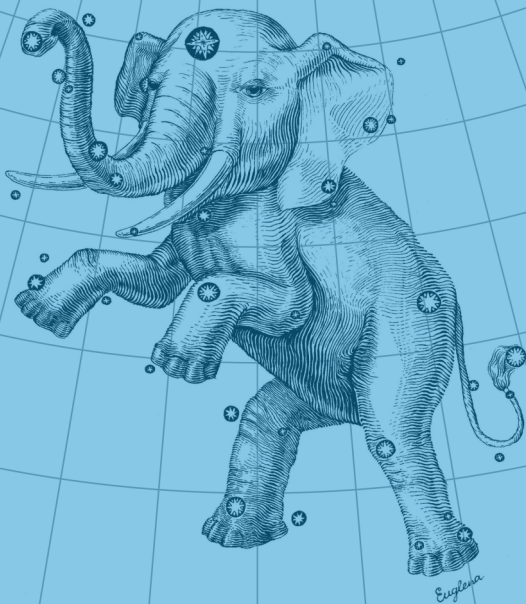


PostgreSQL



П. Лузанов, Е. Рогов, И. Лёвшин

ДЛЯ НАЧИНАЮЩИХ

v.4

Предисловие

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

- Что вообще такое, этот PostgreSQLс.2
- Как установить PostgreSQL на Linux и Windows ...с.16
- Как подключиться к серверу, начать писать SQL-запросы и применять транзакции с.29
- Как продолжить самостоятельное изучение SQL с помощью демобазы с.59
- Про возможности полнотекстового поиска с.88
- Про работу с данными в формате JSONс.96
- Как использовать PostgreSQL в качестве базы данных для вашего приложения.....с.106
- Про полезную программу pgAdmin с.120
- Где найти документацию и пройти обучение с.128
- Как быть в курсе происходящегос.139
- И немного про компанию Postgres Professional . с.143

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

Электронная версия книги доступна по адресу:
postgrespro.ru/education/introbook

Желаем удачи!

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 Майкл Стоунбрейкер уже 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, описыва-

ющих модель данных, язык запросов POSTQUEL (в то время SQL еще не был общепризнанным стандартом) и устройство хранилища данных.

POSTGRES иногда еще 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 требовало новых приложений, а от баз данных – поддержки новых 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 испытание временем, и было выбрано новое имя – PostgreSQL, которое отражает связь и с оригинальным проектом POSTGRES, и с переходом на SQL. Именно поэтому PostgreSQL произносится «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) проекта. В настоящий момент он состоит из пяти 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 группы разработчиков со временем меняется, появляются новые члены, кто-то отходит от проекта. Актуальный список разработчиков поддерживается на официальном сайте PostgreSQL www.postgresql.org.

Вклад российских разработчиков в PostgreSQL весьма значителен. Это, пожалуй, самый крупный глобальный проект с открытым исходным кодом с таким широким российским представительством.

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 настоящий момент в число основных разработчиков входят трое представителей России – Олег Бартунов, Федор Сигаев и Александр Коротков, – основавших в 2015 году компанию Postgres Professional. Среди крупных направлений выполненных ими работ можно выделить локализацию PostgreSQL (поддержка национальных кодировок и Unicode), систему полнотекстового поиска, работу с массивами и слабо-структурированными данными (hstore, json, jsonb), новые методы индексации (GiST, SP-GiST, GIN и RUM, Bloom). 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 новый релиз.

В некоторый момент 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 версия (major version) 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 – она и является последней актуальной версией PostgreSQL. Следующая версия, запланированная на осень 2018 года, будет иметь номер 11.

В процессе работы над новой версией находят и исправляются ошибки. Наиболее критические из них исправляются не только в текущей, но и в предыдущих версиях. По мере накопления таких исправлений принимается решение о выпуске дополнительных версий (minor version), совместимых со старыми. Например, версия 9.6.6 содержит только исправления ошибок, найденных в 9.6, а 10.2 будет содержать исправления для версии 10.

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 (www.postgrespro.ru), предоставляя услуги по поддержке в режиме 24x7.

Современное состояние

PostgreSQL является одной из самых популярных баз данных. За более чем 20-летнюю историю развития на прочном фундаменте, заложенном академической разработкой, PostgreSQL выросла в полноценную СУБД уровня предприятия и составляет реальную альтернативу коммерческим базам данных. Чтобы убедиться в этом, достаточно посмотреть на важнейшие характеристики новейшей на сегодняшний день версии PostgreSQL 10.

Надежность и устойчивость

Вопросы обеспечения надежности особенно важны в приложениях уровня предприятия для работы с критически важными данными. С этой целью PostgreSQL позволяет настраивать горячее резервирование, восстановление на заданный момент времени в прошлом, различные виды репликации (синхронную, асинхронную, каскадную).

Безопасность

PostgreSQL позволяет работать по защищенному SSL-соединению и предоставляет большое количество методов аутентификации, включая аутентификацию по паролю, клиентским сертификатам, а также с помощью внешних сервисов (LDAP, RADIUS, PAM, Kerberos).

При управлении пользователями и доступом к объектам БД предоставляются следующие возможности:

- Создание и управление пользователями 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, выпущенная компанией Postgres Professional, сертифицирована ФСТЭК для использования в системах обработки конфиденциальной информации и персональных данных.

Соответствие стандартам

По мере развития стандарта ANSI SQL его поддержка постоянно добавлялась в PostgreSQL. Это относится

ко всем версиям стандарта: SQL:92, SQL:1999, SQL:2003, SQL:2008, SQL:2011. Поддержку стандартизированной в недавней версии SQL:2016 работы с JSON планируется добавить в PostgreSQL 11. В целом PostgreSQL обеспечивает высокий уровень соответствия стандарту и поддерживает 160 из 179 обязательных возможностей, а также большое количество необязательных.

Поддержка транзакционности

PostgreSQL обеспечивает полную поддержку свойств ACID и обеспечивает эффективную изоляцию транзакций. Для этого в PostgreSQL используется механизм многоверсионного управления одновременным доступом (MVCC). Он позволяет обходиться без блокировок во всех случаях, кроме одновременного изменения одной и той же строки данных в нескольких процессах. При этом читающие транзакции никогда не блокируют пишущие транзакции, а пишущие — читающих. Это справедливо и для самого строгого уровня изоляции serializable, который, используя инновационную систему Serializable Snapshot Isolation, обеспечивает полное отсутствие аномалий сериализации и гарантирует, что при одновременном выполнении транзакций результат будет таким же, как и при последовательном выполнении.

Для разработчиков приложений

Разработчики приложений получают в свое распоряжение богатый инструментарий, позволяющий реализовать

приложения любого типа:

- Возможность использования различных языков для серверного программирования: встроенного PL/pgSQL (удобного тесной интеграцией с SQL), С для критичных по производительности задач, Perl, Python, Tcl, а также JavaScript, Java и других;
- 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, ограничения целостности, представления и материализованные представления, последовательности, секционирование, подзапросы и with-запросы (в том числе рекурсивные), агрегатные и оконные функции, хранимые функции, триггеры и т. д.;
- Встроенная гибкая система полнотекстового поиска с поддержкой русского и всех европейских языков, дополненная эффективным индексным доступом;
- Поддержка слабоструктурированных данных в духе NoSQL: хранилище пар «ключ-значение» hstore, xml, json (как в текстовом, так и в 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 – его производительность растет практически линейно с увеличением количества ядер.

Начиная с версии 9.6 PostgreSQL умеет работать с данными в параллельном режиме, что на сегодняшний день включает параллельное чтение (включая индексное сканирование), соединение и агрегации. Это еще больше повышает возможности использования аппаратных средств для ускорения операций.

Планировщик запросов

В PostgreSQL используется планировщик запросов, основанный на стоимости. Используя собираемую статистику и учитывая в своих математических моделях как дисковые операции, так и время работы процессора, планировщик позволяет оптимизировать самые сложные запросы. В его распоряжении находятся все методы доступа к данным и способы выполнения соединений, характерные для передовых коммерческих СУБД.

Возможности индексирования

В PostgreSQL реализованы различные методы индексирования. Помимо традиционных B-деревьев, также доступны:

- GiST – обобщенное сбалансированное дерево поиска, которое может применяться для данных, не допускающих упорядочения. Примерами могут служить R-деревья для индексирования точек на плоскости с возможностью поиска ближайших соседей (k-NN search) и индексирование операции пересечения интервалов;
- SP-GiST – обобщенное несбалансированное дерево поиска, основанное на разбиении области значений на непересекающиеся вложенные области. Примерами могут служить дерево квадрантов и префиксное дерево;
- GIN – обобщенный инвертированный индекс. Основной областью применения является полнотекстовый поиск, где требуется находить документы, в которых встречается 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, позволяет ускорить фразовый поиск и сразу выдавать результаты упорядоченными по релевантности;
- BRIN – индекс небольшого размера, позволяющий найти компромисс между размером индекса и скоростью поиска. Эффективен на больших кластеризованных таблицах;
- Bloom – индекс, основанный на фильтре Блума (появился в версии 9.6). Такой индекс, имея очень ком-

пактное представление, позволяет быстро отсеять заведомо ненужные строки, однако требует переверки оставшихся.

За счет расширяемости набор доступных методов индексного доступа постоянно увеличивается.

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

Несколько индексов могут автоматически объединяться с помощью битовой карты для ускорения доступа.

Кроссплатформенность

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

За счет открытого и переносимого кода на языке C PostgreSQL можно собрать 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 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 – система обработки геоинформационных данных.

Только стандартный комплект, входящий в сборку PostgreSQL 10, содержит около полусотни расширений, доказавших свою надежность и полезность.

Доступность

Лицензия PostgreSQL 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 конкретного вендора, тем самым в любой ситуации сохраняя вложенные в них инвестиции.

Установка и начало работы

Что нужно для начала работы с PostgreSQL? В этой главе мы объясним, как установить службу PostgreSQL и управлять ей, а потом создадим простую базу данных и покажем, как создать в ней таблицы. Мы расскажем и про основы языка SQL, на котором формулируются запросы. Будет неплохо, если вы сразу начнете пробовать команды по мере чтения.

Мы будем использовать дистрибутив Postgres Pro Standard 10, разработанный в нашей компании Postgres Professional. Этот дистрибутив полностью совместим с обычной СУБД PostgreSQL и дополнительно включает в себя некоторые разработки, выполненные в нашей компании, а также ряд возможностей, которые будут доступны только в следующей версии PostgreSQL.

Итак, приступим. Установка и запуск сервера PostgreSQL зависит от того, какую операционную систему вы используете. Если у вас Windows, читайте дальше; если Linux семейства Debian или Ubuntu – переходите сразу к с. 24.

Инструкцию по установке для других операционных систем вы найдете на сайте нашей компании: postgrespro.ru/products/download.

Если нужной вам версии там не оказалось – воспользуйтесь обычным дистрибутивом PostgreSQL: инструкции находятся по адресу www.postgresql.org/download.

Windows

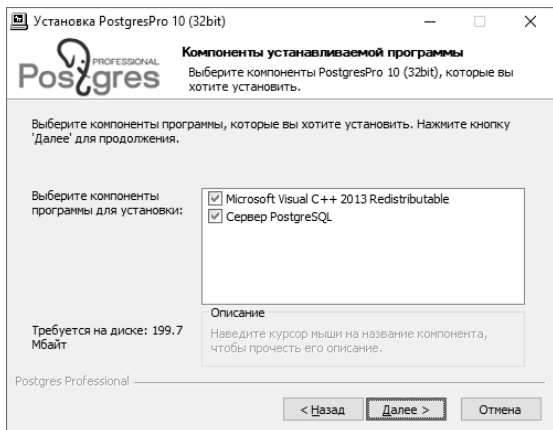
Установка

Скачайте установщик СУБД с нашего сайта: repo.postgrespro.ru/pgpro-10/win.

Выберите 32- или 64-разрядную версию в зависимости от того, какая у вас версия Windows. Запустите скачанный файл и выберите язык установки.

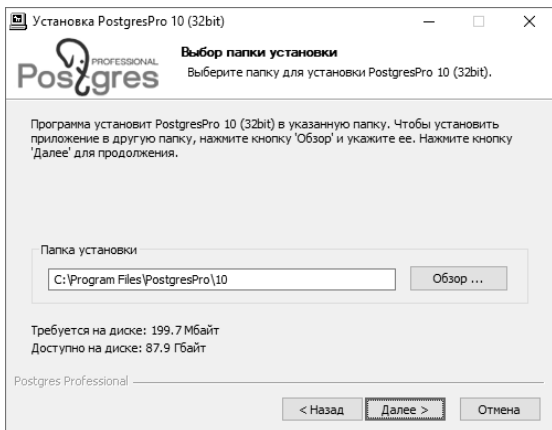
Установщик построен в традиционном стиле «мастера»: вы можете просто нажимать на кнопку «Далее», если вас устраивают предложенные варианты. Остановимся подробнее на основных шагах.

Компоненты устанавливаемой программы:

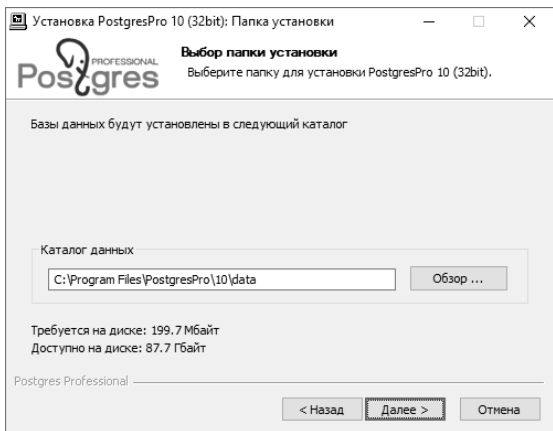


Оставьте оба флажка, если не уверены, какие выбрать.

Выбор папки установки:



По умолчанию PostgreSQL устанавливается в каталог C:\Program Files\PostgresPro\10 (или C:\Program Files (x86)\PostgresPro\10 для 32-разрядной версии на 64-разрядной системе).



Отдельно можно выбрать расположение каталога для баз данных. Именно здесь будет находиться хранящаяся в СУБД информация, так что убедитесь, что на диске достаточно места, если вы планируете хранить много данных.

Параметры сервера:

The screenshot shows the 'Параметры сервера' (Server Parameters) window for PostgreSQL 10 (32bit). The window title is 'Установка PostgresPro 10 (32bit)'. The PostgreSQL logo is visible in the top left. The main heading is 'Параметры сервера' with the instruction 'Пожалуйста, задайте параметры сервера'. The form contains the following fields and options:

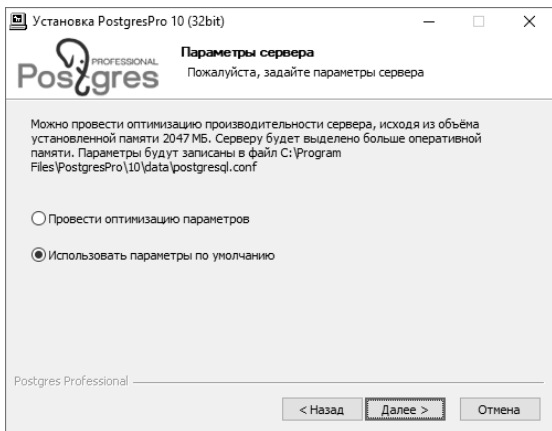
- Порт: 5432
- Адреса: Разрешить подключения с любых IP-адресов:
- Локаль: Настройка ОС (dropdown menu)
- Супер-пользователь: postgres
- Пароль: [masked with 7 dots]
- Подтверждение: [masked with 7 dots]
- Настроить переменные среды

At the bottom, there is a 'Postgres Professional' logo and three buttons: '< Назад', 'Далее >', and 'Отмена'.

Если вы планируете хранить данные на русском языке, выберите локаль «Russian, Russia» (или оставьте вариант «Настройка ОС», если в Windows используется русская локаль).

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

Остальные поля можно оставить со значениями по умолчанию.



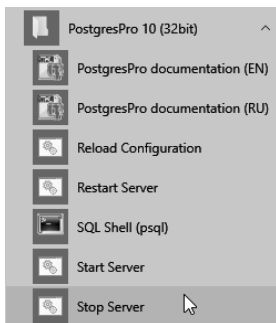
Если вы планируете установить PostgreSQL только для ознакомительных целей, можно отметить вариант «Использовать параметры по умолчанию», чтобы СУБД не занимала много оперативной памяти.

Управление службой и основные файлы

При установке Postgres Pro в вашей системе регистрируется служба «postgrespro-X64-10» (или «postgrespro-X86-10» для 32-разрядной версии). Она запускается автоматически при старте компьютера под учетной записью Network Service (Сетевая служба). При необходимо-

сти вы можете изменить параметры службы с помощью стандартных средств Windows.

Чтобы временно остановить службу сервера баз данных, выполните программу «Stop Server» из папки в меню «Пуск», которую вы указали при установке:



Для запуска службы там же есть программа «Start Server».

Если при запуске службы произошла ошибка, для поиска причины следует заглянуть в журнал сообщений сервера. Он находится в подкаталоге log того каталога, который был выбран при установке для баз данных (обычно это будет C:\Program Files\PostgresPro\10\data\log). Журнал настроен таким образом, 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. Вам не нужно их изменять, чтобы начать знакомство с PostgreSQL, но в реальной работе они непременно потребуются:

- `postgresql.conf` — 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.

Загляните в эти файлы — они прекрасно документированы.

Теперь мы готовы подключиться к базе данных и попробовать некоторые команды и запросы. Переходите к разделу «Пробуем SQL» на с. 29.

Debian и Ubuntu

Установка

Если вы используете Linux, то для установки необходимо подключить пакетный репозиторий нашей компании.

Для ОС Debian (в настоящее время поддерживаются версии 7 «Wheezy», 8 «Jessie» и 9 «Stretch») выполните в терминале следующие команды:

```
$ sudo apt-get install lsb-release
$ sudo sh -c 'echo "deb \
http://repo.postgrespro.ru/pgpro-10/debian \
$(lsb_release -cs) main" > \
/etc/apt/sources.list.d/postgrespro.list'
```

Для ОС Ubuntu (в настоящее время поддерживаются версии 14.04 «Trusty», 16.04 «Xenial», 16.10 «Yakkety», 17.04 «Zesty» и 17.10 «Artful») команды немного отличаются:

```
$ sudo sh -c 'echo "deb \
http://repo.postgrespro.ru/pgpro-10/ubuntu \
$(lsb_release -cs) main" > \
/etc/apt/sources.list.d/postgrespro.list'
```

Дальше все одинаково для обеих систем:

```
$ wget --quiet -O - http://repo.postgrespro.ru/pgpro-10/keys/GPG-KEY-POSTGRESPRO | sudo apt-key add -
$ sudo apt-get update
```

Перед установкой проверьте настройки локализации:

```
$ locale
```

Если вы планируете хранить данные на русском языке, значение переменных LC_STYPE и LC_COLLATE должно быть равно «ru_RU.UTF8» (значение «en_US.UTF8» тоже подходит, но менее предпочтительно). При необходимости установите эти переменные:

```
$ export LC_STYPE=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. Дистрибутив позволяет полностью управлять установкой, но для начала работы удобно использовать пакет, который выполнит всю установку и настройку в автоматическом режиме:

```
$ sudo apt-get install postgrespro-std-10
```

Как только эта команда выполнится, СУБД PostgreSQL будет 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 на вашей системе автоматически был создан специальный пользователь `postgres`, от имени которого работают процессы, обслуживающие сервер, и которому принадлежат все файлы, относящиеся к СУБД. PostgreSQL будет автоматически запускаться при перезагрузке операционной системы. С настройками по умолчанию это не проблема: если вы не работаете с сервером базы данных, он потребляет совсем немного ресурсов вашей системы. Если вы все-таки захотите отключить автозапуск, выполните:

```
$ sudo pg-setup service disable
```

Чтобы временно остановить службу сервера баз данных, выполните команду:

```
$ sudo service postgrespro-std-10 stop
```

Запустить службу сервера можно командой:

```
$ sudo service postgrespro-std-10 start
```

Полный список команд можно получить, выполнив:

```
$ sudo service postgrespro-std-10
```

Если при запуске службы произошла ошибка, для поиска причины следует посмотреть журнал сообщений сервера. Как правило, вы получите последние журнальные сообщения по команде:

```
$ sudo journalctl -xeu postgrespro-std-10
```

Но в некоторых старых версиях операционных систем вам может потребоваться заглянуть в файл `/var/lib/pgpro/std-10/pgstartup.log`.

Вся информация, которая попадает в базу данных, располагается в файловой системе в специальном каталоге `/var/lib/pgpro/std-10/data/`. Убедитесь, что у вас достаточно свободного места, если собираетесь хранить много данных.

Есть несколько важных конфигурационных файлов, которые определяют настройки сервера. Для начала работы вам не придется их изменять, но с ними лучше сразу познакомиться, потому что в дальнейшем эти файлы вам непременно понадобятся:

- `/var/lib/pgpro/std-10/data/postgresql.conf` – основной конфигурационный файл, содержащий значения параметров сервера;
- `/var/lib/pgpro/std-10/data/pg_hba.conf` – файл, определяющий настройки доступа. В целях безопасности по умолчанию доступ разрешен только с локального компьютера и только для пользователя ОС `postgres`.

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

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, входящий в любую сборку PostgreSQL, и поэтому он всегда под рукой. Безусловно, хорошо иметь настроенную под себя среду, но нет никакого резона оказаться беспомощным в незнакомом окружении.

Во-вторых, `psql` действительно удобен для повседневных задач по администрированию баз данных, для написания небольших запросов и автоматизации процессов, например, для периодической установки изменений

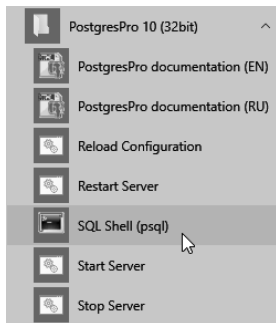
программного кода на сервер СУБД. Он имеет собственные команды, позволяющие сориентироваться в объектах, хранящихся в базе данных, и удобно представить информацию из таблиц.

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 Windows запустите программу «SQL Shell (psql)» из папки меню «Пуск», которую вы указали при установке:



В ответ на запрос введите пароль пользователя postgres, который вы указали при установке PostgreSQL.

Пользователи Windows могут столкнуться с неправильным отображением русских букв в терминале. В этом случае убедитесь, что свойствах окна терминала установлен TrueType-шрифт (обычно «Lucida Console» или «Consolas»).

В итоге и в одной, и в другой операционной системе вы увидите одинаковое приглашение postgres=#. «Postgres» здесь — имя базы данных, к которой вы сейчас подключены. Один сервер PostgreSQL может одновременно обслуживать несколько баз данных, но одновременно вы работаете только с одной из них.

Дальше мы будем приводить некоторые команды. Вводите только то, что выделено жирным шрифтом; приглашение и ответ системы на команду приведены исключительно для удобства.

База данных

Давайте 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 (так что команду можно разбить на несколько строк).

Теперь переключимся на созданную базу:

```
postgres=# \c test
```

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

```
test=#
```

Как вы 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 на косую черту, или попытайтесь найти аналог).

Команд psql довольно много, и с некоторыми из них мы познакомимся чуть позже, а полный список с кратким описанием можно получить прямо сейчас:

```
test=# \?
```

Поскольку справочная информация довольно объемна, она будет показана с помощью настроенной в операционной системе команды-пейджера; обычно это more или less.

Таблицы

В реляционных СУБД данные представляются в виде **таблиц**. Заголовок таблицы определяет **столбцы**; собственно данные располагаются в **строках**. Данные не упорядочены (в частности, нельзя полагаться на то, что строки хранятся в порядке их добавления в таблицу).

Для каждого столбца устанавливается **тип данных**; значения соответствующих полей строк должны удовлетворять этим типам. PostgreSQL располагает большим числом встроенных типов данных (postgrespro.ru/doc/datatype.html) и возможностями для создания новых, но мы ограничимся несколькими из основных:

- `integer` – целые числа;
- `text` – текстовые строки;
- `boolean` – логический тип, принимающий значения `true` (истина) или `false` (ложь).

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.)

В этой команде 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 мы можем определить ограничения целостности, которые будут автоматически проверяться — СУБД не допустит появление в базе некорректных данных. В нашем примере мы добавили ограничение `PRIMARY KEY` для столбца `c_no`, которое говорит о том, что значения в этом столбце должны быть уникальными, а неопределенные значения не допускаются. Такой столбец можно использовать для того, чтобы отличить одну строку в таблице от других. Полный список ограничений целостности: postgrespro.ru/doc/ddl-constraints.html.

Точный синтаксис команды `CREATE TABLE` можно посмотреть в документации, либо попросить справку прямо в `psql`:

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

Такая справка есть по каждой команде `SQL`, а полный список команд легко получить с помощью `\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
```

Если вам требуется массовая загрузка данных из внешнего источника, команда INSERT — не лучший выбор; посмотрите на специально предназначенную для этого команду COPY: postgrespro.ru/doc/sql-copy.html.

Для дальнейших примеров нам потребуется еще две таблицы: студенты и экзамены. Для каждого студента мы будем хранить его имя и год поступления; идентифицироваться он будет числовым 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
```

Экзамен содержит оценку, полученную студентом по некоторой дисциплине. 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 имени студента и номера курса. Такое ограничение целостности, относящее сразу к нескольким столбцам, определяется с помощью фразы 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 фразы REFERENCES мы определили два ограничения ссылочной целостности, называемые **внешними ключами**. Такие ключи показывают, что значения в одной таблице **ссылаются** на строки в другой таблице. Теперь при любых действиях СУБД будет проверять, что все идентификаторы s_id, указанные в таблице экзаменов, соответствуют реальным студентам (то есть записям в таблице студентов), а номера c_no – реальным курсам. Таким образом, будет исключена возможность оценить несуществующего студента или поставить оценку 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 SELECT. Например, выведем два столбца из таблицы courses:

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

Конструкция AS позволяет переименовать столбец, если это необходимо. 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 результирующей выборке мы можем получить несколько одинаковых строк. Даже если все строки были различны в исходной таблице, дубликаты могут появиться, если выводятся не все столбцы:

```
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.html#SQL-DISTINCT

Вообще после слова SELECT можно указывать и любые выражения. А без фразы FROM результирующая таблица будет содержать одну строку. Например:

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

Обычно при выборке данных требуется получить не все строки, а только удовлетворяющие какому-либо условию. Такое условие фильтрации записывается во фразе 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 отношения =, <> (или !=), >, >=, <, <=; 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 результирующую таблицу 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 отношения IS NULL (IS NOT NULL) и IS DISTINCT FROM (IS NOT DISTINCT FROM),

а также бывает удобно воспользоваться функцией `coalesce`.

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

Соединения

Грамотно спроектированная реляционная база данных не содержит избыточных данных. 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 во фразе `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 во фразе WHERE условие соединения. Получим 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 обе формы эквивалентны, так что можно использовать тот способ, который представляется более наглядным.

Этот пример показывает, что в результат не включаются строки исходной таблицы, для которых не нашлось пары в другой таблице: хотя условие наложено на дисциплины, но при этом исключаются и студенты, которые не сдавали экзамен по данной дисциплине. Чтобы в выборку попали все студенты, независимо от того, сдавали они экзамен или нет, надо использовать операцию внешнего соединения:

```
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)
```

В этом примере в результат добавляются 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 во фразе WHERE применяются к уже готовому результату соединений, поэтому, если вынести ограничение на дисциплины из условия соединения, Нина не попадет в выборку – ведь для нее exams.c_no не определен:

```
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.html#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 одну строку и 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 подзапрос, использованный в списке выражений SELECT, не содержит ни одной строки, возвращается неопределенное значение (как в последней строке результата примера).

Такие скалярные подзапросы можно использовать и 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 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)
```

Вариантом является отношение NOT IN, возвращающее противоположный результат. Например, список студентов, получивших только отличные оценки (то есть не получивших более низкие оценки):

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

```
 name | start_year
-----+-----
 Анна |      2014
 Нина |      2015
(2 rows)
```

Другая возможность – предикат EXISTS, проверяющий, что подзапрос возвратил 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
-----+-----
 Анна |      2014
  Нина |      2015
(2 rows)
```

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

В примерах выше мы уточняли имена столбцов названиями таблиц, чтобы избежать неоднозначности. Иногда этого недостаточно. Например, в запросе одна и та же таблица может участвовать два раза, или вместо таблицы в предложении FROM мы можем использовать безымянный подзапрос. В этих случаях после подзапроса можно указать произвольное имя, которое называется псевдонимом (alias). Псевдонимы можно использовать и для обычных таблиц.

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 обычно выбирают так, чтобы они были короткими, но оставались понятными.

Тот же запрос можно записать и без подзапросов, например, так:

```
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, но часто бывает важно получить строки результата в строго определенном порядке. Для этого 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 сначала по возрастанию оценки, для совпадающих оценок — по возрастанию номера студенческого билета, а при совпадении первых двух ключей — 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.html#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)

Аналогичную 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.html.

В запросах, использующих группировку, 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 можно также использовать столбцы таблицы-результата).

Выберем имена студентов, получивших более одной пятёрки 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.html#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.html.

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

```
test=# DELETE FROM exams WHERE score < 5;
DELETE 1
```

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

Транзакции

Давайте немного расширим нашу схему данных и распределим студентов по группам. При этом потребуем, чтобы у каждой группы в обязательном порядке был 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 в таблице студентов нам необходимо еще одно поле — номер группы, — о котором мы не подумали при создании. 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 поля 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.

Тут возникает затруднение. С одной стороны, мы не можем создать группу, не указав старосту. А с другой, как мы можем назначить Анну старостой, если она еще не входит в группу? Это привело бы к тому, что в базе данных некоторое время (пусть и небольшое) находились бы логически некорректные, несогласованные данные.

Мы столкнулись с тем, что две операции надо совершить одновременно, потому что ни одна из них не имеет смысла без другой. Такие операции, составляющие логически неделимую единицу работы, называются **транзакцией**.

Начнем транзакцию:

```
test=# BEGIN;  
BEGIN
```

Затем добавим группу вместе со старостой. Поскольку мы не помним наизусть номер студенческого билета Анны, 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
```

Откройте теперь 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 он 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 изменения:

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

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

```

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 гарантирует выполнение нескольких важных свойств.

Во-первых, транзакция либо выполняется целиком (как в нашем примере), либо не выполняется совсем. Если бы в 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, когда фиксируются изменения транзакции, все ограничения целостности должны быть выполнены, иначе транзакция прерывается. Таким образом, в начале транзакции данные находятся в согласованном состоянии, и в конце своей работы транзакция оставляет их согласованными; это свойство так и называется — **согласованность**.

В-третьих, как мы убедились на примере, другие пользователи никогда не увидят несогласованные данные,

которые транзакция еще не зафиксировала. Это свойство называется **изоляция**; 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 одновременном изменении одной и той же строки двумя 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 управления базами данных.

Подробнее о транзакциях:

postgrespro.ru/doc/tutorial-transactions.html

(и еще более подробно:

postgrespro.ru/doc/mvcc.html).

Полезные команды psql

\?	Справка по командам psql.
\h	Справка по SQL: список доступных команд или синтаксис конкретной команды.
\x	Переключает обычный табличный вывод (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.
\d+ имя	Еще более подробная информация по конкретному объекту.
\timing on	Показывать время 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`: для того, чтобы завершить сеанс работы, наберите

```
test=# \q
```

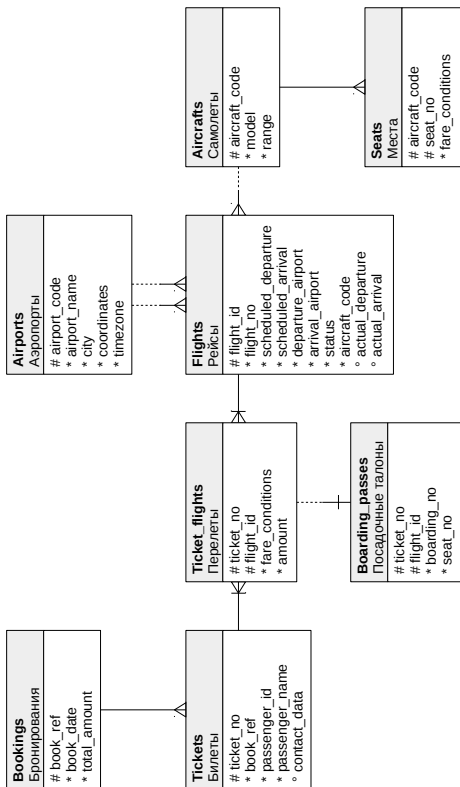
Демонстрационная база данных

Описание

Общая информация

Чтобы двигаться дальше и учиться писать более сложные запросы, нам понадобится более серьезная база данных — не три таблицы, а целых восемь, — и наполнение ее данными. Схема такой базы данных изображена в виде диаграммы «сущность-связи» на с. 60.

В качестве предметной области мы выбрали авиаперевозки: будем считать, что речь идет о нашей (пока еще несуществующей) авиакомпании. Тем, кто хотя бы раз летал на самолетах, эта область должна быть понятна; в любом случае мы сейчас все объясним. Хочется отметить, что мы старались сделать схему данных как можно проще, не загромождая ее многочисленными деталями, но, в то же время, не слишком простой, чтобы на ней можно было учиться писать интересные и осмысленные запросы.



Итак, основной сущностью является **бронирование** (bookings).

В одно бронирование можно включить несколько пассажиров, каждому из которых выписывается отдельный **билет** (tickets). Как таковой пассажир не является отдельной сущностью: для простоты можно считать, что все пассажиры уникальны.

Каждый билет включает один или несколько **перелетов** (ticket_flights). Несколько перелетов могут включаться в билет в нескольких случаях:

1. Нет прямого рейса, соединяющего пункты отправления и назначения (полет с пересадками);
2. Взят билет «туда и обратно».

В схеме данных нет жесткого ограничения, но предполагается, что все билеты в одном бронировании имеют одинаковый набор перелетов.

Каждый **рейс** (flights) следует из одного **аэропорта** (airports) в другой. Рейсы с одним номером имеют одинаковые пункты вылета и назначения, но будут отличаться датой отправления

При регистрации на рейс пассажиру выдается **посадочный талон** (boarding_passes), в котором указано место в самолете. Пассажир может зарегистрироваться только на тот рейс, который есть у него в билете. Комбинация рейса и места в самолете должна быть уникальной, чтобы не допустить выдачу двух посадочных талонов на одно место.

Количество **мест** (seats) в самолете и их распределение по классам обслуживания зависит от конкретной модели **самолета** (aircrafts), выполняющего рейс. Предполагается, что каждая модель имеет только одну компоновку салона. Схема данных не контролирует, что места в посадочных талонах соответствуют имеющимся в салоне.

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

Бронирование

Намереваясь воспользоваться услугами нашей авиакомпании, пассажир заранее (`book_date`, максимум за месяц до рейса) бронирует необходимые билеты. Бронирование идентифицируется своим 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, удостоверяющего личность пассажира `passenger_id`, а также его фамилию и имя `passenger_name` и контактную информацию `contact_data`.

Заметим, что ни идентификатор пассажира, ни 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

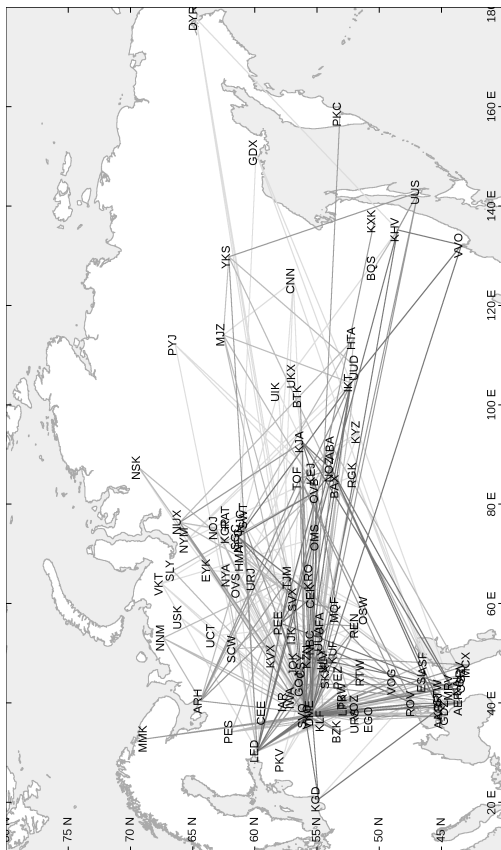
Перелет соединяет билет с рейсом и идентифицируется двумя их номерами.

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

Рейс

Естественный ключ таблицы рейсов состоит из двух полей – 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 из одного аэропорта до другого нет прямого рейса, в билет просто включаются несколько необходимых рейсов.

У каждого рейса есть запланированные дата и время вылета `scheduled_departure` и прибытия `scheduled_arrival`. Реальное время вылета `actual_departure` и прибытия `actual_arrival` могут отличаться: обычно не сильно, но иногда и на несколько часов, если рейс задержан.

Статус рейса `status` может принимать одно из следующих значений:

- **Scheduled**
Рейс доступен для бронирования. Это происходит за месяц до плановой даты вылета; до этого запись о рейсе не существует в базе данных.
- **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. Название можно использовать для того, чтобы определить аэропорты 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

Места определяют схему салона каждой модели. Каждое место определяется своим 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`, содержащее дополнительную 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 вылетел и находился в воздухе.

Позиция «среза» сохранена в функции `bookings.now`. Ей можно пользоваться в запросах там, где в обычной жизни 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 данных:

- 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).

Небольшая база годится для того, чтобы тренироваться писать запросы, и при этом не займет много места на диске. Если же вы хотите погрузиться в вопросы оптимизации, выберите большую базу, чтобы сразу понять, как ведут себя запросы на больших объемах данных.

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

Чтобы установить демонстрационную базу данных в Linux, скачайте один из файлов, предварительно переключившись на пользователя postgres. Например, для базы небольшого размера:

```
$ sudo su - postgres
```

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

Затем выполните команду:

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

В Windows любым веб-браузером скачайте с сайта файл `edu.postgrespro.ru/demo-small.zip`, дважды кликните на нем, чтобы открыть архив, и скопируйте файл `demo-small-20170815.sql` в каталог `C:\Program Files\PostgresPro10`.

Затем запустите psql (ярлык «SQL Shell (psql)») и выполните команду:

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

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

Примеры запросов

Пара слов о схеме

Теперь, когда установка выполнена, запустите `psql` и подключитесь к демонстрационной базе:

```
postgres=# \c demo
```

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

```
demo=#
```

Все интересующие нас объекты находятся в схеме `bookings`. При подключении к базе данных эта схема будет использоваться 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)
```

Однако для функции `bookings.now` все равно необходимо указывать схему, чтобы отличать ее от стандартной функции `now`:


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

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

Как вы заметили, названия самолетов выводятся по-русски. Так же обстоит дело и с названиями городов и аэропортов:

```
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)
```

Если вы предпочитаете английские названия, установите параметр `bookings.lang` в значение `en`. Это можно сделать, например, так:

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

```
ALTER DATABASE
```

```
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)
```

Как это устроено, вы можете разобраться, посмотрев определение `aircrafts` или `airports` с помощью команды `psql \d+`.

Подробнее про управление схемами:

postgrespro.ru/doc/ddl-schemas.html

и про установку конфигурационных параметров:

postgrespro.ru/doc/config-setting.html.

Простые запросы

Ниже мы покажем некоторые примеры задач на демонстрационной схеме. Большинство из них приведены вместе с решениями, остальные предлагается решить самостоятельно.

Задача. Кто летел позавчера рейсом Москва (SVO) – Новосибирск (OVB) на месте 1A, и когда он забронировал свой 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. Первый вариант использует конструкцию NOT EXISTS, чтобы определить места, на которые нет 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
       );
```

Второй вариант использует операцию вычитания множеств:

```
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;
```

Какой вариант использовать, во многом зависит от личных предпочтений. Необходимо только учитывать, что выполняться такие запросы будут по-разному, так что если важна производительность, то имеет смысл попробовать оба.

Задача. На каких маршрутах произошли самые длительные задержки рейсов? Выведите список из десяти «лидеров».

Решение. В запросе надо учитывать только рейсы, которые уже вылетели:

```
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`.

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 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 человек бывает включено в одно бронирование?

Решение. Сначала посчитаем количество человек в каждом бронировании, а затем число бронирований для каждого количества человек.

```
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 неделю назад.

Решение. Используем оконные функции, чтобы не обращаться к одним и тем же данным два раза.

Глядя в результаты приведенного ниже запроса, можно обратить внимание, что запас времени в некоторых случаях составляет несколько дней. Как правило, это билеты, взятые туда и обратно, то есть мы видим уже не время пересадки, а время нахождения в пункте назначения. Используя решение одной из задач в разделе «Массивы», можно учесть этот факт в запросе.

```

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);

```

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

Решение. Оконная функция используется для подсчета общего числа пассажиров.

```

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 имен.

```
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: не стоит объединять в одном текстовом поле несколько значений, если вы собираетесь работать с ними по отдельности; по-научному это называется «первой нормальной формой».

Массивы

Задача. В билете нет указания, в один ли он конец, или туда и обратно. Однако это можно вычислить, сравнив первый пункт отправления с последним пунктом назначения. Выведите для каждого билета аэропорты отправления и назначения без учета пересадок, и признак, взят ли билет 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-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.

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 с помощью операции `&&`:

```
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);
```

Рекурсивные запросы

Задача. Как с помощью минимального числа пересадок можно долететь из Усть-Кута (UKX) в Нерюнгри (CNN), и какое время 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;
```

Защивление предотвращается проверкой по массиву пересадок `hops`.

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

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

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

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.

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;

```

Задача. Найдите кратчайший путь 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.

```
WITH RECURSIVE p(
  last_arrival,
  destination,
  hops,
  flights,
  flight_time,
  min_time
) AS (
  SELECT a_from.airport_code,
         a_to.airport_code,
         array[a_from.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.hops || r.arrival_airport)::char(3)[],
         (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  NOT r.arrival_airport = ANY(p.hops)
  AND    p.flight_time + r.duration <
         coalesce(p.min_time, INTERVAL '1 year')
)
```

```

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. У нас имеются координаты аэропортов. Чтобы рассчитать расстояние, можно воспользоваться расширением earthdistance (и перевести 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.

Дополнительные ВОЗМОЖНОСТИ

Полнотекстовый поиск

Несмотря на мощь языка запросов SQL, его возможностей не всегда достаточно для эффективной работы с данными. Особенно это стало заметно в последнее время, когда лавины данных, обычно плохо структурированных, заполнили хранилища информации. Изрядная доля Больших Данных приходится на тексты, плохо поддающиеся разбиению на поля баз данных. Поиск документов на естественных языках, обычно с сортировкой результатов по релевантности поисковому запросу, называют полнотекстовым поиском. В самом простом и типичном случае запросом считается набор слов, а соответствие определяется частотой слов в документе. Примерно таким поиском мы занимаемся, набирая фразу в поисковике Google или Яндекс.

Существует большое количество поисковиков, платных и бесплатных, которые позволяют индексировать всю вашу коллекцию документов и организовать вполне качественный поиск. В этих случаях индекс — важнейший

инструмент и ускоритель поиска — не является частью базы данных. А это значит, что такие ценимые пользователями СУБД особенности, как синхронизация содержимого БД, транзакционность, доступ к метаданным и использование их для ограничения области поиска, организация безопасной политики доступа к документам и многое другое, оказываются недоступны.

Недостатки у все более популярных документо-ориентированных СУБД обычно в той же области: у них есть развитые средства полнотекстового поиска, но безопасность и заботы о синхронизации для них не приоритетны. 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. В результате производителям СУБД пришлось добавлять собственные расширения к стандарту SQL. У PostgreSQL это операторы сравнения ILIKE, ~, ~*, но и они не решают всех проблем, так как не умеют учитывать грамматические вариации слов, 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 контексте российский пользователь PostgreSQL получает немалые преимущества: расширения полнотекстового поиска для этой

СУБД созданы российскими разработчиками, поэтому возможность прямого контакта со специалистами или даже посещение их лекций поможет углубиться в технологические детали, если в этом возникнет потребность. Здесь же мы ограничимся простыми примерами.

Для изучения возможностей полнотекстового поиска создадим еще одну таблицу в демонстрационной базе данных. Пусть это будут наброски конспекта лекций преподавателя курсов, 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, то есть используя оператор LIKE:

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

Мы получим предсказуемый ответ: 0 строк. Ведь LIKE не знает, что в родительном падеже следует искать «баз данных» или «базу данных» в творительном. Запрос

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

выдаст строку из главы II (но не из главы I, где база в другом падеже):

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

В PostgreSQL есть оператор ILIKE, который позволяет не заботиться о регистрах, а то бы пришлось еще думать и о прописных и строчных буквах. Конечно, в распоряжении знатока SQL есть и регулярные выражения (шаблоны поиска), составление которых занятие увлекательное, сродни искусству. Но когда не до искусства, хочется иметь инструмент, который думал бы за тебя.

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

Для более продвинутого поиска нам хотелось бы включить в поисковую область и названия глав. Причем, дабы подчеркнуть их важность, мы наделим их весом при помощи функции `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 — В и D (из четырех возможных — А, В, С, 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 'база & данных' и другие грамматические вариации дадут тот же результат. Мы использовали оператор сравнения `@@` (две собаки), выполняющий работу, аналогичную `LIKE`. Синтаксис оператора не допускает выражение естественного языка с пробелами, такие как «база данных», поэтому слова соединяются логическим оператором «и».

Аргумент `russian` указывает на конфигурацию, которую использует СУБД. Она определяет подключаемые словари и парсер, разбивающий фразу на отдельные лексемы. Словари, несмотря на такое название, позволяют выполнять любые преобразования лексем. Например, простой словарь-стеммер типа `snowball`, используемый

по умолчанию, оставляет от слова только неизменяемую часть — именно поэтому поиск игнорирует окончания слов в запросе. Можно подключать и другие словари, такие, как `hunspell` (позволяет более точно учитывать морфологию слов), `unaccent` (превращает букву «ё» в «е») и другие.

Введенные нами веса позволяют вывести записи по результатам 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 повышает значимость найденной строки, помогает ранжировать результаты.

В заключительном эксперименте модифицируем задачу. 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 узнать в документации PostgreSQL: www.postgrespro.ru/doc/textsearch.html.

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: куда более простая структура данных (вначале это были прежде всего огромные таблицы с всего двумя колонками: ключ-значение) позволяла ускорить поиск на порядки. Они могли обрабатывать небывалые объемы информации и легко масштабировались, всю используя параллельные вычисления. В NoSQL-базах не было необходимости хранить информацию по строкам, а хранение по столбцам для многих задач позволяло еще больше ускорить и распараллелить вычисления.

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

Поскольку в реляционных СУБД изменение схемы данных связано с большими вычислительными издержками, оказался как никогда кстати новый тип данных — JSON. Изначально он предназначался для JS-программистов, в том числе для AJAX-приложений, отсюда JS в названии. Он как бы брал сложность добавляемых данных на себя, позволяя создавать линейные и иерархические структуры-объекты, добавление которых не требовало пересчета всей базы.

Тем, кто делал приложения, уже не было необходимости модифицировать схему базы данных. Синтаксис JSON похож на XML своим строгим соблюдением иерархии

данных. JSON достаточно гибок для того, чтобы работать с разнородной, иногда непредсказуемой структурой данных.

Допустим, в нашей демо-базе студентов появилась возможность ввести личные данные: запустили анкету, спросили преподавателей. В анкете не обязательно заполнять все пункты, а некоторые из них включают графу «другое» и «добавьте о себе данные по вашему усмотрению».

Если бы мы добавили в базу новые данные в привычной манере, то в многочисленных появившихся столбцах или дополнительных таблицах было бы большое количество пустых полей. Но еще хуже то, что в будущем могут появиться новые столбцы, а тогда придется существенно переделывать всю базу.

Мы решим эту проблему, используя тип `json` и появившийся позже `jsonb`, в котором данные хранятся в экономичном бинарном виде, и который, в отличие от `json`, приспособлен к созданию индексов, ускоряющих 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",
         "Гаша": "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 отсутствуют имена студентов:

```

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, где есть информация о достоинствах студентов. Мы можем обратиться к содер­жанию ключа «достоинство», используя специальный оператор ->>:

```

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 две записи имеют отношение к достоинствам Анны и Нины, однако такой ответ нас вряд ли удовлетворит, так как 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, 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. Такие данные можно плотно упаковать и поиск по ним работает быстрее. 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.

Для работы с jsonb набор операторов больше. Один из полезнейших операторов – оператор вхождения в объект @>. Он напоминает #> для json.

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, пожалуй, возможность создавать для `jsonb` индексы, поддерживающие оператор `@>`, обратный ему `<@` и многие другие. Среди имеющихся для `jsonb` индексов, как правило, лучше всего подходит `GIN`. Для `json` индексы не поддерживаются, поэтому для приложений с серьезной нагрузкой как правило лучше выбирать `jsonb`, а не `json`.

Подробнее о типах `json` и `jsonb` и о функциях для работы с ними можно узнать на страницах документации PostgreSQL [postgresql.org/docs/current/datatype-json.html](https://www.postgresql.org/docs/current/datatype-json.html) и [postgresql.org/docs/current/functions-json.html](https://www.postgresql.org/docs/current/functions-json.html).

PostgreSQL

для приложения

Отдельный пользователь

В предыдущей главе мы подключались к серверу баз данных под пользователем `postgres`, единственным существующим сразу после установки СУБД. Но `postgres` обладает правами суперпользователя, поэтому приложению не следует использовать его для подключения к базе данных. 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 привилегии:

postgrespro.ru/doc/user-manag.html
и postgrespro.ru/doc/ddl-priv.html.

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 подсказка изменилась: вместо «решетки» (#) теперь отображается символ «больше» (>) – решетка указывает на суперпользователя по аналогии с пользователем root в Unix.

Со своей базой данных пользователь app может работать без ограничений. Например, можно создать в ней 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, вы можете 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, **файл основных настроек** `postgresql.conf` (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 пытается подключиться к серверу, PostgreSQL находит в этом файле первую сверху строку, соответствующую параметрам соединения по четырем параметрам: тип соединения, имя базы данных, имя пользователя и IP-адрес клиента. В той же строке написано, как пользователь должен подтвердить, что он действительно тот, за кого себя выдает.

Например, в Debian и Ubuntu в этом файле есть, в числе прочих, такая строка:

```
local    all         all                                peer
```

Она говорит о том, что локальные соединения (`local`) к любой базе данных (`all`) под любым пользователем (`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 пользователю обычно не приходится вводить пароль при подключении к серверу на своем компьютере: достаточно того, что пароль был введен при входе в систему.

А вот Windows не поддерживает локальных соединений, и там строка выглядит следующим образом:

```
host    all    all    127.0.0.1/32    md5
```

Она означает, что сетевые соединения (`host`) к любой базе данных (`all`) под любым пользователем (`all`) с локального адреса (`127.0.0.1`) должны проверяться методом `md5`. Этот метод подразумевает ввод пользователем пароля.

Итак, для наших целей допишите в конец `pg_hba.conf` следующую строку:

```
host    appdb  app    all                md5
```

Это разрешит доступ к базе данных `appdb` пользователю `app` с любого адреса при указании верного пароля.

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.html

Проверка связи

Для того, чтобы обратиться к PostgreSQL из программы на каком-либо языке программирования, необходимо использовать подходящую библиотеку и установить драйвер СУБД.

Ниже приведены простые примеры для нескольких популярных языков, которые помогут быстро проверить соединение с базой данных. Приведенные программы намеренно содержат только минимально необходимый код для запроса к базе данных; в частности, не предусмотрена никакая обработка ошибок. Не стоит рассматривать эти фрагменты как пример для подражания.

Если вы используете ОС Windows, не забудьте в окне Command Prompt сменить шрифт на TrueType (например, «Lucida Console» или «Consolas») и выполнить команды:

```
C:\> chcp 1251
```

```
Active code page: 1251
```

```
C:\> set PGCLIENTENCODING=WIN1251
```

PHP

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

```
$ sudo apt-get install php5-cli php5-pgsql
```

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

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

Пример программы (`test.php`):

```
<?php
$conn = pg_connect('host=localhost port=5432 ' .
                  'dbname=appdb user=app ' .
                  'password=password') 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, которые перечислены на сайте www.perl.org/get.html. ActiveState Perl и Strawberry Perl уже включают необходимый для PostgreSQL драйвер.

Пример программы (test.pl):

```
use DBI;
my $conn = DBI->connect(
    'dbi:Pg:dbname=appdb;host=localhost;port=5432',
    'app', 'password') 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 Debian и Ubuntu язык Python версии 2 предустановлен, так что нужен только драйвер:

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

(Если вы используете Python 3, установите вместо этого пакет `python3-psycopg2`.)

Python для операционной системы Windows можно взять с сайта www.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:

```
$ python test.py
```

Привет, мир!

Java

В языке Java работа с базой данных организована через интерфейс JDBC. Устанавливаем JDK 1.7; дополнительно нам потребуется пакет с драйвером JDBC:

```
$ sudo apt-get install openjdk-7-jdk
```

```
$ sudo apt-get install libpostgresql-jdbc-java
```

JDK для Windows можно скачать с www.oracle.com/technetwork/java/javase/downloads. Драйвер JDBC доступен на сайте jdbc.postgresql.org (выберите версию, соответствующую 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", "pgssw0rd");
        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, указывая 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 `appdb` всего одна таблица, не помешает задуматься и о сохранности данных. Пока в вашем приложении немного данных, сделать резервную копию проще всего утилитой `pg_dump`:

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

Если вы посмотрите получившийся файл `appdb.dump` с помощью текстового редактора, то обнаружите в нем обычные команды SQL, создающие и заполняющие данными все объекты `appdb`. Этот файл можно подать на вход `psql`, чтобы восстановить содержимое базы данных. Например, можно создать новую БД и загрузить данные туда:

```
$ createdb appdb2
```

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

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 всего 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 можно получить утилитой `pg_basebackup`. Подробнее про доступные средства резервного копирования и восстановления смотрите в документации: postgrespro.ru/doc/backup.

Штатные средства PostgreSQL позволяют сделать практически все, что нужно, но требуют выполнения многочисленных шагов, нуждающихся в автоматизации. Поэтому многие компании создают собственные инструменты резервирования для упрощения этой задачи. Такой инструмент есть и у нашей компании PostgreSQL Professional. Он называется **pg_probackup**, распространяется свободно и позволяет выполнять инкрементальное резервное копирование на уровне страниц, контролировать целостность данных, работать с большими объемами информации за счет параллелизма и сжатия, реализовывать различные стратегии резервного

копирования. Полная документация доступна по адресу postgrespro.ru/doc/app-pgprobackup.

Что дальше?

Теперь вы готовы к разработке вашего 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. А клиентская — все, что работает вне СУБД и подключается к ней; с точки зрения базы данных не важно, будет ли 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. Особенно часто это происходит, когда команда разработчиков не знакома на детальном уровне с возможностями СУБД и предпочитает полагаться на то, что знает хорошо: на прикладной код. 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:

- **Поддержка согласованности данных возлагается на приложение.**

Вместо того, чтобы поручить СУБД следить за согласованностью данных (а это именно то, чем сильны реляционные системы), приложение самостоятельно выполняет необходимые проверки. 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. Но 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. При этом, во-первых, выполняется лишняя пересылка данных по сети, а во-вторых, алгоритмы работы с данными (сканирования, соединения, сортировки, агрегации) в СУБД отлаживаются и оптимизируются десятилетиями и справятся в задаче 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.

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.

Тем не менее, третья версия еще не окончательно забыта, знамя разработки подхватила команда BigSQL: www.openscg.com/bigsql/pgadmin3.

Мы же посмотрим на основные возможности нового pgAdmin 4.

Установка

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

Для систем Debian и Ubuntu, к сожалению, пока нет готового пакета, поэтому опишем процесс подробно. Сначала устанавливаем пакеты для языка Python:

```
$ sudo apt-get install virtualenv python-pip \
libpq-dev python-dev
```

Затем инициализируем виртуальную среду в каталоге pgadmin4 (конечно, можно выбрать и другой каталог):

```
$ cd ~
$ virtualenv pgadmin4
$ cd pgadmin4
$ source bin/activate
```

Теперь устанавливаем собственно программу pgAdmin. Последнюю актуальную версию можно посмотреть здесь: www.pgadmin.org/download/pgadmin-4-python-wheel/.

```
$ pip install https://ftp.postgresql.org/pub/pgadmin/
pgadmin4/v2.0/pip/pgadmin4-2.0-py2.py3-none-any.whl
$ rm -rf ~/.pgadmin/
```

Последнее, что осталось сделать – сконфигурировать pgAdmin для работы в автономном режиме (другой, серверный, режим мы не будем здесь рассматривать).

```
$ cat <<EOF \  
>lib/python2.7/site-packages/pgadmin4/config_local.py  
import os  
DATA_DIR = os.path.realpath(  
    os.path.expanduser(u'~/pgadmin/'))  
LOG_FILE = os.path.join(DATA_DIR, 'pgadmin4.log')  
SQLITE_PATH = os.path.join(DATA_DIR, 'pgadmin4.db')  
SESSION_DB_PATH = os.path.join(DATA_DIR, 'sessions')  
STORAGE_DIR = os.path.join(DATA_DIR, 'storage')  
SERVER_MODE = False  
EOF
```

К счастью, эти действия надо проделать один раз.

Для запуска pgAdmin 4 выполните:

```
$ cd ~/pgadmin4  
$ source bin/activate  
$ python \  
    lib/python2.7/site-packages/pgadmin4/pgAdmin4.py
```

Теперь пользовательский интерфейс (полностью переведенный на русский язык нашей компанией) доступен вам в веб-браузере по адресу localhost:5050.

Возможности

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, а на

вкладке **Соединение** (Connection) – **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).

The image shows a dialog box titled "Создание Сервер" (Create Server) with a close button (X) in the top right corner. The dialog has four tabs: "Общие" (General), "Соединение" (Connection), "SSL", and "Дополнительно" (Advanced). The "Соединение" tab is active. It contains the following fields and controls:

- Имя/адрес сервера** (Server name/address): text input field containing "localhost".
- Порт** (Port): text input field containing "5432".
- Служебная база данных** (Service database): text input field containing "postgres".
- Имя пользователя** (Username): text input field containing "postgres".
- Пароль** (Password): text input field with 10 black dots representing a masked password.
- Сохранить пароль?** (Save password?): a checked checkbox.
- Роль** (Role): empty text input field.

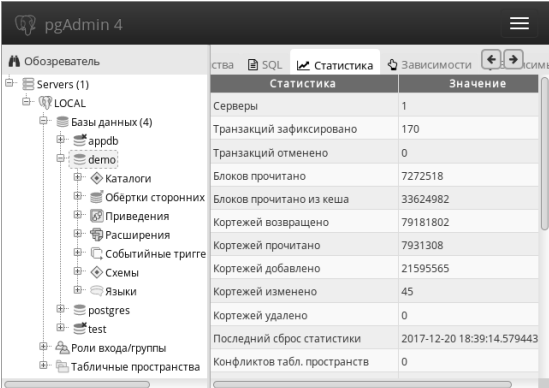
At the bottom of the dialog, there are three buttons: "Сохранить" (Save) with a floppy disk icon, "Отмена" (Cancel) with an X icon, and "Сбросить" (Reset) with a refresh icon. On the left side of the bottom bar, there are two small buttons: one with an information icon (i) and one with a question mark (?).

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, вы можете спуститься до сервера, который мы назвали LOCAL. Ниже будут перечислены имеющиеся в нем базы данных:

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



The screenshot shows the pgAdmin 4 interface. On the left is the 'Обозреватель' (Object Browser) pane, which is expanded to show the 'LOCAL' server. Under 'LOCAL', there are four databases: 'appdb', 'demo', 'postgres', and 'test'. The 'demo' database is selected, and its contents are listed: 'Каталоги', 'Обертки сторонних', 'Приведения', 'Расширения', 'Событийные триггеры', 'Схемы', and 'Языки'. Below these are 'Роли входа/группы' and 'Табличные пространства'. On the right, the 'Статистика' (Statistics) pane is active, displaying a table with two columns: 'Статистика' and 'Значение'.

Статистика	Значение
Серверы	1
Транзакций зафиксировано	170
Транзакций отменено	0
Блоков прочитано	7272518
Блоков прочитано из кеша	33624982
Кортежей возвращено	79181802
Кортежей прочитано	7931308
Кортежей добавлено	21595565
Кортежей изменено	45
Кортежей удалено	0
Последний сброс статистики	2017-12-20 18:39:14.579443
Конфликтов табл. пространств	0

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.

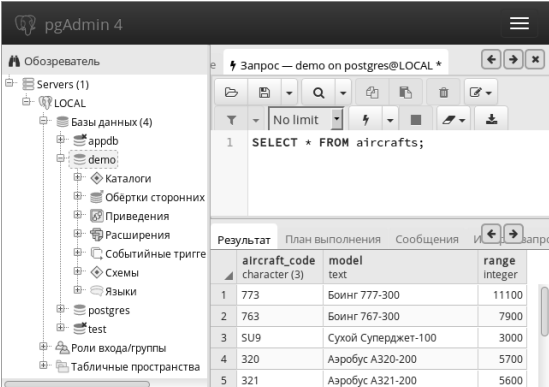
В правой части окна на отдельных вкладках выводится справочная информация:

- **Панель информации** (Dashboard) – показывает графики, отражающие активность системы;
- **Свойства** (Properties) – свойства выбранного объекта (например, 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** и **Зависимые** (Dependencies, Dependents) – 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.



The screenshot shows the pgAdmin 4 interface. On the left is the 'Обозреватель' (Browser) tree with 'demo' database selected. The main window is titled 'Запрос — demo on postgres@LOCAL *' and contains the SQL query: `1 SELECT * FROM aircrafts;`. Below the query editor, the 'Результат' (Results) tab is active, displaying a table with 5 rows and 3 columns: `aircraft_code` (character), `model` (text), and `range` (integer).

	aircraft_code	model	range
1	773	Боинг 777-300	11100
2	763	Боинг 767-300	7900
3	SU9	Сухой Суперджет-100	3000
4	320	Аэробус A320-200	5700
5	321	Аэробус A321-200	5600

Вы можете вводить следующий запрос на новой строке, не стирая предыдущий; просто выделите нужный фрагмент кода перед тем, как нажимать F5. Таким образом история ваших действий всегда будет у вас перед глазами – 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, таблиц системного каталога, административных функций и команд SQL. Особо отметим встроенный отладчик PL/pgSQL-кода. Со всеми возможностями этой программы вы можете познакомиться на сайте продукта www.pgadmin.org или в справочной системе самой программы.

Обучение и документация

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.

Мы в Postgres Professional перевели на русский язык весь комплект документации — он доступен на нашем сайте: www.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 и на нашем сайте, и по адресу www.postgresql.org/docs.

Учебные курсы

Помимо документации, мы занимаемся разработкой учебных курсов для администраторов баз данных и разработчиков приложений:

- DBA1. Администрирование PostgreSQL. Базовый курс.
- DBA2. Администрирование PostgreSQL. Расширенный курс.
- DEV1. Базовый курс для разработчиков серверной части приложения.
- DEV2. Расширенный курс для разработчиков серверной части приложения.

Деление курсов на базовые и расширенные вызвано большим объемом информации, который невозможно изложить и усвоить за несколько дней. Не стоит считать, что базовый курс предназначен только для новичков, а расширенный — только для опытных администраторов или разработчиков. Хотя между курсами есть пересечения по темам, но их не очень много.

Например, базовый трехдневный курс DBA1 знакомит с PostgreSQL и подробно объясняет базовые понятия администрирования, а пятидневный DBA2 охватывает подробности внутреннего устройства СУБД и ее настройки, оптимизацию запросов и ряд других тем. Расширенный курс предполагает, что к темам базового курса возвращаться не нужно. Те же принципы лежат в основе деления на две части курса для разработчиков.

Документация PostgreSQL содержит полные детальные сведения, которые, однако, разбросаны по разным главам и требуют многократного внимательного прочтения. В отличие от документации, каждый курс состоит из модулей, каждый из которых в свою очередь содержит связанный набор тем, последовательно раскрывающих его содержание. Акцент делается не на полноте охвата, а на выделении важной и практически полезной информации. Таким образом, курсы призваны не заменить документацию, а дополнить ее.

Каждая тема курса состоит из теоретической части и практики. Теория – это не только презентация, но в большинстве случаев еще и демонстрация работы на «живой» системе. На практике слушателям предлагается выполнить ряд заданий для закрепления пройденного материала.

Материал поделен между темами таким образом, чтобы теоретическая часть не превышала часа, так как большее время значительно усложняет восприятие материала. Практика, как правило, не превышает 30 минут.

Слушатели курса получают презентации с подробными комментариями к каждому слайду, результат работы демонстрационных скриптов, решения практических заданий, и, в некоторых случаях, дополнительные справочные материалы.

Для некоммерческого использования материалы курсов доступны на нашем сайте всем желающим.

Курсы для администраторов

DBA1. Администрирование PostgreSQL. Базовый курс

Продолжительность: 3 дня

Предварительные знания:

Минимальные сведения 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 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 кэш 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 DBA1 (презентации, демонстрации, практические задания, видеозапись лекций) доступны для самостоятельного изучения по адресу www.postgrespro.ru/education/courses/DBA1.

DBA2. Администрирование PostgreSQL. Расширенный курс

Продолжительность: 5 дней

Предварительные знания:

Владение Unix.

Базовые знания об архитектуре, установке, настройке, обслуживании СУБД.

Какие навыки будут получены:

Понимание архитектуры PostgreSQL.

Мониторинг и настройка базы, решение задач оптимизации производительности.

Выполнение задач сопровождения.

Резервирование и репликация.

Темы:

Введение

1. Архитектура PostgreSQL

Изоляция и многоверсионность

2. Изоляция транзакций
3. Страницы и версии строк
4. Снимки и блокировки
5. Очистка
6. Автоочистка и заморозка

Журналирование

7. Буферный кэш
8. Упреждающий журнал
9. Контрольная точка

Репликация

10. Файловая репликация
11. Поточковая репликация
12. Переключение на реплику
13. Репликация: варианты

Основы оптимизации

14. Обработка запроса
15. Методы доступа
16. Способы соединения
17. Статистика
18. Использование памяти
19. Профилирование
20. Оптимизация запросов

Разные темы

21. Секционирование
22. Локализация
23. Обновление сервера
24. Управление расширениями
25. Внешние данные

Материалы учебного курса DBA2 (презентации, демонстрации, практические задания, видеозапись лекций) доступны для самостоятельного изучения по адресу: www.postgrespro.ru/education/courses/DBA2.

Курсы для прикладных разработчиков

DEV1. Разработка серверной части приложений PostgreSQL. Базовый курс

Продолжительность: 4 дня

Предварительные знания:

Основы SQL.

Опыт работы с каким-нибудь процедурным языком программирования.

Минимальные сведения о работе в Unix.

Какие навыки будут получены:

Общие сведения об архитектуре PostgreSQL.

Использование основных объектов БД:

таблиц, индексов, представлений.

Программирование на стороне сервера на языках SQL и PL/pgSQL.

Использование основных типов данных, включая записи и массивы.

Организация взаимодействия с клиентской частью приложения.

Темы:

Базовый инструментарий

1. Установка и управление, psql

Архитектура

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

Организация данных

5. Логическая структура
6. Физическая структура

Приложение «Книжный магазин»

7. Схема данных приложения
8. Взаимодействие клиента с СУБД

SQL

9. Функции
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

Материалы учебного курса DEV1 (презентации, демонстрации, практические задания, видеозапись лекций) доступны для самостоятельного изучения по адресу: www.postgrespro.ru/education/courses/DEV1.

DEV2. Разработка серверной части приложений PostgreSQL. Расширенный курс

Этот курс мы готовим в настоящее время, он должен появиться в ближайшем будущем.

Где пройти обучение

Если вы хотите пройти обучение по перечисленным курсам в специализированном учебном центре, под руководством опытного преподавателя и с получением сертификата, то есть и такая возможность. Мы авторизовали несколько известных учебных центров, которые читают наши курсы. Их список можно посмотреть здесь: www.postgrespro.ru/education/where.

Курсы для разработчиков СУБД

Помимо курсов, которые читаются в учебных центрах на постоянной основе, разработчики ядра PostgreSQL из нашей компании тоже время от времени проводят обучение.

Hacking PostgreSQL

Курс «Hacking PostgreSQL» собран из личного опыта разработчиков, материалов конференций, статей и вдум-

чивого чтения документации и исходных кодов. В первую очередь он адресован начинающим разработчикам ядра PostgreSQL, но будет интересен и администраторам, которым иногда приходится обращаться к коду, и просто всем неравнодушным к архитектуре большой системы и желающим узнать, «как это работает на самом деле?».

Предварительные знания:

Знания основ языка SQL, функционала транзакций, индексов и т. п.

Знание языка C в объеме, достаточном как минимум для чтения исходных кодов (лучше иметь практические навыки).

Знакомство с базовыми структурами и алгоритмами.

Темы:

1. Обзор архитектуры
2. Сообщество PostgreSQL и инструменты разработчика
3. Расширяемость
4. Обзор исходного кода
5. Физическое представление данных
6. Разделяемая память и блокировки
7. Локальная память процессов
8. Базовое устройство планировщика и исполнителя запросов

Материалы курса Hacking PostgreSQL доступны для самостоятельного изучения по адресу www.postgrespro.ru/education/courses/hacking.

Путеводитель по галактике

Новости и обсуждения

Если вы собираетесь работать с PostgreSQL, вам захочется быть в курсе событий, узнавать о новых возможностях предстоящего выпуска, знакомиться с другими новостями. Много людей ведут свои блоги, публикуя интересные и полезные материалы. Удобный способ получить все англоязычные заметки в одном месте — читать сайт `planet.postgresql.org`.

Не забывайте и про `wiki.postgresql.org` — сборник статей, поддерживаемый и развиваемый сообществом. Здесь вы найдете ответы на часто задаваемые вопросы, обучающие материалы, статьи про настройку и оптимизацию, про особенности миграции с разных СУБД и многое другое. Часть материалов этого сайта доступна и на русском языке: `wiki.postgresql.org/wiki/Russian`. Вы тоже можете помочь сообществу, переведя заинтересовавшую вас англоязычную статью.

Более двух тысяч русскоязычных пользователей PostgreSQL входят в группу «PostgreSQL в России» на фейс-

буке: www.facebook.com/groups/postgresql.

Свой вопрос можно задать и на профильных сайтах. Например, на stackoverflow.com на английском языке или ru.stackoverflow.com на русском (не забудьте поставить метку «postgresql»), или на форуме www.sql.ru/forum/postgresql.

Новости нашей компании вы найдете по адресу postgrespro.ru/blog.

Списки рассылки

Если вы хотите узнавать обо всем первым, не дожидаясь, пока кто-нибудь напишет заметку в блоге, читайте списки рассылки. Разработчики PostgreSQL по старой традиции обсуждают между собой все вопросы исключительно по электронной почте. Для этого используется список рассылки `pgsql-hackers` (часто называемый просто «hackers»).

Полный перечень всех списков рассылки находится по адресу www.postgresql.org/list. Среди них:

- `pgsql-general` для обсуждения общих вопросов,
- `pgsql-bugs` для сообщений о найденных ошибках,
- `pgsql-announce` для новостей о выходе новых версий продуктов

и многие другие.

На любой список может подписаться каждый желающий, чтобы регулярно получать сообщения по электронной почте и при необходимости принять участие в дискуссии.

Другой вариант – время от времени читать архив сообщений на www.postgresql.org/list, или, в несколько более удобном виде, на www.postgresql-archive.org.

Commitfest

Еще один способ быть в курсе событий, не тратя на это много времени – заглядывать на сайт commitfest.postgresql.org. В этой системе периодически открываются «окна», в которых разработчики должны зарегистрировать свои патчи. Например, окно 01.03.2017–31.03.2017 относилось к версии 10, а следующее за ним окно 01.09.2017–30.09.2017 – 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, потом либо принимаются, либо переносятся в следующее окно, либо – если совсем не повезло – отвергаются.

Таким образом вы можете быть в курсе возможностей, которые уже включены или предполагаются к включению в еще не вышедшую версию.

Конференции

В России ежегодно проводятся две крупные международные конференции, собирающие сотни разработчиков и пользователей PostgreSQL:

5–7 февраля 2018 – PGConf в Москве (pgconf.ru)

июль 2018 – PGDay в Санкт-Петербурге (pgday.ru)

Конференции по PostgreSQL проводятся и во всем мире:

май – PGCon в Оттаве (pgcon.org)

ноябрь – PGConf Europe (pgconf.eu)

Кроме того, в разных городах России проводятся конференции с более широкой тематикой, на которых представлено и направление баз данных и, в том числе, PostgreSQL:

март – CodeFest в Новосибирске (codefest.ru)

апрель – Dump в Екатеринбурге (dump-conf.ru)

апрель – SECON в Пензе (www.secon.ru)

апрель – Стачка в Ульяновске (nastachku.ru)

декабрь – HappyDev в Омске (happydev.ru)

Помимо конференций проходят и неофициальные регулярные встречи, в том числе онлайн: www.meetup.com/postgresqlrussia.

О компании

Компания Postgres Professional была основана в 2015 году и объединила всех ключевых российских разработчиков, вклад которых в развитие PostgreSQL существенен и признан мировым сообществом. Компания является российским вендором PostgreSQL и выполняет разработки на уровне ядра СУБД и расширений, оказывает услуги по проектированию и поддержке прикладных систем, миграции на PostgreSQL.

Компания уделяет большое внимание образовательной деятельности, организует крупнейшую международную конференцию PgConf.Russia в Москве и принимает участие в конференциях по всему миру.

Наш адрес:

117036, г. Москва, ул. Дмитрия Ульянова, д. 7А

Телефон:

+7 495 150-06-91

Сайт компании и электронная почта:

postgrespro.ru

info@postgrespro.ru

Услуги

Промышленные решения на основе PostgreSQL

- Проектирование и участие в создании критичных высоконагруженных систем с использованием СУБД PostgreSQL.
- Оптимизация конфигурации СУБД.
- Консультирование по вопросам использованию СУБД в промышленных системах.
- Аудит систем заказчика по вопросам использования СУБД, проектирования баз данных, высокопроизводительных и отказоустойчивых архитектур.
- Внедрение СУБД PostgreSQL.

Вендорская техническая поддержка

- Вторая и третья линии техподдержки СУБД PostgreSQL в режиме 24x7.
- Круглосуточная поддержка опытными администраторами: мониторинг, восстановление работоспособности, анализ непредвиденных обстоятельств, обеспечение производительности.
- Исправление ошибок, обнаруженных в СУБД и ее расширениях.

Миграция прикладных систем

- Анализ имеющихся прикладных систем и определение сложности их миграции с других СУБД на PostgreSQL.
- Определение архитектуры нового решения и требований к доработкам прикладных систем.
- Миграция систем на СУБД PostgreSQL, в том числе действующих систем под нагрузкой.
- Поддержка прикладных разработчиков в процессе миграции.

Разработка на уровне ядра и расширений

- Заказные разработки на уровне ядра СУБД и ее модулей расширения.
- Создание специальных модулей расширения для решения прикладных и системных задач заказчика.
- Создание кастомизированных версий СУБД в интересах заказчика.
- Публикация изменений в основную версию кода СУБД.

Организация обучения

- Обучение администраторов баз данных работе с СУБД PostgreSQL.
- Обучение разработчиков и архитекторов прикладных систем особенностям СУБД PostgreSQL и эффективному использованию ее достоинств.
- Информирование о новой функциональности и важных изменениях в новых версиях.
- Проведение семинаров по разбору проектов заказчиков.