Практическое задание 11

PostgreSQL

Модификация базы данных

Возможности программирования на стороне сервера

Функциональность сервера баз данных в PostgreSQL можно расширить с помощью реализации собственных функций, типов данных, триггеров и т. д. Это довольно сложные темы, для освоения которых рекомендуется предварительно изучить и понять всю имеющуюся по адресу postgrespro.ru/docs/postgresql/15/index документацию для пользователей PostgreSQL. В частности, могут быть использованы сторонние языки программирования на стороне сервера, которые поддерживаются дистрибутивом PostgreSQL, а также может быть реализовано почти полноценное программированием на стороне сервера за счет встроенного процедурного расширения SQL - языка PL/pgSQL.

PostgreSQL является расширяемым благодаря тому, что его работа управляется каталогами. Традиционные реляционные системы баз данных хранят информацию о базах, таблицах, столбцах и т. д., в структурах, которые обычно называются системными каталогами (иногда они называются словарями данных). Эти каталоги представляются пользователю в виде таблиц, подобных любым другим, но СУБД ведёт в них свои внутренние записи. Ключевое отличие хранит в этих каталогах намного больше информации: информацию не только о таблицах и столбцах, но также о типах данных, функциях, методах доступа и т. д. Эти таблицы могут быть изменены пользователями, а так как PostgreSQL в своих действиях руководствуется этими таблицами, это означает, что пользователи могут расширять PostgreSQL. Обычные же СУБД можно расширять, только модифицируя жёстко запрограммированные процедуры в исходном коде или загружая модули, специально разработанные производителем СУБД.

Кроме того, сервер PostgreSQL может динамически загружать в свой процесс код, написанный пользователем. То есть, пользователь может подключить файл с объектным кодом (например, разделяемую библиотеку), который реализует новый тип или функцию, а PostgreSQL загрузит его по мере надобности. Код, написанный на SQL, добавляется на сервер ещё проще. Эта способность менять своё поведение «на лету» делает PostgreSQL исключительно подходящим для быстрого прототипирования новых приложений и структур хранения.

Пользовательские функции и процедуры

В PostgreSQL представлены *функции* четырёх видов:

- функции на языке запросов (функции, написанные на SQL)
- функции на процедурных языках (функции, написанные, например, на PL/pgSQL или PL/Tcl)
- внутренние функции
- функции на языке C

Функции любых видов могут принимать в качестве аргументов (параметров) базовые типы, составные типы или их сочетания. Кроме того, любые функции могут возвращать значения базового или составного типа. Также можно определить функции, возвращающие наборы базовых или составных значений.

Функции многих видов могут также принимать или возвращать определённые псевдотипы (например, полиморфные типы), но доступные средства для работы с ними различаются.

Проще всего определить функции на языке SQL. Многие концепции, касающиеся функций на SQL, затем распространятся и на другие виды функций. При изучении этой темы будет полезно обращаться к странице справки по команде CREATE FUNCTION, чтобы лучше понимать примеры. Кроме того, некоторые примеры можно найти в файлах funcs.sql и funcs.c в каталоге src/tutorial исходного кода развернутой PostgreSQL.

Процедура — объект базы данных, подобный функции, но имеющий следующие отличия:

- Процедуры определяются командой CREATE PROCEDURE, а не CREATE

- Процедуры, в отличие от функций, не возвращают значение; поэтому в CREATE PROCEDURE отсутствует предложение RETURNS. Однако процедуры могут выдавать данные в вызывающий код через выходные параметры.
- Функции вызываются как часть запроса или команды DML, а процедуры вызываются отдельно командой CALL.

- Процедура, в отличие от функции, может фиксировать или откатывать транзакции во время её выполнения (а затем автоматически начинать новую транзакцию), если вызывающая команда CALL находится не в явном блоке транзакции.

Некоторые атрибуты функций (например, STRICT) неприменимы к процедурам. Эти атрибуты влияют на вызов функций в запросах и не имеют отношения к процедурам.

Функции и процедуры в совокупности также называются подпрограммами. Существуют команды, такие как ALTER ROUTINE и DROP ROUTINE, которые способны работать и с функциями, и с процедурами, не требуя указания точного вида объекта. Однако следует заметить, что команды CREATE ROUTINE нет.

Много полезной информации об использовании SQL функций размещено по адресу postgrespro.ru/docs/postgresql/15/xfunc-sql. Мы изложим в этом разделе только основные тезисы, позволяющие начать пользоваться этим инструментом.

SQL-функции выполняют произвольный список операторов SQL и возвращают результат последнего запроса в списке. В простом случае (не с множеством) будет возвращена первая строка результата последнего запроса. (Помните, что понятие «первая строка» в наборе результатов с несколькими строками определено точно, только если присутствует ORDER BY.) Если последний запрос вообще не вернёт строки, будет возвращено значение NULL.

Кроме того, можно объявить SQL-функцию как возвращающую множество (то есть, несколько строк), указав в качестве возвращаемого типа функции SETOF

некий_тип, либо объявив её с указанием RETURNS TABLE(столбцы). В этом случае будут возвращены все строки результата последнего запроса.

Тело SQL-функции должно представлять собой список SQL-операторов (команд), разделённых точкой с запятой. Точка с запятой после последнего оператора может отсутствовать. Если только функция не объявлена как возвращающая void, последним оператором должен быть SELECT, либо INSERT, UPDATE или DELETE с предложением RETURNING.

Помимо запросов SELECT, эти команды могут включать запросы, изменяющие данные (INSERT, UPDATE и DELETE и MERGE), а также другие SQL-команды. (В SQL-функциях нельзя использовать команды управления транзакциями, например COMMIT, SAVEPOINT, и некоторые вспомогательные команды, в частности VACUUM.) Однако, последней командой должна быть SELECT или команда с предложением RETURNING, возвращающая результат с типом возврата функции. Если же вы хотите определить функцию SQL, выполняющую действия, но не возвращающую полезное значение, вы можете объявить её как возвращающую тип void.

Если записать то же самое в виде процедуры, то можно не указывать возвращаемый тип. В таких простых случаях разница между процедурой и функцией, возвращающей void, в основном стилистическая. Однако по сравнению с функциями процедуры предоставляют дополнительную функциональность, например возможности управления транзакциями. Кроме того, процедуры соответствуют стандарту SQL, а возвращающие void функции специфичны для PostgreSQL.

Стоит обратить внимание на то, что прежде чем начинается выполнение команд, разбирается всё тело SQL-функции. Когда SQL-функция содержит команды, модифицирующие системные каталоги (например, CREATE TABLE), действие таких команд не будет проявляться на стадии анализа последующих команд этой функции. Так, например, команды CREATE TABLE foo (...); INSERT INTO foo VALUES(...); не будут работать, как ожидается, если их упаковать в одну SQL-функцию, так как foo не будет существовать к моменту разбору команды

INSERT. В подобных ситуациях вместо SQL-функции рекомендуется использовать PL/pgSQL.

Синтаксис команды CREATE FUNCTION требует, чтобы тело функции было записано как строковая константа. Обычно для этого удобнее всего заключать строковую константу в доллары. Если вы решите использовать обычный синтаксис с заключением строки в апострофы, вам придётся дублировать апострофы (') и обратную косую черту (\) (предполагается синтаксис спецпоследовательностей) в теле функции.

Процедурный язык PL/pgSQL

PostgreSQL позволяет разрабатывать собственные функции и на языках, отличных от SQL и C. Эти другие языки в целом обычно называются процедурными языками (PL, ProceduralLanguages). Процедурные языки не встроены в сервер PostgreSQL - они предлагаются загружаемыми модулями.

Если функция написана на процедурном языке, сервер баз данных сам по себе не знает, как интерпретировать её исходный текст. Вместо этого он передаёт эту задачу специальному обработчику, понимающему данный язык. Обработчик может либо выполнить всю работу по разбору, синтаксическому анализу, выполнению кода и т. д., либо действовать как «прослойка» между PostgreSQL и внешним исполнителем языка программирования. Сам обработчик представляет собой функцию на языке C, скомпилированную в виде разделяемого объекта и загружаемую по требованию, как и любая другая функция на C.

В настоящее время стандартный дистрибутив PostgreSQL включает четыре процедурных языка: PL/pgSQL, PL/Tcl, PL/Perl и PL/Python. В стандартной инсталляции PostgreSQL обработчик языка PL/pgSQL уже собран и установлен в каталог «библиотек»; более того, сам язык PL/pgSQL установлен во всех базах данных.

Целью проектирования PL/pgSQL было создание загружаемого процедурного языка, который:

- используется для создания функций, процедур и триггеров,
- добавляет управляющие структуры к языку SQL,
- может выполнять сложные вычисления,

- наследует все пользовательские типы, функции, процедуры и операторы, •может быть определён как доверенный язык,
- прост в использовании.

Функции PL/pgSQL могут использоваться везде, где допустимы встроенные функции. Например, можно создать функции со сложными вычислениями и условной логикой, а затем использовать их при определении операторов или в индексных выражениях.

PostgreSQL и большинство других СУБД используют SQL в качестве языка запросов. SQL хорошо переносим и прост в изучении. Однако каждый оператор SQL выполняется индивидуально на сервере базы данных. Это значит, что ваше клиентское приложение должно каждый запрос отправлять на сервер, ждать пока он будет обработан, получать результат, делать некоторые вычисления, затем отправлять последующие запросы на сервер. Всё это требует межпроцессного взаимодействия, а также несёт нагрузку на сеть, если клиент и сервер базы данных расположены на разных компьютерах.

PL/pgSQL позволяет сгруппировать блок вычислений и последовательность запросов внутри сервера базы данных, таким образом, мы получаем силу процедурного языка и простоту использования SQL при значительной экономии накладных расходов на клиент-серверное взаимодействие, т.е:

- Исключаются дополнительные обращения между клиентом и сервером
- Промежуточные ненужные результаты не передаются между сервером и клиентом
- Есть возможность избежать многочисленных разборов одного запроса

В результате это приводит к значительному увеличению производительности по сравнению с приложением, которое не использует хранимых функций. Кроме того, PL/pgSQL позволяет использовать все типы данных, операторы и функции SQL.

Функции на PL/pgSQL могут принимать в качестве аргументов все поддерживаемые сервером скалярные типы данных или массивы и возвращать в качестве результата любой из этих типов. Они могут принимать и возвращать любой именованный составной тип (тип кортежа). Также есть возможность объявить функцию на PL/pgSQL как принимающую record, то есть ей может быть

передан любой составной тип, или как возвращающую record, то есть её результатом будет кортеж, столбцы которого определит спецификация вызывающего запроса.

Функции, написанные на PL/pgSQL, определяются на сервере командами CREATE FUNCTION. Такая команда обычно выглядит, например, так:

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'тело функции'
LANGUAGE plpgsql;
```

Если рассматривать CREATE FUNCTION, тело функции представляет собой просто текстовую строку. Часто для написания тела функции удобнее заключать эту строку в доллары, а не в обычные апострофы. Если не применять заключение в доллары, все апострофы или обратные косые черты в теле функции придётся экранировать, дублируя их.

PL/pgSQL это блочно-структурированный язык. Текст тела функции должен быть блоком. Структура блока:

```
[ <<метка>> ]
[ DECLARE
объявления ]
BEGIN
операторы
END [ метка ];
```

Каждое объявление и каждый оператор в блоке должны завершаться символом «;» (точка с запятой). Блок, вложенный в другой блок, должен иметь точку с запятой после END, как показано выше. Однако финальный END, завершающий тело функции, не требует точки с запятой.

Часто пользователи добавляют точку с запятой сразу после BEGIN, однако это неправильно и ведёт к синтаксической ошибке.

Метка требуется только тогда, когда нужно идентифицировать блок в операторе EXIT, или дополнить имена переменных, объявленных в этом блоке. Если метка указана после END, то она должна совпадать с меткой в начале блока.

Ключевые слова не чувствительны к регистру символов. Как и в обычных SQL-командах, идентификаторы неявно преобразуются к нижнему регистру, если они не взяты в двойные кавычки.

Комментарии в PL/pgSQL коде работают так же, как и в обычном SQL. Двойное тире (--) начинает комментарий, который завершается в конце строки. Блочный комментарий начинается с /* и завершается */. Блочные комментарии могут быть вложенными.

Любой оператор в выполняемой секции блока может быть вложенным блоком. Вложенные блоки используются для логической группировки нескольких операторов или локализации области действия переменных для группы операторов. Во время выполнения вложенного блока переменные, объявленные в нём, скрывают переменные внешних блоков с такими же именами. Чтобы получить доступ к внешним переменным, нужно дополнить их имена меткой блока.

Важно не путать использование BEGIN/END для группировки операторов в PL/pgSQL с одноимёнными SQL-командами для управления транзакциями. BEGIN/END в PL/pgSQL служат только для группировки предложений; они не начинают и не заканчивают транзакции. Кроме того, блок с предложением EXCEPTION по сути создаёт вложенную транзакцию, которую можно отменить, не затрагивая внешнюю транзакцию.

Более подробную информацию по использованию языка PL/pgSQL вы можете получить, обратившись к документации по адресу

Обратите внимание, что весь используемый при выполнении лабораторной работы понятийный и операторный аппарат вам необходимо изложить в теоретической части вашего отчета. Например, там в обязательном порядке должна быть информация об используемых объявлениях, выражениях, управляющих структурах и курсорах.

Выполнение лабораторной работы без ознакомления с материалом по разделу процедурного языка PL/pgSQL из документации к СУБД с сайта

Процедура Customer_Insert

Галерее View Ridge требуется возможность добавлять в базу данных новых клиентов и записывать информацию о художниках, работами которых клиенты

интересуются. В частности, нужно записывать имя и телефоны клиента, а также ассоциировать его со всеми художниками выбранной национальности.

В листинге 1 показан скрипт для создания процедуры, выполняющей эту задачу. Процедура, которая называется Customer_Insert, принимает четыре параметра:

newname (имя нового клиента), newareacode (код региона), newphone (телефон) и artistnationality (национальность художника). Ключевое слово **IN** указывает на то, что все эти параметры являются входными. Выходные параметры (которых у этой процедуры нет) обозначаются ключевым словом **OUT**. **Листинг 1.** ПроцедураCustomerInsert

```
CREATE OR REPLACE procedure CustomerInsert (
newName IN char,
   newAreaCode IN char,
   newPhone IN char,
   artistNationality IN char
) AS $func$ DECLARE
     artistCursor CURSOR for
     SELECT ArtistID from ARTIST
     WHERE nationality = artistNationality;
      rowcountint;
BEGIN
     SELECT Count(*) INTO rowcount
     FROM CUSTOMER
WHERE name = newName AND area_code = newAreaCode AND phone_number =
newPhone;

      IF rowcount> 0 THEN
          BEGIN
        raisenotice 'Клиент уже есть в базе данных - никаких действий
не предпринято';
               RETURN;
          END;
     END IF;

     INSERT INTO CUSTOMER(CustomerID, Name, Area_Code,
Phone_Number)
VALUES(nextval('CustID'), newName, newAreaCode, newPhone);
     FOR artist IN artistCursor LOOP
   INSERT INTO CUSTOMER_ARTIST_INT(customerid , artistid)
   values(currval('CustID'), artist.artistid);
     END LOOP;
```

```
      raisenotice 'Новый клиент успешно добавлен в базу';
END;
$func$ languageplpgsql;
```

Раздел объявления переменных следует за ключевым словом DECLARE. Обратите внимание на новый элемент языка - *переменную-курсор* (cursorvariable) с именем artistcursor, определенный с помощью оператора SELECT.

Ознакомьтесь с разделом документации СУБД, посвященным ему: postgrespro.ru/docs/postgresql/15/plpgsql-cursors. Курсор выделяет из таблицы ARTIST для последующей обработки строки всех художников заданной национальности.

В первом разделе процедуры выполняется проверка, нет ли уже в базе информации о данном клиенте. В этом случае никакие действия не предпринимаются, а пользователю выводится соответствующее сообщение.

Вторая часть процедуры в листинге 1 вставляет данные о новом клиенте и затем перебирает всех художников выбранной национальности. Обратите внимание на использование управляющей структуры PL/pgSQL:

```
FOR artist IN artistcursor
```

Эта конструкция выполняет несколько задач. Прежде всего, она открывает курсор и считывает первую строку. Затем она последовательно обрабатывает все строки под курсором и по окончании обработки передает управление следующему оператору после FOR.

Заметьте также, что обращение к столбцу ArtistID текущей строки происходит с использованием синтаксиса artist.ArtistID, где artist — это имя *переменной цикла* FOR, а не курсора.

После того как процедура написана, ее необходимо скомпилировать и сохранить в базе данных прямо в psql. Также вы можете набрать текст процедуры в текстовом редакторе и сохранить в файле с расширением *.sql, после чего запустить из терминала, либо использовать возможности pgAdmin.

Если синтаксических ошибок не было, вы получите сообщение процедуру с помощью команды CALL:

```
call CustomerInsert('Имя', 'Код области', 'Телефон',
'Национальность');
```

Если возникнут ошибки на этапе выполнения процедуры, то терминальный клиент psql даст необходимую для поиска ошибки информацию.

Процедура NewCustomerWithTransaction

Приведем листинг 2 хранимой процедуры, реализующей процесс создания представления

Листинг2 ПроцедураNewCustomerWithTransaction

```
CREATE OR REPLACE PROCEDURE NewCustomerWithTransaction(
     IN newname character,
     IN newareacode character,
     IN newphone character,
     IN artistname character,
     IN worktitle character,
     IN workcopy character,
     IN price integer)
LANGUAGE 'plpgsql'
AS $BODY$
DECLARE  transCursor
   CURSOR for
   SELECT TransactionID, ARTIST.artistid
   FROM ARTIST, WORK, TRANSACTION
   WHERE Artist.Name = artistName AND Work.Title = workTitle
   AND Work.Copy = workCopy AND TRANSACTION.CustomerID IS NULL
   AND ARTIST.ArtistID = WORK.ArtistID AND WORK.WorkID =
   TRANSACTION.WorkID; rowcount int; tid int; aid int;
BEGIN
   SELECT Count(*) INTO rowcount
   FROM CUSTOMER
   WHERE name = newName AND area_code = newAreaCode AND
   phone_number = newPhone;
   /* Клиент уже есть в базе данных? */
   IF rowcount> 0 THEN BEGIN raisenotice 'Клиент уже есть в базе
   данных - никаких действий не предпринято';
```

```
RETURN;
END;
END IF;
/* Клиента нет в базе данных, добавляем нового клиента */
INSERT INTO CUSTOMER(CustomerID, Name, Area_Code, Phone_Number)
VALUES(nextval('CustID'), newName, newAreaCode, newPhone);
/* Ищем одну и только одну свободную строку в таблице
TRANSACTION. */ rowcount = 0;
FOR trans In transCursor
LOOP tid =
trans.TransactionID; aid
= trans.artistid;
rowcount = rowcount + 1;
END LOOP;
IF rowcount> 1 Then
BEGIN
/* Слишком много свободных строк -- выдаем сообщение об ошибке,
отменяем изменения и выходим из процедуры */ ROLLBACK;
raise notice 'Неверные данные в таблицах
ARTIST/WORK/TRANSACTION -- никаких действий не предпринято';
RETURN;
END;
END IF;
IF rowcount = 0 Then
BEGIN
/* Нет ни одной свободной строки - выдаем сообщение об ошибке,
отменяем изменения и выходим из процедуры */ ROLLBACK;
raisenotice 'Ни одной свободной строки в таблице TRANSACTION -
- никаких действий не предпринято';
RETURN;
END;
END IF;
/* Есть ровно одна строка -- используем ее. Идентификатор
транзакции, полученный из transcursor, находится в переменной tid
*/
raise notice '%', tid;
UPDATE TRANSACTION
SET customerid = currval('CustID'),
SalesPrice = price, PurchaseDate = current_date WHERE
TransactionID = tid; raise notice 'Клиентдобавленвбазуданных,
данныетранзакцийобновлены';
```

```
    /* Теперь регистрируем интерес данного клиента к данному
    художнику. Используем идентификатор художника, находящийся в
    переменной aid и текущее значение последовательности CurrVal */
    INSERT INTO CUSTOMER_ARTIST_INT(artistid , customerid)
    VALUES(aid, currval('CustID'));
END;
$BODY$;
```

Логикаэтойпроцедуры, носящейимяNewCustomerWithTransaction, такова. Сначала создаются данные нового клиента, и в таблице TRANSACTION, где регистрируются купленные произведения, ведется поиск строк, у которых столбец CustomerID имеет пустое значение. Этот поиск ведется по соединению таблиц ARTIST, WORK и TRANSACTION, так как имя художника (Name) хранится в таблице ARTIST, а название произведения (Title) и номер копии (Copy) хранятся в таблице WORK. Если найдена ровно одна такая строка, в ней обновляются столбцы CustomerID (идентификатор клиента), SalesPrice (цена продажи) и PurchaseDate (дата приобретения). Затем добавляется запись в таблицу пересечения, чтобы зарегистрировать интерес клиента к данному художнику. В противном случае, если число найденных строк больше или меньше 1, никаких изменений в базе данных не производится.

Параметры процедуры NewCustomerWithTransaction содержат информацию о клиенте и приобретенном произведении. В процедуре объявлены несколько переменных и курсор. Курсор определен на соединении таблиц ARTIST, WORK и TRANSACTION. Курсор выделяет столбцы TransactionID и ArtistID из строк, содержащих данные об искомом художнике и произведении и имеющих нулевое значение столбца CustomerID.

Прежде всего, процедура выполняет проверку, нет ли уже в базе данных информации о данном клиенте. Если нет, данные нового клиента добавляются в базу.

После того как данные о новом покупателе добавлены в базу, обрабатывается курсор TransCursor. Переменная rowcount используется для подсчета строк, в переменную tid записывается значение TransactionID, а в переменную aid — значение ArtistID.

В соответствии с логикой процедуры, если найдена одна и только одна строка, удовлетворяющая критериям, то в переменных tid и aid будут содержаться значения, нужные нам для успешного завершения транзакции. Если же таких строк не найдено или найдено более одной, то транзакция будет прервана, и ни tid, ни aid не будут использованы.

Для подсчета строк мы могли бы использовать выражение Count(*), и затем, если Count(*) = 1, запустить другой оператор, который бы получал нужные нам значения aid и tid. При той логике, которая реализована в процедуре сейчас, необходимость во втором SQL-операторе отпадает.

Если число строк RowCount больше единицы или равно нулю, то выдается сообщение об ошибке и выполняется откат транзакции, отменяющий вставку данных в таблицу CUSTOMER. Если RowCount равно 1, обновляется соответствующая строка в таблице TRANSACTION. Обратите внимание на использование функции CURRENT_DATE, с помощью которой вызывается и записывается текущая дата. Наконец, в таблицу пересечения добавляется строка с информацией о покупателе и авторе купленного произведения (aid).

Следующие два оператора отображают данные после обновления:

```
SELECT    CUSTOMER.Name, Copy, Title, ARTIST.Name
FROM        CUSTOMER, TRANSACTION, WORK, ARTIST
WHERE    CUSTOMER.CustomerID = TRANSACTION.CustomerID AND
TRANSACTION.WorkID = WORK.WorkID AND
WORK.ArtistID = ARTIST.ArtistID; и

SELECT CUSTOMER.Name, ARTIST.Name
FROM CUSTOMER, CUSTOMER_ARTIST_INT, ARTIST
WHERE     CUSTOMER.CustomerID=CUSTOMER_ARTIST_INT.CustomerID
AND ARTIST.ArtistID=CUSTOMER_ARTIST_INT.ArtistID;
```

Задания к лабораторной работе 5-6.

1. Модифицируйте процедуру 1, обеспечив добавление дополнительных данных о новых покупателях, с учетом их интереса к художникам определенных национальностей.
2. Добавьте 2 новых покупателей (через вызов процедуры 1)
3. Добавьте в таблицу CUSTOMER новое поле SecondName (Фамилия)
4. Модифицируйте процедуру 2 в соответствии с ранее сделанными изменениями в 1 процедуре.
5. Добавьте 2-х новых покупателей (через вызов процедуры 2)