоры",
           "гитары":["страт","телек"]
         }
     }
}'
),
```

```
(3, 1556,
'{ "хобби": "косплей",
   "достоинства":
     { "мать-героиня":
         { "Вася": "м",
           "Семен": "м",
           "Люся": "ж",
           "Макар": "м",
           "Саша":"сведения отсутствуют"
         }
     }
}'
),
(4, 1451,
'{ "статус": "отчислена"
}'
);
```

Проверим, все ли данные на месте. Для удобства соединим таблицы `student_details` и `students` при помощи конструкции `WHERE`, ведь в новой таблице имена студентов отсутствуют:

```
test=# SELECT s.name, sd.details
FROM student_details sd, students s
WHERE s.s_id = sd.s_id \gx

-[ RECORD 1 ]-------------------------------------
name    | Анна
details | { "достоинства": "отсутствуют",          +
        |    "недостатки":                         +
        |    "неумеренное употребление мороженого" +
        | }
-[ RECORD 2 ]-------------------------------------
name    | Виктор
details | { "хобби":                               +
        |      { "гитарист":                       +
        |          { "группа": "Постгрессоры",     +
        |            "гитары":["страт","телек"]    +
        |          }                               +
        |      }                                   +
        | }
```

```
-[ RECORD 3 ]-------------------------------------
name    | Нина
details | { "хобби": "косплей",                    +
        |    "достоинства":                        +
        |      { "мать-героиня":                   +
        |          { "Вася": "м",                  +
        |            "Семен": "м",                 +
        |            "Люся": "ж",                  +
        |            "Макар": "м",                 +
        |            "Саша":"сведения отсутствуют" +
        |          }                               +
        |      }                                   +
        | }
-[ RECORD 4 ]-------------------------------------
name    | Анна
details | { "статус": "отчислена"                  +
        | }
```

Допустим, нас интересуют записи, содержащие информацию о достоинствах студентов. Обратимся к значению ключа «достоинство» с помощью оператора ->>:

```
test=# SELECT s.name, sd.details
FROM  student_details sd, students s
WHERE s.s_id = sd.s_id
AND   sd.details ->> 'достоинства' IS NOT NULL \gx
-[ RECORD 1 ]-------------------------------------
name    | Анна
details | { "достоинства": "отсутствуют",          +
        |    "недостатки":                         +
        |    "неумеренное употребление мороженого" +
        | }
-[ RECORD 2 ]-------------------------------------
name    | Нина
details | { "хобби": "косплей",                    +
        |    "достоинства":                        +
        |      { "мать-героиня":                   +
        |          { "Вася": "м",                  +
        |            "Семен": "м",                 +
        |            "Люся": "ж",                  +
        |            "Макар": "м",                 +
        |            "Саша":"сведения отсутствуют" +
```

```
        |          }                                  +
        |      }                                      +
        | }
```

Мы убедились, что к достоинствам Анны и Нины имеют отношение две записи, однако такой ответ нас вряд ли удовлетворит: на самом деле достоинства Анны «отсутствуют». Скорректируем запрос:

```
test=# SELECT s.name, sd.details
FROM  student_details sd, students s
WHERE s.s_id = sd.s_id
AND   sd.details ->> 'достоинства' IS NOT NULL
AND   sd.details ->> 'достоинства' != 'отсутствуют';
```

Убедитесь, что этот запрос оставит в списке только Нину, обладающую реальными, а не отсутствующими достоинствами.

Но такой способ срабатывает не всегда. Попробуем найти, на каких гитарах играет музыкант Витя:

```
test=# SELECT sd.de_id, s.name, sd.details
FROM  student_details sd, students s
WHERE s.s_id = sd.s_id
AND   sd.details ->> 'гитары' IS NOT NULL \gx
```

Запрос ничего не выдаст. Дело в том, что соответствующая пара ключ-значение находится внутри иерархии JSON, то есть вложена в пары более высокого уровня:

```
name    | Виктор
details | { "хобби":                                 +
        |       { "гитарист":                        +
        |           { "группа": "Постгрессоры",      +
        |             "гитары":["страт","телек"]     +
        |           }                                +
        |       }                                    +
        | }
```

Чтобы добраться до гитар, воспользуемся оператором #> и спустимся с «хобби» вниз по иерархии:

```
test=# SELECT sd.de_id, s.name,
       sd.details #> '{хобби,гитарист,гитары}'
FROM  student_details sd, students s
WHERE s.s_id = sd.s_id
AND   sd.details #> '{хобби,гитарист,гитары}'
      IS NOT NULL \gx
```

— и убедимся, что Виктор фанат фирмы Fender:

```
 de_id |  name  |     ?column?
-------+--------+------------------
     2 | Виктор | ["страт","телек"]
```

У типа данных json есть младший брат jsonb. Буква «b» подразумевает бинарный (а не текстовый) способ хранения данных и их структуры, что во многих случаях ускоряет поиск. В последнее время jsonb используется намного чаще, чем json.

```
test=# ALTER TABLE student_details
ADD details_b jsonb;

test=# UPDATE student_details
SET details_b = to_jsonb(details);

test=# SELECT de_id, details_b
FROM student_details \gx

-[ RECORD 1 ]-------------------------------------
de_id     | 1
details_b | {"недостатки": "неумеренное
            употребление мороженого",
            "достоинства": "отсутствуют"}
-[ RECORD 2 ]-------------------------------------
de_id     | 2
details_b | {"хобби": {"гитарист": {"гитары":
            ["страт", "телек"], "группа":
            "Постгрессоры"}}}
```

```
-[ RECORD 3 ]-------------------------------------
de_id     | 3
details_b | {"хобби": "косплей", "достоинства":
            {"мать-героиня": {"Вася": "м", "Люся":
            "ж", "Саша": "сведения отсутствуют",
            "Макар": "м", "Семен": "м"}}}
-[ RECORD 4 ]-------------------------------------
de_id     | 4
details_b | {"статус": "отчислена"}
```

Можно заметить, что, кроме иной формы записи, изменился порядок значений в парах: Саша, сведения о которой, как мы помним, отсутствуют, заняла теперь место в списке перед Макаром. Это не недостаток jsonb относительно json, а особенность хранения информации.

Операторов для работы с jsonb больше, чем для работы с json. Один из полезнейших — оператор вхождения в объект @>, похожий на #> для json.

Например, найдем запись, где упоминается дочь матери-героини Люся:

```
test=# SELECT s.name,
       jsonb_pretty(sd.details_b) json
FROM   student_details sd, students s
WHERE s.s_id = sd.s_id
AND    sd.details_b @>
       '{"достоинства":{"мать-героиня":{}}}' \gx
```

```
-[ RECORD 1 ]-------------------------------------
name | Нина
json | {                                          +
     |     "хобби": "косплей",                    +
     |     "достоинства": {                       +
     |         "мать-героиня": {                  +
     |             "Вася": "м",                   +
     |             "Люся": "ж",                   +
     |             "Саша": "сведения отсутствуют",+
     |             "Макар": "м",                  +
     |             "Семен": "м"                   +
```

```
        |         }                                    +
        |     }                                        +
        | }
```

Мы использовали функцию jsonb_pretty(), которая форматирует вывод типа jsonb.

Или можно воспользоваться функцией jsonb_each(), разворачивающей пары ключ-значение:

```
test=# SELECT s.name,
       jsonb_each(sd.details_b)
FROM   student_details sd, students s
WHERE s.s_id = sd.s_id
AND    sd.details_b @>
       '{"достоинства":{"мать-героиня":{}}}'
\gx

-[ RECORD 1 ]-------------------------------------
name       | Нина
jsonb_each | (хобби,"""косплей""")
-[ RECORD 2 ]-------------------------------------
name       | Нина
jsonb_each | (достоинства,"{""мать-героиня"":
             {""Вася"": ""м"", ""Люся"": ""ж"",
             ""Саша"": ""сведения отсутствуют"",
             ""Макар"": ""м"", ""Семен"":
             ""м""}}")
```

Между прочим, вместо имени ребенка Нины в запросе было оставлено пустое место {}. Такой синтаксис добавляет гибкости процессу разработки реальных приложений.

Но главное в jsonb — это, пожалуй, индексы, поддерживающие оператор @>, обратный ему <@ и многие другие (эффективнее всех обычно работает индекс GIN). Тип json не поддерживает индексы, так что для приложений с серьезной нагрузкой лучше выбирать jsonb.

Подробнее о типах json и jsonb и о функциях для работы с ними можно узнать на страницах документации PostgreSQL postgrespro.ru/doc/datatype-json и postgrespro.ru/doc/functions-json.

Впрочем, пользователям было недостаточно и функциональности jsonb, и в 2014-м году для версии 9.4 Ф. Сигаевым, А. Коротковым и О. Бартуновым было разработано расширение jsquery, определяющее язык запросов для извлечения данных из jsonb и индексы для ускорения этих запросов. Для этого создали и новый тип данных — jsquery.

С помощью языка запросов можно, например, искать записи, указывая путь. Нотация с точками отображает иерархию внутри jsonb:

```
test=# SELECT *
FROM student_details
WHERE details::jsonb @@
      'хобби.гитарист.группа=Постгрессоры'::jsquery;
```

Если путь неизвестен, ветви можно подменить звездочкой:

```
test=# SELECT s_id, details
FROM student_details
WHERE details::jsonb @@
      'хобби.*.группа=Постгрессоры'::jsquery;
```

Но при этом без знания иерархии работать с нужным значением очень сложно.

Когда вышел стандарт SQL:2016, в который входит и язык путей SQL/JSON Path, в Postgres Professional была разработана его реализация, добавляющая тип jsonpath и набор

функций для работы с JSON с помощью этого языка. Эти возможности вошли в PostgreSQL 12.

Нотация в SQL/JSON Path отличается от обычных операторов PostgreSQL для JSON. К нотации расширения `jsquery` она ближе: иерархию тоже размечают точками. Но грамматика SQL/JSON Path более развитая.

- `$.a.b.c` — в версии PostgreSQL 11 пришлось бы написать `'a'->'b'->'c'`.
- `$` — представляет текущий контекст, то есть фрагмент документа JSON, который подлежит обработке.
- `@` — текущий элемент в выражении-фильтре. Перебираются пути, доступные в выражении с `$`.
- `*` — метасимвол (wildcard). В выражениях с `$` или `@` означает любое значение участка пути, но при этом с учетом иерархии.
- `**` — как часть выражения с `$` или `@` может означать любое значение участка пути без учета иерархии. Полезно, когда неизвестен уровень вложенности элементов.
- `?` позволяет организовать фильтр, аналогичный `WHERE`, например `$.a.b.c ? (@.x > 10)`.

Запрос с функцией `jsonb_path_query()` для поиска увлекающихся косплеем может выглядеть так:

```
test=# SELECT s_id, jsonb_path_query(
    details::jsonb,
    '$.хобби ? (@  == "косплей")'
)
FROM student_details;
 s_id | jsonb_path_query
------+------------------
 1556 | "косплей"
(1 row)
```

Запрос заглядывает только в ту ветвь JSON, которая начинается с ключа «хобби», и проверяет, равно ли соответствующее значение «косплею». Но если заменить «косплей» на «гитарист», не будет выведено ничего, так как в нашей таблице «гитарист» — не значение, а ключ вложенной записи.

В запросе используются две иерархии: одна действует внутри выражения $, ограничивающего поиск, а вторая — внутри @, то есть выражения, подставляемого при переборе. Это позволяет добиваться одной цели разными способами.

Например, такой запрос

```
test=# SELECT s_id, jsonb_path_query(
  details::jsonb,
  '$.хобби.гитарист.группа?(@=="Постгрессоры")'
)
FROM student_details;
```

— и такой

```
test=# SELECT s_id, jsonb_path_query(
  details::jsonb,
  '$.хобби.гитарист?(@.группа=="Постгрессоры").группа'
)
FROM student_details;
```

— дадут одинаковый результат:

```
 s_id | jsonb_path_query
------+------------------
 1432 | "Постгрессоры"
(1 row)
```

В первый раз мы задавали для каждой записи область поиска внутри ветви «хобби.гитарист.группа», которой, если

взглянуть на сам JSON, соответствует единственное значение — «Постгрессоры», так что и перебирать было нечего. Во втором варианте перебирать надо было все ветви, идущие от «хобби.гитарист», но в выражении фильтра мы прописали путь-ветвь «группа» — иначе запись не была бы найдена. В такой синтаксической конструкции нам надо заранее знать иерархию внутри JSON. Но что делать, если мы ее не знаем?

В этом случае подойдет двойной метасимвол **. Чрезвычайно полезная возможность! Допустим, мы забыли, что такое «страт» — то ли высоко летающий воздушный шар, то ли гитара, то ли представитель высшей социальной страты, но нам надо выяснить, есть ли вообще это слово в нашей таблице. В предыдущих реализациях операций с JSON пришлось бы делать сложный перебор (если работать с типом `jsonb`, не преобразуя его в текст). Теперь же можно сказать так:

```
test=# SELECT s_id, jsonb_path_exists(
  details::jsonb,
  '$.** ? (@ == "страт")'
)
FROM student_details;
 s_id | jsonb_path_exists
------+-------------------
 1451 | f
 1432 | t
 1556 | f
 1451 | f
(4 rows)
```

С возможностями SQL/JSON Path можно ознакомиться не только в документации (postgrespro.ru/doc/datatype-json#DATATYPE-JSONPATH), но и в статье «Что заморозили на feature freeze 2019. Часть I. JSONPath» (habr.com/ru/company/postgrespro/blog/448612/).

Интеграция с внешними системами

Приложения живут не поодиночке, а среди себе подобных, и зачастую общаются друг с другом. Такое общение можно реализовать средствами самих приложений, например, при помощи веб-сервисов или обмена файлами, а можно воспользоваться инструментами СУБД.

В PostgreSQL реализована поддержка стандарта ISO/IEC 9075-9 (SQL/MED, Management of External Data) по работе в SQL с внешними источниками информации через специальный механизм оберток сторонних данных (foreign data wrapper).

Идея механизма в том, чтобы к внешним (сторонним) данным можно было обращаться как к обычным таблицам. Для этого предварительно создаются сторонние таблицы (foreign table), которые сами не содержат данных, а перенаправляют все обращения к внешнему источнику. Такой подход упрощает разработку приложений, так как не требует знания специфики работы с конкретным внешним источником.

Процесс создания сторонних таблиц состоит из нескольких последовательных действий.

1. Командой `CREATE FOREIGN DATA WRAPPER` подключаем библиотеку для работы с конкретным источником данных.

2. Командой `CREATE SERVER` определяем сервер, где находится источник внешних данных. Для этого в команде обычно указывают такие параметры, как имя сервера, номер порта, имя базы данных.

3. К одному и тому же внешнему источнику данных могут подключаться разные пользователи PostgreSQL, используя для этого разные учетные записи, поэтому командой `CREATE USER MAPPING` указываем сопоставление имен.

4. Для необходимых таблиц и представлений удаленного сервера создаем сторонние таблицы командой `CREATE FOREIGN TABLE`. А команда `IMPORT FOREIGN SCHEMA` позволяет импортировать описания всех или части таблиц из указанной схемы.

Мы рассмотрим интеграцию PostgreSQL с наиболее популярными СУБД: Oracle, MySQL, SQL Server и PostgreSQL. Но сначала нужно установить соответствующие библиотеки для работы с базами данных.

Установка расширений

В дистрибутив PostgreSQL входят две обертки сторонних данных: `postgres_fdw` и `file_fdw`. Первая предназначена для работы с удаленными базами PostgreSQL, вторая — с файлами на сервере. Помимо этого сообществом разработаны и поддерживаются библиотеки для доступа ко многим распространенным системам баз данных. Их список можно посмотреть на сайте pgxn.org/tag/fdw.

Обертки сторонних данных для Oracle, MySQL и SQL Server доступны в виде расширений:

1. Oracle — github.com/laurenz/oracle_fdw;
2. MySQL — github.com/EnterpriseDB/mysql_fdw;
3. SQL Server — github.com/tds-fdw/tds_fdw.

Следуйте инструкциям с этих сайтов, и сборка и установка не вызовет затруднений. Если все сделать правильно, в списке доступных расширений появятся соответствующие обертки сторонних данных. Например, для oracle_fdw:

```
test=# SELECT name, default_version
FROM pg_available_extensions
WHERE name = 'oracle_fdw' \gx
-[ RECORD 1 ]---+-----------
name            | oracle_fdw
default_version | 1.2
```

Oracle

Вначале устанавливаем расширение, которое в свою очередь создаст обертку сторонних данных:

```
test=# CREATE EXTENSION oracle_fdw;
CREATE EXTENSION
```

Проверим, что соответствующая обертка создана:

```
test=# \dew
List of foreign-data wrappers
-[ RECORD 1 ]-------------------
Name      | oracle_fdw
Owner     | postgres
Handler   | oracle_fdw_handler
Validator | oracle_fdw_validator
```

Следующий шаг — создание сервера сторонних данных. В предложении OPTIONS указывается параметр dbserver, определяющий специфическую для подключения к экземпляру Oracle информацию: имя сервера, номер порта и название экземпляра.

```
test=# CREATE SERVER oracle_srv
  FOREIGN DATA WRAPPER oracle_fdw
  OPTIONS (dbserver '//localhost:1521/orcl');
CREATE SERVER
```

Пользователь PostgreSQL `postgres` будет подключаться к экземпляру Oracle как `scott`.

```
test=# CREATE USER MAPPING FOR postgres
  SERVER oracle_srv
  OPTIONS (user 'scott', password 'tiger');
CREATE USER MAPPING
```

Сторонние таблицы будем импортировать в отдельную схему. Создадим ее:

```
test=# CREATE SCHEMA oracle_hr;
CREATE SCHEMA
```

Импортируем описания удаленных таблиц. Ограничимся двумя популярными таблицами `dept` и `emp`:

```
test=# IMPORT FOREIGN SCHEMA "SCOTT"
  LIMIT TO (dept, emp)
  FROM SERVER oracle_srv
  INTO oracle_hr;
IMPORT FOREIGN SCHEMA
```

Заметим, что названия объектов в словаре данных Oracle хранятся в верхнем регистре, а в системном каталоге PostgreSQL — в нижнем. Поэтому, работая с внешними данными в PostgreSQL, пишите имя схемы Oracle заглавными буквами и в двойных кавычках, чтобы избежать преобразования в нижний регистр.

Смотрим список сторонних таблиц:

```
test=# \det oracle_hr.*
     List of foreign tables
  Schema   | Table |   Server
-----------+-------+------------
 oracle_hr | dept  | oracle_srv
 oracle_hr | emp   | oracle_srv
(2 rows)
```

Теперь для обращения к удаленным данным выполняем запросы к сторонним таблицам:

```
test=# SELECT * FROM oracle_hr.emp LIMIT 1 \gx
-[ RECORD 1 ]-------------------
empno    | 7369
ename    | SMITH
job      | CLERK
mgr      | 7902
hiredate | 1980-12-17
sal      | 800.00
comm     |
deptno   | 20
```

Можно не только читать данные, но и делать изменения:

```
test=# INSERT INTO oracle_hr.dept(deptno, dname, loc)
  VALUES (50, 'EDUCATION', 'MOSCOW');
INSERT 0 1
test=# SELECT * FROM oracle_hr.dept;
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | EDUCATION  | MOSCOW
(5 rows)
```

MySQL

Создаем расширение и вместе с ним обертку сторонних данных:

```
test=# CREATE EXTENSION mysql_fdw;
CREATE EXTENSION
```

Сторонний сервер, описывающий экземпляр, определяется параметрами `host` и `port`:

```
test=# CREATE SERVER mysql_srv
    FOREIGN DATA WRAPPER mysql_fdw
    OPTIONS (host 'localhost', port '3306');
CREATE SERVER
```

Подключаться будем под суперпользователем MySQL:

```
test=# CREATE USER MAPPING FOR postgres
  SERVER mysql_srv
  OPTIONS (username 'root', password 'p@ssw0rd');
CREATE USER MAPPING
```

Обертка поддерживает команду `IMPORT FOREIGN SCHEMA`, но можно создать внешнюю таблицу и вручную:

```
test=# CREATE FOREIGN TABLE employees (
  emp_no      int,
  birth_date  date,
  first_name  varchar(14),
  last_name   varchar(16),
  gender      varchar(1),
  hire_date   date)
SERVER mysql_srv
  OPTIONS (dbname 'employees',
           table_name 'employees');
CREATE FOREIGN TABLE
```

Проверяем:

```
test=# SELECT * FROM employees LIMIT 1 \gx
-[ RECORD 1 ]----------
emp_no     | 10001
birth_date | 1953-09-02
first_name | Georgi
last_name  | Facello
gender     | M
hire_date  | 1986-06-26
```

Как и для Oracle, обертка `mysql_fdw` разрешает не только чтение, но и изменение данных.

SQL Server

Создаем расширение и вместе с ним обертку сторонних данных:

```
test=# CREATE EXTENSION tds_fdw;
CREATE EXTENSION
```

Создаем сторонний сервер:

```
test=# CREATE SERVER sqlserver_srv
  FOREIGN DATA WRAPPER tds_fdw
  OPTIONS (servername 'localhost', port '1433',
           database 'AdventureWorks');
CREATE SERVER
```

Предоставляемая информация не меняется: нужно указать имя сервера, номер порта, базу данных. Но количество и названия параметров в предложении `OPTIONS` отличаются от того, что мы видели для `oracle_fdw` и `mysql_fdw`.

Будем подключаться под учетной записью суперпользователя SQL Server:

```
test=# CREATE USER MAPPING FOR postgres
    SERVER sqlserver_srv
    OPTIONS (username 'sa', password 'p@ssw0rd');
CREATE USER MAPPING
```

Создадим отдельную схему для сторонних таблиц:

```
test=# CREATE SCHEMA sqlserver_hr;
CREATE SCHEMA
```

Импортируем целиком схему `HumanResources` в созданную схему PostgreSQL:

```
test=# IMPORT FOREIGN SCHEMA HumanResources
    FROM SERVER sqlserver_srv
    INTO sqlserver_hr;
IMPORT FOREIGN SCHEMA
```

Список импортированных таблиц можно проверить командой `\det`, а можно найти в системном каталоге следующим запросом:

```
test=# SELECT ft.ftrelid::regclass AS "Table"
FROM pg_foreign_table ft;
                  Table
-----------------------------------------
 sqlserver_hr.Department
 sqlserver_hr.Employee
 sqlserver_hr.EmployeeDepartmentHistory
 sqlserver_hr.EmployeePayHistory
 sqlserver_hr.JobCandidate
 sqlserver_hr.Shift
(6 rows)
```

Имена объектов созданы с учетом регистра символов, поэтому обращаться к ним в PostgreSQL следует в двойных кавычках:

```
test=# SELECT "DepartmentID", "Name", "GroupName"
FROM sqlserver_hr."Department"
LIMIT 4;
 DepartmentID |    Name     |        GroupName
--------------+-------------+------------------------
            1 | Engineering | Research and Development
            2 | Tool Design | Research and Development
            3 | Sales       | Sales and Marketing
            4 | Marketing   | Sales and Marketing
(4 rows)
```

В настоящий момент `tds_fdw` поддерживает только чтение, но не изменение данных.

PostgreSQL

Создаем расширение и обертку:

```
test=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

Будем подключаться к другой базе данных этого же кластера, поэтому при создании стороннего сервера достаточно указать только параметр `dbname`, а параметры `host`, `port` и другие можно опустить:

```
test=# CREATE SERVER postgres_srv
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (dbname 'demo');
CREATE SERVER
```

При сопоставлении пользователей этого же кластера баз данных пароль указывать не нужно:

```
test=# CREATE USER MAPPING FOR postgres
  SERVER postgres_srv
  OPTIONS (user 'postgres');
CREATE USER MAPPING
```

Импортируем все таблицы и представления, принадлежащие схеме bookings:

```
test=# IMPORT FOREIGN SCHEMA bookings
  FROM SERVER postgres_srv
  INTO public;
IMPORT FOREIGN SCHEMA
```

Проверяем:

```
test=# SELECT * FROM bookings LIMIT 3;
 book_ref |       book_date         | total_amount
----------+-------------------------+--------------
 000004   | 2015-10-12 14:40:00+03 |     55800.00
 00000F   | 2016-09-02 02:12:00+03 |    265700.00
 000010   | 2016-03-08 18:45:00+03 |     50900.00
 000012   | 2017-07-14 09:02:00+03 |     37900.00
 000026   | 2016-08-30 11:08:00+03 |     95600.00
(5 rows)
```

Подробнее про postgres_fdw можно почитать в документации: postgrespro.ru/doc/postgres-fdw.

Механизм оберток сторонних данных интересен и тем, что рассматривается сообществом как основа для создания встроенного в PostgreSQL шардинга. Шардирование напоминает секционирование: и в том, и в другом случае таблица разделяется по какому-либо признаку на несколько частей, хранящихся отдельно друг от друга. Разница в том, что

секции располагаются на одном и том же сервере, а шарды — на разных. Возможность секционирования существует в PostgreSQL довольно давно. Начиная с версии 10 этот механизм активно развивается: добавлен декларативный синтаксис, динамическое исключение секций, параллельная обработка, сделаны другие улучшения. В качестве секций можно использовать и внешние таблицы, и таким образом секционирование превращается в шардирование.

На этом пути еще предстоит многое сделать, чтобы шардированием действительно можно было пользоваться:

- не гарантируется согласованность: работа с внешними серверами ведется не в единой распределенной транзакции, а в отдельных локальных транзакциях;
- отсутствует возможность дублировать одни и те же данные на нескольких серверах для повышения отказоустойчивости;
- все необходимые действия по созданию таблиц на шардах и соответствующих внешних таблиц пока приходится выполнять вручную.

Часть из перечисленных задач уже решена в нашем экспериментальном расширении `pg_shardman`, доступном на github.com/postgrespro/shardman.

Для взаимодействия с базами PostgreSQL существует еще одно расширение, входящее в дистрибутив, — `dblink`. Оно позволяет явно управлять соединениями (подключаться, отключаться), выполнять запросы и получать результаты асинхронно: postgrespro.ru/doc/dblink.

X Обучение и сертификация

Документация

Для серьезной работы с PostgreSQL не обойтись без чтения документации. Это не только описание всех возможностей СУБД, но и исчерпывающее справочное руководство, которое всегда должно быть под рукой. Читая документацию, вы получаете емкую и точную информацию из первых рук — она написана самими разработчиками и всегда аккуратно поддерживается в актуальном состоянии.

В нашей компании Postgres Professional выполнен перевод всего комплекта документации PostgreSQL, включая самую последнюю версию, на русский язык — он доступен на сайте postgrespro.ru/docs.

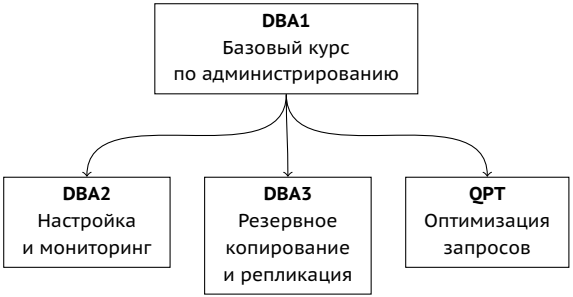
Глоссарий, составленный нами для перевода, опубликован по адресу postgrespro.ru/education/glossary. Мы рекомендуем использовать его, чтобы грамотно переводить англоязычные документы и использовать единую, понятную всем терминологию для материалов на русском языке.

Предпочитающие оригинальную документацию на английском языке найдут ее как на нашем сайте, так и по адресу postgresql.org/docs.

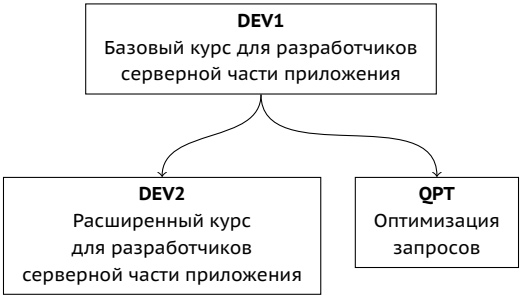
Учебные курсы

Мы разрабатываем учебные курсы для тех, кто начинает работать с PostgreSQL или повышает свою квалификацию.

Курсы для администраторов баз данных:



И для прикладных разработчиков:



Документация PostgreSQL содержит полные детальные сведения, которые, однако, разбросаны по разным главам и требуют многократного внимательного прочтения.

Курсы не заменяют документацию, а дополняют ее. Учебные модули последовательно и связно раскрывают содержание, выделяют важную и практически полезную информацию. Прохождение учебных курсов дает необходимую широту знаний, систематизирует ранее полученные отрывочные сведения и позволяет лучше ориентироваться в документации и быстро уточнять необходимые детали.

Каждая тема курса состоит из теоретической части и практики. Теория — это в большинстве случаев не только презентация, но и демонстрация работы на «живой» системе. Слушатели курса получают презентации с подробными комментариями к каждому слайду, результат работы демонстрационных скриптов, решения практических заданий, а в некоторых случаях и дополнительные справочные материалы.

Где и как пройти обучение

Для самостоятельного обучения и некоммерческого использования все материалы курсов, включая видеозаписи, доступны на нашем сайте всем желающим. Вы найдете их по адресу postgrespro.ru/education/courses.

Также вы можете пройти обучение по перечисленным курсам в одном из специализированных учебных центров под руководством опытного преподавателя. По окончании курса выдается сертификат слушателя. Список авторизованных нами учебных центров: postgrespro.ru/education/where.

DBA1. Базовый курс по администрированию PostgreSQL

Продолжительность: 3 дня

Предварительные знания:

Минимальные представления о базах данных и SQL.
Знакомство с Unix.

Какие навыки будут получены:

Общие сведения об архитектуре PostgreSQL.
Установка, базовая настройка, управление сервером.
Организация данных на логическом и физическом уровнях.
Базовые задачи администрирования.
Управление пользователями и доступом.
Представление о резервном копировании, восстановлении и репликации.

Темы:

Базовый инструментарий

1. Установка и управление сервером
2. Использование psql
3. Конфигурирование

Архитектура

4. Общее устройство PostgreSQL
5. Изоляция и многоверсионность
6. Буферный кеш и журнал

Организация данных

7. Базы данных и схемы

8. Системный каталог
9. Табличные пространства
10. Низкий уровень

Задачи администрирования

11. Мониторинг
12. Сопровождение

Управление доступом

13. Роли и атрибуты
14. Привилегии
15. Политики защиты строк
16. Подключение и аутентификация

Резервное копирование

17. Обзор

Репликация

18. Обзор

Материалы учебного курса доступны для самостоятельного изучения по адресу: postgrespro.ru/education/courses/DBA1.

DBA2. Настройка и мониторинг PostgreSQL

Продолжительность: 4 дня

Предварительные знания:

Основы языка SQL.
Владение ОС Unix.
Знакомство с PostgreSQL в объеме курса DBA1.

Какие навыки будут получены:

Настройка различных конфигурационных параметров исходя из понимания внутреннего устройства сервера.
Мониторинг сервера и использование полученных данных для итеративной настройки параметров.
Настройки, связанные с локализацией.
Управление расширениями и знакомство с процедурой обновления сервера.

Темы:

Многоверсионность

1. Изоляция
2. Страницы и версии строк
3. Снимки данных
4. HOT-обновления
5. Очистка
6. Автоочистка
7. Заморозка

Журналирование

8. Буферный кеш
9. Журнал предзаписи
10. Контрольная точка
11. Настройка журнала

Блокировки

12. Блокировки объектов
13. Блокировки строк
14. Блокировки в оперативной памяти

Задачи администрирования

15. Управление расширениями
16. Локализация
17. Обновление сервера

Материалы учебного курса доступны для самостоятельного изучения по адресу: postgrespro.ru/education/courses/DBA2.

DBA3. Резервное копирование и репликация PostgreSQL

Продолжительность: 2 дня

Предварительные знания:

Основы языка SQL.
Владение ОС Unix.
Знакомство с PostgreSQL в объеме курса DBA1.

Какие навыки будут получены:

Выполнение резервного копирования.
Настройка серверов для физической и логической репликации.
Знакомство со сценариями использования репликации.
Представление о способах построения кластеров.

Темы:

Резервное копирование

1. Логическое резервирование
2. Базовая резервная копия
3. Архив журнала предзаписи

Репликация

4. Физическая репликация
5. Переключение на реплику
6. Логическая репликация
7. Сценарии использования

Кластерные технологии

8. Обзор

Материалы учебного курса доступны для самостоятельного изучения по адресу: postgrespro.ru/education/courses/DBA3.

DEV1. Базовый курс по разработке серверной части приложений

Продолжительность: 4 дня

Предварительные знания:

Основы языка SQL.
Опыт работы с каким-нибудь процедурным языком программирования.
Минимальные представления о работе в Unix.

Какие навыки будут получены:

Общие сведения об архитектуре PostgreSQL.
Использование основных объектов БД.
Программирование на стороне сервера на языках SQL и PL/pgSQL.
Использование основных типов данных, включая записи и массивы.
Организация взаимодействия с клиентской частью.

Темы:

Базовый инструментарий

1. Установка и управление; psql

Архитектура

2. Общее устройство PostgreSQL
3. Изоляция и многоверсионность
4. Буферный кеш и журнал

Организация данных

5. Логическая структура
6. Физическая структура

Приложение «Книжный магазин»

7. Схема данных приложения

SQL

8. Функции
9. Процедуры
10. Составные типы

PL/pgSQL

11. Обзор и конструкции языка
12. Выполнение запросов
13. Курсоры
14. Динамические команды
15. Массивы
16. Обработка ошибок
17. Триггеры
18. Отладка

Разграничение доступа

19. Обзор разграничения доступа

Резервное копирование

20. Логическое резервирование

Материалы учебного курса доступны для изучения по адресу: postgrespro.ru/education/courses/DEV1.

DEV2. Расширенный курс по разработке серверной части приложений

Продолжительность: 4 дня

Предварительные знания:

Общие представления об архитектуре PostgreSQL.
Уверенное владение SQL и PL/pgSQL.
Минимальные представления о работе в Unix.

Какие навыки будут получены:

Понимание внутренней организации сервера.
Полное использование возможностей, предоставляемых PostgreSQL для реализации логики приложения.
Расширение возможностей СУБД для решения специальных задач.

Темы:

Архитектура

1. Изоляция
2. Внутреннее устройство
3. Очистка
4. Журналирование
5. Блокировки

«Книжный магазин»

6. Приложение 2.0

Расширяемость

7. Пул соединений
8. Типы для больших значений
9. Пользовательские типы данных
10. Классы операторов

11. Слабоструктурированные данные
12. Фоновые процессы
13. Асинхронная обработка
14. Создание расширений
15. Языки программирования
16. Агрегатные и оконные функции
17. Полнотекстовый поиск
18. Физическая репликация
19. Логическая репликация
20. Внешние данные

Материалы учебного курса доступны для изучения по адресу: postgrespro.ru/education/courses/DEV2.

QPT. Оптимизация запросов PostgreSQL

Продолжительность: 2 дня

Предварительные знания:

Знакомство с ОС Unix.
Уверенное владение SQL.
Владение языком PL/pgSQL будет полезно, но не является обязательным.
Знакомство с PostgreSQL в объеме курса DBA1 (для администраторов) или DEV1 (для разработчиков).

Какие навыки будут получены:

Детальное понимание механизмов планирования и выполнения запросов.
Настройка параметров экземпляра, связанных с производительностью.
Поиск проблемных запросов и их оптимизация.

Темы:

1. Демобаза «Авиаперевозки»
2. Выполнение запросов
3. Последовательный доступ
4. Индексный доступ
5. Сканирование по битовой карте
6. Соединение вложенным циклом
7. Соединение хешированием
8. Соединение слиянием
9. Статистика
10. Профилирование
11. Приемы оптимизации

Материалы учебного курса доступны для самостоятельного изучения по адресу: postgrespro.ru/education/courses/QPT.

Профессиональная сертификация

Программа профессиональной сертификации, запущенная в 2019 году, полезна как самим специалистам, так и работодателям. Владельцы сертификатов могут получить дополнительные преимущества при поиске работы и обсуждении уровня оплаты труда. К тому же это возможность подтвердить свой уровень знаний, пользуясь независимой системой оценки.

Для работодателей программа облегчает поиск новых специалистов и позволяет проверить уровень владения PostgreSQL у имеющихся, дает возможность контролировать

качество полученных знаний при направлении сотрудников на обучение, позволяет убедиться в компетентности сотрудников компаний-партнеров и поставщиков услуг.

Сейчас сертифицирование доступно только администраторам баз данных, но в дальнейшем планируется распространить программу и на разработчиков приложений.

Сертификация предполагает три уровня, для достижения каждого из которых потребуется пройти ряд тестов.

Уровень «Профессионал» подтверждает знания в следующих областях:

- общие представления об архитектуре PostgreSQL;
- варианты установки сервера, навыки работы в psql, управление настройками конфигурации;
- организация данных на логическом и физическом уровне;
- управление пользователями и доступом;
- общие представления о резервном копировании и репликации баз данных.

Для получения сертификата необходимо успешно пройти тест по курсу DBA1.

Уровень «Эксперт» дополнительно подтверждает знания в следующих областях:

- внутреннее устройство PostgreSQL;
- мониторинг и настройка сервера, выполнение задач сопровождения;
- решение задач оптимизации производительности, настройки запросов;
- выполнение резервного копирования;
- настройка физической и логической репликации для различных сценариев работы.

Для получения сертификата необходимо иметь сертификат уровня «Профессионал» и успешно пройти тесты по курсам DBA2, DBA3, QPT.

Уровень «Мастер» дополнительно подтверждает практические навыки администрирования PostgreSQL.

Для получения сертификата необходимо иметь сертификат уровня «Эксперт» и успешно пройти практический тест. Этот тип сертификации находится в разработке.

Зарегистрируйтесь на postgrespro.ru/user и запишитесь на тестирование в личном кабинете.

Для успешной сдачи тестов необходимо:

- уверенно владеть материалом соответствующих курсов и быть знакомым с разделами документации, на которые в курсах приводятся ссылки;
- иметь навыки практической работы с PostgreSQL в среде psql.

Во время тестирования доступны материалы курсов и документация к PostgreSQL, но любыми другими источниками информации пользоваться запрещено.

Достижение очередного уровня подтверждается сертификатом. Сертификат бессрочен, но привязан к конкретной версии сервера и устаревает вместе с ней, так что по прошествии нескольких лет может возникнуть необходимость пройти тестирование по более актуальной версии PostgreSQL.

Подробнее о программе сертификации читайте на сайте postgrespro.ru/education/cert.

Курсы для вузов

Одним из важнейших направлений деятельности нашей компании является подготовка кадров в области систем управления базами данных. Начинать готовить будущих специалистов необходимо уже с учебной скамьи, а это возможно только при взаимодействии с высшими учебными заведениями.

Мы предлагаем несколько учебных курсов, которые являются результатом сотрудничества компании с опытными преподавателями ведущих вузов. Материал рассчитан на студентов бакалавриата, имеющих базовую подготовку по программированию. Все курсы свободны для использования в образовательной деятельности. В распоряжении преподавателей — учебные пособия, слайды презентаций и видеозаписи лекций, а также другие учебные материалы, представленные на нашем сайте postgrespro.ru/education/university.

Курсы, разработанные при участии компании, читаются в таких вузах, как Московский государственный университет им. М. В. Ломоносова, Высшая школа экономики, Московский авиационный институт, Сибирский государственный университет науки и технологий им. М. Ф. Решетнева и Сибирский федеральный университет. Если вы являетесь представителем вуза и заинтересованы во внедрении курсов по базам данных в учебный план, свяжитесь с нами.

Также мы приглашаем к сотрудничеству преподавателей, готовых разрабатывать новые авторские курсы с использованием PostgreSQL. Мы, в свою очередь, оказываем поддержку, консультируем, редактируем рукописи и доводим их до публикации, организуем для авторов открытые лекции в ведущих вузах страны.

Основы языка SQL

Слушатели курса без предварительной подготовки смогут разобраться, что представляет собой система PostgreSQL, и научатся с ней работать. Начиная с разработки простых запросов на языке SQL слушатели постепенно осваивают более сложные конструкции, знакомятся с концепцией транзакций и оптимизацией производительности.

В основе курса лежит учебное пособие «PostgreSQL. Основы языка SQL».

Содержание:

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 (печатное издание)
ISBN 978-5-6041193-2-7 (электронное издание)

В электронном виде книга доступна на нашем сайте: postgrespro.ru/education/books/sqlprimer.

Курс состоит из 36 часов лекционных и практических занятий. На протяжении нескольких лет он постоянно читается автором в ведущих вузах Москвы и Красноярска. Материалы курса доступны по адресу postgrespro.ru/education/university/sqlprimer.

Евгений Павлович Моргунов, кандидат технических наук, доцент кафедры информатики и вычислительной техники Сибирского государственного университета науки и технологий имени академика М. Ф. Решетнева.

Живет в Красноярске. До перехода в вуз в 2000-м году более десяти лет работал программистом, в том числе занимался разработкой прикладной системы для банка. Познакомился с СУБД PostgreSQL в 1998 году. Сторонник использования в учебном процессе открытого и свободного программного обеспечения. По его инициативе в ходе изучения дисциплины «Технология программирования» стали применяться операционная система FreeBSD и система управления базами данных PostgreSQL. Член Международного общества инженерной педагогики (IGIP). Опыт использования PostgreSQL в преподавании составляет более двадцати лет.

Основы технологий баз данных

Современный университетский курс, сочетающий глубокую теоретическую составляющую с актуальными практическими аспектами применения и проектирования систем управления базами данных.

Первая часть содержит основные сведения о системах управления базами данных: реляционная модель данных, язык SQL, обработка транзакций.

Во второй части подробно рассмотрены технологии, лежащие в основе функционирования СУБД, и тенденции их развития. Некоторые темы изучаются повторно на более глубоком уровне.

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

ISBN 978-5-97060-841-8 (печатное издание)
ISBN 978-5-6041193-5-8 (электронное издание)

Содержание книги:

В электронном виде книга доступна на нашем сайте: postgrespro.ru/education/books/dbtech.

Курс рассчитан на 24 часа лекционных и 8 часов практических занятий. Он был прочитан Борисом Асеновичем Новиковым на факультете ВМК МГУ им. М. В. Ломоносова. Материалы курса доступны по адресу postgrespro.ru/education/university/dbtech.

Борис Асенович Новиков, доктор физико-математических наук, профессор департамента информатики НИУ ВШЭ в Санкт-Петербурге.

Научные интересы в основном связаны с различными аспектами проектирования, разработки и применения систем управления базами данных и их приложений, а также распределенных масштабируемых систем для обработки и анализа больших потоков данных.

Горшкова Екатерина Александровна, кандидат физико-математических наук.

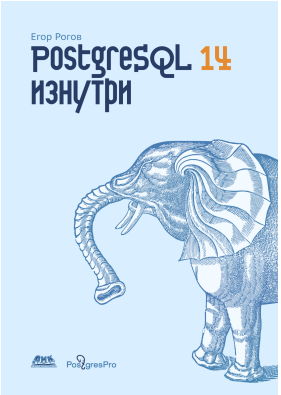
Специалист в проектировании высоконагруженных приложений с интенсивным использованием данных. В область научных интересов входит машинное обучение, анализ потоковых данных, информационный поиск.

Графеева Наталья Генриховна, кандидат физико-математических наук, доцент кафедры информационно-аналитических систем СПбГУ.

Научные интересы связаны с базами данных, информационным поиском, большими данными и интеллектуальным анализом данных. Имеет значительный опыт разработки, проектирования и сопровождения информационных систем, разработки и преподавания учебных курсов.

PostgreSQL изнутри

Эта книга для тех, кого не устраивает работа с базой данных как с черным ящиком. Книга рассчитана на читателей, имеющих некоторый опыт использования PostgreSQL. Она будет полезной и тем, кто хорошо знаком с устройством другой СУБД, но переходит на PostgreSQL и хочет разобраться в отличиях.



В книге вы не найдете готовых рецептов, зато понимание внутренней механики позволит критически переосмысливать чужой опыт и делать собственные выводы. Автор объясняет подробности устройства PostgreSQL и показывает, как проводить эксперименты и самостоятельно проверять неумолимо устаревающие сведения.

Рогов Е. В.
PostgreSQL изнутри. — М.: ДМК Пресс, 2022. — 660 с.

ISBN 978-5-93700-122-1 (печатное издание)
ISBN 978-5-6041193-9-6 (электронное издание)

Егор Рогов работает в образовательном отделе Postgres Professional с 2015 года: разрабатывает и читает учебные курсы, публикует статьи, пишет и редактирует книги.

Содержание книги:

В электронном виде книга доступна на нашем сайте: postgrespro.ru/education/books/internals.

XI Путеводитель по галактике

Новости и обсуждения

Знакомиться с новостями, узнавать о новых возможностях предстоящего выпуска PostgreSQL и вообще оставаться в курсе событий легко может любой желающий.

Множество интересных и полезных материалов публикуется в различных тематических блогах. Так, с полной подборкой заметок на английском удобно знакомиться на сайте-агрегаторе planet.postgresql.org, а многочисленные статьи на русском публикуются (в том числе и нашей компанией) на Хабре: habr.com/hub/postgresql. Обратите внимание на каналы youtube.com/RuPostgres и youtube.com/PostgresTV.

Существует также вики-проект wiki.postgresql.org с ответами на типичные вопросы, обучающими материалами и статьями про настройку и оптимизацию, про особенности миграции с разных СУБД и многое другое. Часть материалов оттуда доступна и на русском языке по адресу wiki.postgresql.org/wiki/Russian. Не забывайте, что каждый может помочь сообществу, переведя заинтересовавшую англоязычную статью.

Более 9 000 русскоязычных пользователей подписаны на телеграм-канал «pgsql — PostgreSQL» (t.me/pgsql); более 4 000 состоят в группе «PostgreSQL в России» на фейсбуке (facebook.com/groups/postgresql).

Можно задавать вопросы и на профильных сайтах, например, на stackoverflow.com (на английском языке) или ru.stackoverflow.com (на русском; в любом случае не забудьте поставить метку «postgresql»). К сожалению, известный форум sql.ru закрылся, но в сети можно найти архив.

Свои собственные новости компания Postgres Professional публикует по адресу postgrespro.ru/blog.

Списки рассылки

Не обязательно дожидаться, пока кто-нибудь напишет заметку в блоге, — можно подписаться на список рассылки. Разработчики PostgreSQL по старой традиции обсуждают все вопросы исключительно по электронной почте.

Полный перечень всех списков рассылки находится по адресу postgresql.org/list. Среди них:

- pgsql-hackers (обычно называемый просто «hackers») — основной список по всему, что касается разработки;
- pgsql-general для обсуждения общих вопросов;
- pgsql-bugs для сообщений о найденных ошибках;
- pgsql-docs для обсуждения документации;
- pgsql-translators для переводчиков;

- pgsql-announce для новостей о выходе новых версий продуктов...

...и многие другие.

Подписавшись на любой из этих списков, вы будете регулярно получать сообщения по электронной почте, а при желании сможете и принять участие в дискуссии. Другой вариант — читать архив сообщений на postgresql.org/list или на сайте нашей компании postgrespro.ru/list.

Commitfest

Еще один способ быть в курсе событий, не тратя на это много времени — заглядывать на commitfest.postgresql.org. В этой системе периодически открываются «окна», в которых разработчики должны регистрировать свои патчи. Например, окно 01.03.2022–31.03.2022 относилось к версии PostgreSQL 15, а следующее за ним окно 01.07.2022–31.07.2022 — уже к следующей. Это делается для того, чтобы примерно за полгода до выхода новой версии PostgreSQL прекратить прием новых возможностей и успеть стабилизировать код.

Патчи проходят несколько этапов: рецензируются и исправляются по результатам рецензии, а потом либо принимаются, либо переносятся в следующее окно, либо — если совсем не повезло — отвергаются.

Так можно узнавать как об уже включенных, так и о предполагающихся к включению в очередную версию возможностях.

Конференции

В России регулярно проводятся две крупные международные конференции, собирающие сотни пользователей и разработчиков PostgreSQL:

PGConf в Москве (pgconf.ru);

PGDay в Санкт-Петербурге (pgday.ru).

Периодически проходят и региональные конференции PGConf; например, **PGConf.Сибирь** проводилась в Новосибирске и Красноярске.

Кроме того, в разных городах России проводятся конференции с более широкой тематикой, на которых представлено направление баз данных, в том числе и PostgreSQL. Отметим лишь несколько:

CodeFest в Новосибирске (codefest.ru);

HighLoad++ в Москве и других городах (highload.ru).

Разумеется, конференции проводятся и в других странах. Самые крупные из них — это:

PGCon в Оттаве (pgcon.org);

Европейская **PGConf Europe** (pgconf.eu).

Помимо конференций, проходят и неофициальные встречи, в том числе онлайн.

XII О компании

Компания Postgres Professional была основана в 2015 году и объединила ключевых российских разработчиков, вклад которых в развитие PostgreSQL признан мировым сообществом. Здесь готовят квалифицированные отечественные кадры в области разработки СУБД. В настоящее время в ней работает около 150 программистов, архитекторов и инженеров.

Postgres Professional выпускает несколько версий системы Postgres Pro, построенной на основе PostgreSQL, выполняет разработки на уровне ядра СУБД и расширений, оказывает услуги по проектированию и поддержке прикладных систем и миграции на PostgreSQL.

Компания уделяет большое внимание образовательной деятельности, организует крупнейшую ежегодную международную конференцию PgConf.Russia в Москве и принимает участие в конференциях по всему миру.

Контактная информация:

117036, г. Москва, ул. Дмитрия Ульянова, д. 7А

+7 495 150-06-91

info@postgrespro.ru

СУБД Postgres Pro

Postgres Pro компании Postgres Professional — российская коммерческая СУБД, основанная на свободно распространяемой СУБД PostgreSQL и разработанная сообразно с требованиями корпоративных заказчиков. Входит в реестр российского ПО.

Postgres Pro Standard содержит все функциональные возможности PostgreSQL и дополнено различными расширениями и патчами, в том числе еще не принятыми сообществом. Клиент получает доступ к полезному функционалу и выигрывает в производительности, не дожидаясь очередного релиза PostgreSQL.

Postgres Pro Enterprise — это глубоко переработанная версия СУБД, благодаря большей надежности и повышенной производительности пригодная для решения серьезных промышленных задач.

Обе версии Postgres Pro, дополненные необходимыми средствами защиты информации, прошли **сертификацию ФСТЭК**.

Для использования любой версии Postgres Pro необходимо приобрести лицензию. Можно бесплатно получить интересующую вас версию СУБД для тестирования, изучения возможностей СУБД и разработки прикладного программного обеспечения.

Подробнее о возможностях и отличиях версий Postgres Pro читайте на сайте: postgrespro.ru/products/postgrespro

Услуги

Отказоустойчивые решения для СУБД Postgres

Проектирование и участие в создании высоконагруженных, высокопроизводительных и отказоустойчивых промышленных систем; консалтинговые услуги. Внедрение СУБД Postgres и оптимизация конфигурации.

Вендорская техническая поддержка

Техподдержка Postgres Pro и PostgreSQL круглосуточно и без выходных. Мониторинг, восстановление работоспособности, анализ непредвиденных обстоятельств, повышение производительности, исправление ошибок в СУБД и расширениях.

Миграция прикладных систем на СУБД Postgres

Оценка сложности миграции с других СУБД на Postgres. Разработка архитектуры нового решения и необходимых доработок. Миграция прикладных систем на СУБД Postgres и поддержка в процессе миграции.

Обучение Postgres

Обучение администраторов баз данных, разработчиков и архитекторов прикладных систем особенностям СУБД Postgres и эффективному использованию ее преимуществ.

Аудит СУБД

Экспертная оценка состояния СУБД. Аудит информационной безопасности систем на основе Postgres.

Полное описание услуг: postgrespro.ru/services

Лузанов Павел Вениаминович
Рогов Егор Валерьевич
Лёвшин Игорь Викторович

Postgres. Первое знакомство

Редактор: Петр Лагуткин
Дизайнер обложки: Александр Груздев

9-е издание, переработанное и дополненное

postgrespro.ru/education/books/introbook

Москва, Постгрес Профессиональный, 2023

ISBN 978-5-6045970-1-9