

Приложение «Книжный магазин» Взаимодействие клиента с СУБД



Авторские права

© Postgres Professional, 2017 год.

Авторы: Егор Рогов, Павел Лузанов

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

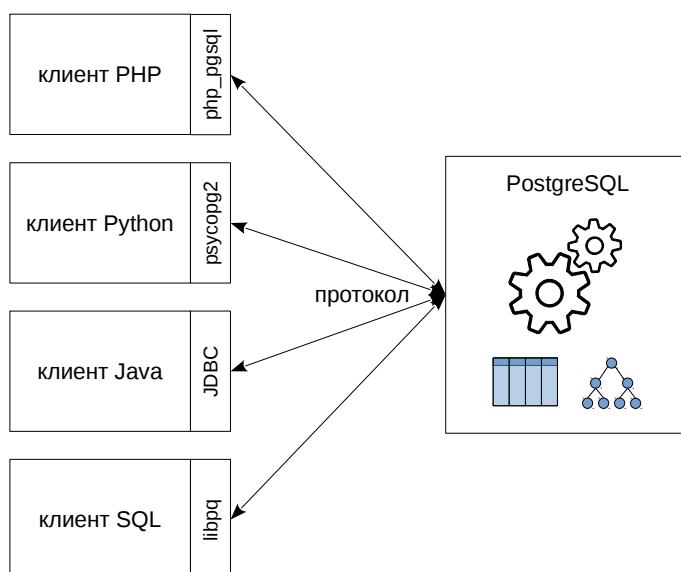
Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Общий протокол для взаимодействия с СУБД

Управление транзакциями

Способы выполнения запросов

Организация интерфейса

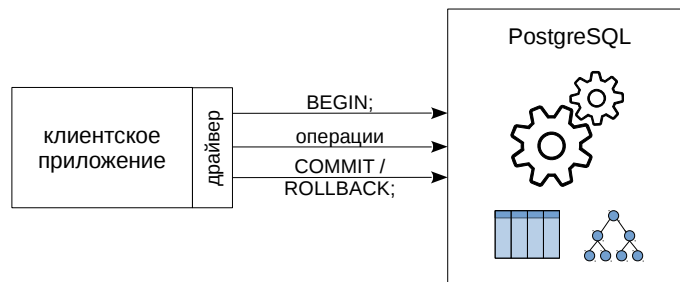


Как мы уже говорили, клиентское приложение может быть написано на любом языке программирования, лишь бы для этого языка был написан драйвер, понимающий протокол взаимодействия с сервером PostgreSQL. Часто такой драйвер — просто обертка вокруг библиотеки `libpq`, которой пользуются клиентские утилиты самого PostgreSQL.

Говоря про разработку *серверной* части приложения, невозможно совсем обойти вниманием *клиентскую* часть, хотя бы потому, что нам надо будет как-то проверять результаты нашей работы. Но про какой язык программирования говорить?

Мы выйдем из положения следующим образом. С одной стороны, у нас есть уже готовое приложение, которое было показано в предыдущей теме. Не важно, с помощью какой технологии оно написано: в нем нас интересует исключительно пользовательский интерфейс. (Разумеется, вы можете самостоятельно посмотреть исходные коды в виртуальной машине или по адресу <https://git.postgrespro.ru/pub/dev1app>, но это не является темой курса.)

С другой стороны, в качестве клиентского языка мы будем использовать SQL с помощью клиента `psql`. Реализация SQL в PostgreSQL включает нестандартные команды, которые допускают такое использование. Конечно, в реальной жизни вряд ли кто-то будет писать клиентскую часть на SQL, но для учебных целей это удобно. Мы рассчитываем, что сопоставить эти команды с аналогичными возможностями вашего любимого языка программирования не составит для вас большого труда.



BEGIN

SAVEPOINT

ROLLBACK TO SAVEPOINT

COMMIT

: AUTOCOMMIT

ROLLBACK

За управление транзакциями в PostgreSQL отвечает клиентское приложение. Внутри функций управление транзакциями запрещено (исключение составляют *автономные* транзакции, но они не являются штатной возможностью PostgreSQL и здесь не рассматриваются).

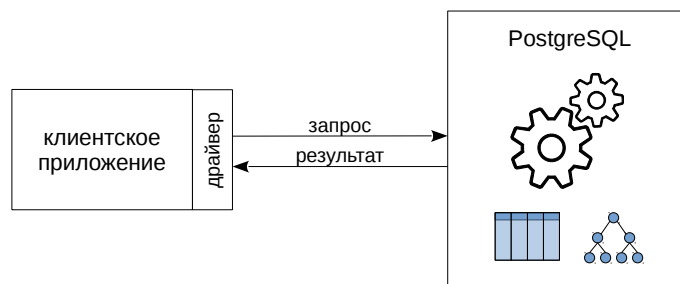
Основные команды (речь идет о SQL) для управления транзакциями:

- BEGIN начинает транзакцию.
- COMMIT завершает транзакцию и фиксирует изменения.
В psql по умолчанию действует режим автофиксации: если выполнить оператор, не начав транзакцию явно, то результат сразу же фиксируется. Режим устанавливается переменной:
`\set AUTOCOMMIT on/off`
Аналогичный режим как правило поддерживают и драйверы.
- ROLLBACK отменяет транзакцию.

Кроме того, можно выполнять частичный откат изменений, не прерывая при этом транзакцию. Для этого сначала устанавливается точка сохранения (SAVEPOINT), а затем при необходимости выполняется откат к этой точке (ROLLBACK TO SAVEPOINT).

Откат к точке сохранения не подразумевает передачу управления (то есть не работает как GOTO); отменяются только изменения состояния БД, выполненные от момента установки точки до текущего момента.

Выполнение запроса

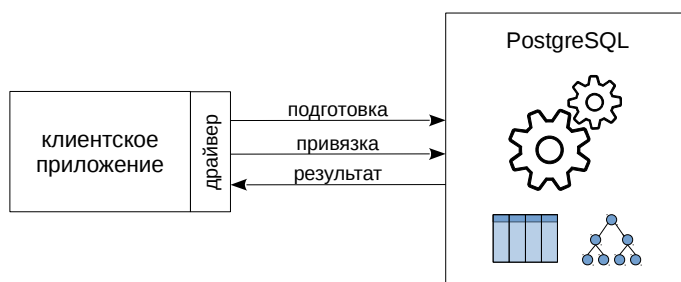


SELECT
INSERT
UPDATE
DELETE
TRUNCATE

Обычные команды (такие, как SELECT, INSERT и т. п.) выполняются сервером сразу и полностью. После того, как запрос разобран и спланирован, он выполняется и результаты накапливаются в памяти сервера. Затем результаты пересылаются клиенту.

Это удобный и простой способ, однако при большом объеме выборки он может оказаться ресурсоемким.

Тогда у приложения есть два пути. Можно либо ограничивать размер выборки (устанавливать предложение WHERE), либо воспользоваться курсорами, о которых речь пойдет чуть дальше.



PREPARE
EXECUTE

Кроме обычного выполнения запросов протокол PostgreSQL предусматривает расширенный режим, который позволяет более детально управлять выполнением операторов.

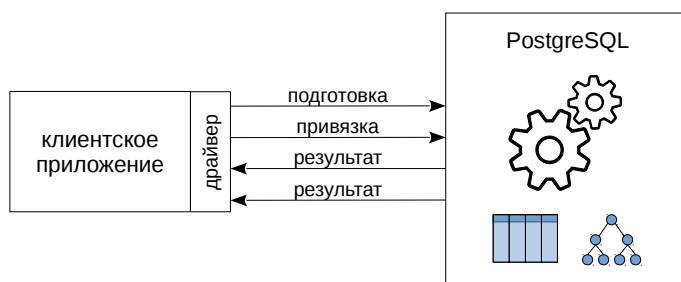
Одна из возможностей расширенного режима — подготовка операторов (SQL-команда PREPARE). При этом выполняется разбор, трансформация и запоминается полученное дерево разбора. Если у запроса нет параметров, то запоминается и построенный план выполнения.

Если же параметры есть, то их фактические значения указываются при выполнении запроса (SQL-команда EXECUTE) и принимаются во внимание при планировании. Если PostgreSQL сочтет, что план, построенный без учета параметров, будет не хуже, то он запомнит его и не будет повторно выполнять планирование.

Напомним, что разобранный запрос сохраняется в локальной памяти обслуживающего процесса; глобального кэша запросов нет. Это, в частности, создает определенные сложности при использовании пула соединений даже на уровне сеансов.

Преимущества подготовленных операторов:

- не повторять каждый раз разбор запроса, если он выполняется часто;
- обезопаситься от внедрения SQL-кода.



DECLARE
FETCH
MOVE
CLOSE

Другая возможность расширенного режима — курсоры. Протокол позволяет открыть курсор для какого-либо оператора (SQL-команда DECLARE), а затем получать результирующие данные постепенно (SQL-команда FETCH).

Курсор можно рассматривать как окно, в которое видно только часть из множества результатов. Размер окна по умолчанию равен одной строке, но настраивается. FETCH сдвигает окно (как правило, вниз, к концу выборки) и возвращает видимое содержимое; MOVE только сдвигает окно.

Открытый курсор представлен на сервере так называемым *порталом* — это место в локальной памяти обслуживающего процесса, в котором хранится текущее состояние курсора. Слово *портал* встречается в документации и может сбивать с толка; просто считайте курсор и портал синонимами. Кроме того, запрос, используемый в курсоре, подготавливается (то есть сохраняется его дерево разбора и, возможно, план выполнения).

По окончании работы открытый курсор закрывается, освобождая ресурсы (SQL-команда CLOSE).

Курсоры позволяют работать с реляционными данными (множествами) итеративно, строка за строкой. Иногда это удобно, но надо иметь в виду накладные расходы на обращение к серверу за каждой новой порцией данных.

Таблицы и триггеры

- чтение данных напрямую из таблицы (представления);
- запись данных напрямую в таблицу (представление),
- плюс триггеры для изменения связанных таблиц

- приложение должно быть в курсе модели данных,
- максимальная гибкость

- сложно поддерживать согласованность

Функции

- чтение данных из табличных функций;
- запись данных через вызов функций

- приложение отделено от модели данных и ограничено API

- большой объем работы по изготовлению функций-оберток,
- потенциальные проблемы с производительностью

Есть несколько способов организации интерфейса между клиентской и серверной частями приложения.

Один вариант — разрешить приложению напрямую обращаться к таблицам в базе данных и изменять их. При этом от приложения требуется детальное «знание» модели данных. Отчасти это требование можно ослабить за счет использования представлений (view).

Кроме того, от приложения требуется и определенная дисциплина — иначе очень сложно поддержать согласованность данных, защищаясь на уровне БД от любых возможных некорректных действий приложения. Но в этом случае достигается максимальная гибкость.

Другой вариант — запретить приложению доступ к таблицам и разрешить только вызовы функций. Чтение данных можно организовать с помощью табличных функций (которые возвращают набор строк). Изменение данных можно выполнять, вызывая другие функции и передавая им необходимые данные. В этом случае внутри функций можно реализовать все необходимые проверки согласованности — база данных будет защищена, но приложение сможет пользоваться только предоставленным и ограниченным набором возможностей. Такой вариант требует написания большого количества функций-оберток и может привести к потере производительности.

Вполне возможны и промежуточные варианты. Например, разрешить чтение данных непосредственно из таблиц, а изменение выполнять только через вызов функций.

Магазин

поиск книг: `get_catalog()`
покупка: `buy_book()`

Авторы

список авторов: `authors_v`
добавление автора: `add_author()`

Книги

список книг: `catalog_v`, список авторов: `authors_v`
добавление книги: `add_book()`

Каталог

список книг: `catalog_v`
заказ книги: `update_catalog_trigger`

В нашем приложении мы попробуем разные варианты организации интерфейса (хотя в реальной жизни обычно лучше систематически придерживаться какого-то одного подхода).

Для получения списка авторов мы создадим представление `authors_v`, для получения списка книг — представление `catalog_v` (в практике к этой теме).

Для поиска книг в Магазине будем использовать функцию `get_catalog` (в теме «SQL. Составные типы»); для покупки книг — функцию `buy_book` («PL/pgSQL. Выполнение запросов»).

Добавление автора будет выполнять функция `add_author` («PL/pgSQL. Выполнение запросов»), добавление книги — функция `add_book` («PL/pgSQL. Массивы»).

Для заказа книг сделаем представление `catalog_v` обновляемым («PL/pgSQL. Триггеры»).



Возможности взаимодействия определяются протоколом и полнотой его реализации в драйвере

Транзакциями управляет клиент

Запрос можно выполнять по-разному:

- полностью
- предварительно подготавливать
- итеративно с помощью курсора

Для клиент-серверного интерфейса можно использовать таблицы, представления, функции, триггеры



Здесь и далее все практические задания, связанные с приложением, выполняются в базе данных bookstore.

1. Создайте представление authors_v для авторов книг.
2. Создайте представление catalog_v для каталога книг.
3. Создайте представление operations_v для операций.
4. Проверьте, что приложение стало показывать данные на вкладках «Книги», «Авторы» и «Каталог».

1. Authors_v:

author_id integer
display_name text — *фамилия, имя и отчество*

Что будет выводить функция, если отчество будет не определено (NULL)?

Не будет ли функция выводить лишний пробел, если отчество пустое ("")?

2. Catalog_v:

book_id integer
display_name text — *название книги*

3. Operations_v:

book_id integer
op_type text — *тип операции (поступление или покупка)*
qty_change integer
date_created date

Влияние размера выборки на производительность.

1. Создайте произвольную таблицу с миллионом строк в ней.
2. Прочитайте всю таблицу с помощью курсора, на каждом шаге выбирая одну строку. Засеките время.
3. Повторите п. 2, увеличив размер выборки до 10 строк. Как изменилось время?
4. Повторите, увеличив размер выборки до 100 строк. Как изменилось время?
5. Повторите, увеличив размер выборки до 1000 строк. Как изменилось время?
6. Какие выводы можно сделать?

В SQL нет циклов, но вы можете сгенерировать файл с нужным числом команд FETCH (например, средствами командного интерпретатора bash или с помощью psql) и затем выполнить его.