Java для молодых специалистов Банка ВТБ

Урок 10

Работа с PostgreSQL

Язык SQL

Язык SQL — это непроцедурный язык, который является стандартным средством работы с данными во всех реляционных СУБД. Язык SQL — очень многообразный, он включает в себя целый ряд команд, которые, в свою очередь, иной раз имеют множество параметров и ключевых слов. Давайте разберем базовый набор возможностей языка SQL.

Для работы понадобиться установить PostgreSQL Server, и понадобится терминал psql. При запуске терминала в Windows могут возникнуть проблемы с кодировкой, для их исправления необходимо в настройках терминала выбрать шрифт Lucida Console и выполнить команду «\! chcp 1251».

Полезные команды. При работе в терминале можно пользоваться командами утилиты psql, они начинаются с символа «\». Для получения информации о таблице «\d имя_таблицы». Вывод списка схем «\dn».

Для создания таблиц в языке SQL служит команда **CREATE TABLE**. Упрощенный синтаксис:

```
CREATE TABLE имя-таблицы (
  имя-поля тип-данных [ограничения-целостности],
  имя-поля тип-данных [ограничения-целостности],
  ...
  имя-поля тип-данных [ограничения-целостности],
  [ограничение-целостности],
  [первичный-ключ],
  [внешний-ключ]
);
```

В квадратных скобках указаны необязательные элементы команды (в самой команде эти квадратные скобки указывать не надо). Завершается ввод команды символом «;». Например, для создания таблицы со студентами можно воспользоваться следующим запросом.

```
CREATE TABLE students (
  id serial NOT NULL,
  name text NOT NULL,
  group_name text NOT NULL,
  score integer NUT NULL,
  PRIMARY KEY (id)
);
```

При корректном вводе и выполнении запроса в консоли psql появится сообщение «CREATE TABLE». Поскольку таблицы, которые мы будем сейчас создавать, очень простые, то в случае выявления какого-либо упущения при их создании вы можете просто удалить таблицу и создать ее заново, с учетом необходимых исправлений. А команду ALTER TABLE, предназначенную для модифицирования структуры таблиц, мы рассмотрим немного позднее. Для удаления таблицы используется команда **DROP TABLE**.

```
DROP TABLE имя-таблицы;
```

Для добавления данных в таблицу служит команда **INSERT**. Упрощенный синтаксис:

© ВТБ совместно с GeekBrains

```
INSERT INTO имя-таблица [(имя-атрибута, имя-атрибута, …)] VALUES (значение-атрибута, значение-атрибута, …), (значение-атрибута, значение-атрибута, …);
```

В начале команды перечисляются атрибуты таблицы, при этом можно указывать их не в том порядке, в котором они были указаны при ее создании. Если не привести список атрибутов, то необходимо задавать значения атрибутов с учетом того порядка, в котором они следуют в определении таблицы, но в случае изменения структуры таблицы может понадобиться корректировать и команду INSERT.

Для выборки информации из таблиц служит команда **SELECT**. Упрощенный синтаксис:

```
SELECT имя-атрибута, имя-атрибута, ... FROM имя-таблицы;
```

Для вывода значений всех столбцов таблицы используется символ «*».

```
SELECT * FROM имя-таблицы;
```

Если сравнить порядок, в котором вы вводили строки в таблицу, с тем порядком, в котором строки выведены из нее по команде SELECT, то можно увидеть совпадение этих порядков. При выполнении простой выборки из таблицы СУБД не гарантирует никакого конкретного порядка вывода строк. Для упорядочивания выводимых строк можно воспользоваться **ORDER BY**. Для указания порядка сортировки добавляется **ASC** (по возрастанию) или **DESC** (по убыванию).

```
SELECT id, name, score
  FROM students
  ORDER BY score [ ASC, DESC ];
```

Далеко не всегда требуется выбирать все строки из таблицы. Множество выбираемых строк можно ограничить с помощью предложения WHERE команды SELECT.

```
SELECT id, name, score
  FROM students
  ORDER BY score
  WHERE условие;
```

Условие выбора строк может быть составным с использованием логических операций AND и OR. Если в условии мы работаем с численным значением, то можно формировать условия с помощью операторов сравнения =, <>, >, > =, < =.

```
SELECT * FROM students WHERE id > 2;
```

При работе со строками может возникнуть необходимость найти все строки начинающиеся с определенного слова. Для этого применяется оператор поиска шаблонов LIKE:

© ВТБ совместно с GeekBrains

```
SELECT * FROM students WHERE group_name LIKE 'RTS%';
```

Символ «%» соответствует любой последовательности символов, то есть искомая строка должна начинаться с RTS. Если же требуется отыскать некоторую последовательность символов где-то внутри строки, то шаблон должен начинаться и завершаться символом «%».

Символ подчеркивания «_», соответствует в точности одному любому символу. Если мы звхотим найти всех студентов, имя которых начинается на A, и состоит из четырех букв, то можем использвать запись вида «A___».

```
SELECT * FROM students WHERE name LIKE 'A___';
```

Если есть необходимость найти все значения в заданных пределах, можно использовать **BETWEEN** (обе границы включительно).

```
SELECT * FROM students WHERE score BETWEEN 60 AND 75;
```

При выборке данных можно проводить вычисления и получать в результирующей таблице вычисляемые столбцы. Если мы захотим представить оценки 100-балльной системы в виде 10-балльной системы, то можем вычислить их «на ходу» и присвоить новому столбцу псевдоним с помощью ключевого слова **AS**.

```
SELECT name, score, round(score / 10, 0) AS new_score FROM students;
```

Для того чтобы оставить в выборке только неповторяющиеся значения, служит ключевое слово **DISTINCT**:

```
SELECT DISTINCT group_name FROM students ORDER BY 1;
```

Обратите внимание, что столбец, по значениям которого будут упорядочены строки, указан не с помощью его имени, а с помощью его порядкового номера в предложении SELECT.

Для ограничения число строк, включаемых в результирующую выборку, служит предложение **LIMIT**. Допустим мы хотим найти трех студентов с наивысшим баллом.

```
SELECT name, score, FROM students ORDER BY score DESC LIMIT 3;
```

Для того, чтобы пропустить N первых строк, прежде чем начать вывод можно воспользоваться ключевым словом **OFFSET**.

```
SELECT name, score FROM students ORDER BY score DESC LIMIT 3 OFFSET 3;
```

В дополнение к вычисляемым столбцам, когда выводимые значения получают путем вычислений, при выборке данных из таблиц можно использовать условные выражения, позволяющие вывести то или иное значение в зависимости от условий. Для этого подойдет конструкция

© ВТБ совместно с GeekBrains

```
CASE WHEN условие THEN выражение [ WHEN ... ] [ ELSE выражение ] END
```

Допустим мы хотим преобразовать балл студента в какую-то оценку вида «Плохо», «Хорошо», «Отлично». Воспользовавшись этой конструкцией в предложении SELECT и назначив новому столбцу имя с помощью ключевого слова AS, получим следующий запрос:

```
SELECT name, score, CASE WHEN score > 80 THEN 'Отлично' WHEN score BETWEEN 60
AND 79 THEN 'Хорошо' ELSE 'Плохо' END AS result FROM students;
```

Среди множества функций, имеющихся в PostgreSQL, важное место занимают агрегатные функции, давайте на них посмотрим.

Если мы хотим посчитать количество студентов, имеющих балл 80, то можно воспользоваться **count(*)**.

```
SELECT count(*) FROM students WHERE score = 80;
```

Если необходимо посчитать количество записей по каждому из возможных баллов, то можно дополнительно воспользоваться **GROUP BY**.

```
SELECT score, count(*) FROM students GROUP BY score;
```

Для расчета среднего значения по столбцу используется функция **avg**. Для получения максимального значения по столбцу используется функция **max**. Для получения минимального значения по столбцу используется функция **min**.

```
SELECT avg(score) FROM students;
SELECT max(score) FROM students;
SELECT min(score) FROM students;
```

При выполнении выборок можно с помощью условий, заданных в предложении WHERE, сузить множество выбираемых строк. Аналогичная возможность существует и при выполнении группировок: можно включить в результирующее множество не все строки, а лишь те, которые удовлетворяют некоторому условию. Это условие можно задать с помощью **HAVING**. Важно помнить, что предложение WHERE работает с отдельными строками еще до выполнения группировки с помощью GROUP BY, а HAVING после. В качестве примера приведем такой запрос: из списка студентов найти количество человек в каждой группе, и отобразить группы, в которых более 10 человек.

```
SELECT group_name, count(*) FROM students GROUP BY group_name HAVING count(*)
>= 10 ORDER BY count DESC;
```

В команде SELECT предусмотрены средства для выполнения операций с выборками, как с множествами, а именно: **UNION** для вычисления объединения множеств строк из двух выборок; **INTERSECT** для вычисления пересечения множеств строк из двух выборок; **EXCEPT** для вычисления разности множеств строк из двух выборок.

© ВТБ совместно с GeekBrains

При использовании UNION строка включается в итоговое множество (выборку), если она присутствует хотя бы в одном из них. Строки-дубликаты в результирующее множество не включаются. Для их включения нужно использовать UNION ALL. Аналогично для INTERSECT и EXCEPT. При выполнении этих операций можно соединять две и более таблицы.

Переходим теперь к способам изменения данных в таблице. Команда **UPDATE** предназначена для обновления данных в таблицах. Упрощенный синтаксис:

```
UPDATE имя-таблицы
SET имя-атрибута1 = значение-атрибута1, имя-атрибута2 = значение-атрибута2,
...
WHERE условие;
```

Условие, указываемое в команде, предназначено для ограничения диапазона обновляемых строк. При отсутствии условия будут обновлены все строки в таблице.

Команда **DELETE** используется для удаления строк из таблицы:

```
DELETE FROM имя-таблицы WHERE условие;
```

Для удаления всех строк из таблицы:

```
DELETE FROM имя-таблицы;
```

Или можно воспользоваться командой TRUNCATE.

```
TRUNCATE имя-таблицы;
```

Основные типы данных

PostgreSQL имеет разнообразный набор встроенных типов данных, кроме того, предоставляет возможность создавать и свои собственные типы данных, что является одной из отличительных черт PostgreSQL.

Числовые

К **числовым типам** данных можно отнести: целочисленные типы, числа фиксированной точности, типы данных с плавающей точкой, последовательные типы (serial).

Целочисленные включают в себя: smallint (2 байта), integer (4 байта), bigint (8 байт). Такие типы позволяют хранить только целочисленные данные. В PostgreSQL существуют псевдонимы для этих стандартизированных имен типов, а именно: int2, int4 и int8.

Числа фиксированной точности представлены двумя типами — numeric и decimal. Для задания значения этого типа используются два базовых понятия: масштаб (scale) и точность (precision). Масштаб показывает число значащих цифр, стоящих справа от десятичной точки (запятой). Точность указывает общее число цифр как до десятичной точки, так и после нее. Параметры этого типа данных указываются в круглых скобках после имени типа: numeric(точность, масштаб). Например, numeric(6, 2). Его главное достоинство — это обеспечение точных результатов

© ВТБ совместно с GeekBrains

при выполнении вычислений. Данный тип следует выбирать для хранения денежных сумм, а также в случаях, когда требуется гарантировать точность вычислений.

Представителями типов данных **с плавающей точкой** являются типы real и double precision. Тип данных real может представить числа в диапазоне, как минимум, от 1E−37 до 1E+37 с точностью не меньше 6 десятичных цифр. Тип double precision имеет диапазон значений примерно от 1E−307 до 1E+308 с точностью не меньше 15 десятичных цифр. Эти типы данных поддерживают и специальные значения Infinity, −Infinity и NaN.

PostgreSQL поддерживает также тип данных float, определенный в стандарте SQL. В объявлении типа может использоваться параметр: float(p). Если его значение лежит в диапазоне от 1 до 24, то это будет равносильно использованию типа real, а при диапазоне от 25 до 53, или отсутствии параметра – double precision.

Последним из числовых типов является тип **serial**. Однако он фактически реализован не как настоящий тип, а просто как удобная замена целой группы SQL-команд. Тип serial удобен в тех случаях, когда требуется в какой-либо столбец вставлять уникальные целые значения, например, значения суррогатного первичного ключа. Синтаксис для создания столбца типа serial таков:

```
CREATE TABLE имя-таблицы (имя-столбца serial);
```

Эта команда эквивалентна следующей группе команд:

```
CREATE SEQUENCE имя-таблицы_имя-столбца_seq;
CREATE TABLE имя-таблицы (имя-столбца integer NOT NULL DEFAULT
nextval('имя-таблицы_имя-столбца_seq' ) );
ALTER SEQUENCE имя-таблицы_имя-столбца_seq
OWNED BY имя-таблицы.имя-столбца;
```

Одним из видов объектов в базе данных являются так называемые последовательности. Это, по сути, генераторы уникальных целых чисел. Для работы с этими последовательностями-генераторами используются специальные функции. Одна из них — это функция nextval, которая как раз и получает очередное число из последовательности, имя которой указано в качестве параметра функции. В команде CREATE TABLE ключевое слово DEFAULT предписывает, чтобы СУБД использовала в качестве значения по умолчанию то значение, которое формирует функция nextval. Поэтому если в команде вставки строки в таблицу INSERT INTO не будет передано значение для поля типа serial, то СУБД обратится к услугам этой функции. В том случае, когда в таблице поле типа serial является суррогатным первичным ключом, тогда нет необходимости указывать явное значение для вставки в это поле.

Символьные (строковые)

Стандартные представители строковых типов — это типы character varying(n) и character(n), где n – максимальное число символов в строке. При работе с многобайтовыми кодировками символов, нужно учитывать, что речь идет о символах, а не о байтах. Если сохраняемая строка символов будет короче, чем указано в определении типа, то значение типа character будет дополнено пробелами до требуемой длины, а значение типа character varying будет сохранено так, как есть. Типы character varying(n) и character(n) имеют псевдонимы varchar(n) и char(n) соответственно, и как правило, используют именно эти псевдонимы.

PostgreSQL дополнительно предлагает еще один символьный тип — text. В столбец этого типа можно ввести сколь угодно большое значение, конечно, в пределах, установленных при компиляции исходных текстов СУБД. Документация рекомендует использовать типы text и varchar, поскольку такое

© ВТБ совместно с GeekBrains

отличительное свойство типа character, как дополнение значений пробелами, на практике почти не востребовано.

В PostgreSQL обычно используется тип text. Константы символьных типов в SQL-командах заключаются в одинарные кавычки.

```
SELECT 'PostgreSQL';
```

В том случае, когда в константе содержится символ одинарной кавычки или обратной косой черты, их необходимо удваивать. В том случае, когда таких символов в константе много, все выражение становится трудно воспринимать. На помощь может прийти использование удвоенного символа «$». Эти символы выполняют ту же роль, что и одинарные кавычки, когда в них заключается строковая константа. При использовании символов «$» в качестве ограничителей уже не нужно удваивать никакие символы, содержащиеся в самой константе: ни одинарные кавычки, ни символы обратной косой черты. Возможность использования символов доллара в роли ограничителей символьной константы не является частью стандарта SQL. Это расширение, предлагаемое PostgreSQL.

Дата/время

PostgreSQL поддерживает все типы данных, предусмотренные стандартом SQL для даты и времени. Даты обрабатываются в соответствии с григорианским календарем, причем это делается даже в тех случаях, когда дата относится к тому моменту времени, когда этот календарь в данной стране еще не был принят.

Для этих типов данных предусмотрены определенные форматы для ввода значений и для вывода. Причем эти форматы могут не совпадать. Важно помнить, что при вводе значений их нужно заключать в одинарные кавычки, как и текстовые строки. Начнем рассмотрение с типа date.

Рекомендуемый стандартом ISO 8601 формат ввода дат таков: «yyyy-mm-dd». PostgreSQL позволяет использовать и другие форматы для ввода, например, «Sep 15, 2018». При выводе значений PostgreSQL использует формат по умолчанию, если не предписан другой формат.

```
SELECT '2018-09-15'::date;
```

Чтобы «сказать» СУБД, что введенное значение является датой, а не простой символьной строкой, используется операция приведения типа. В PostgreSQL она оформляется с использованием двойного символа «двоеточие» и имени того типа, к которому мы приводим данное значение. Важно учесть, что при выполнении приведения типа производится проверка значения на соответствие формату целевого типа и множеству его допустимых значений.

В PostgreSQL предусмотрен целый ряд функций для работы с датами и временем. Например, для получения значения текущей даты служит функция current_date. Ее особенностью является то, что при ее вызове круглые скобки не используются.

```
SELECT current_date;
```

Если нам требуется вывести дату в другом формате, то для разового преобразования формата можно использовать функцию to_char, например:

```
SELECT to_char(current_date, 'dd-mm-yyyy');
```

© ВТБ совместно с GeekBrains

Для хранения времени суток служат два типа данных: time и time with time zone. Первый из них хранит только время суток, а второй — дополнительно — еще и часовой пояс. Однако документация на PostgreSQL не рекомендует использовать тип time with time zone, поскольку смещение (offset), соответствующее конкретному часовому поясу, может зависеть от даты перехода на летнее время и обратно, но в этом типе дата отсутствует. При вводе значений времени допустимы различные форматы, например:

```
SELECT '21:15'::time;
```

При выводе СУБД дополнит введенное значение, в котором присутствуют только часы и минуты, секундами. При указании заведомо недопустимое значение времени будет получено сообщение об ошибке. Для получения значения текущего времени служит функция **current_time**.

Текущее время выводится с высокой точностью и дополняется числовым значением, соответствующим локальному часовому поясу, который установлен в конфигурационном файле сервера PostgreSQL.

В результате объединения типов даты и времени получается интегральный тип — временная отметка. Этот тип существует в двух вариантах: с учетом часового пояса — timestamp with time zone, либо без учета часового пояса — timestamp. Для первого варианта существует сокращенное наименование — timestamptz, которое является расширением PostgreSQL. При вводе и выводе значений этого типа данных используются соответствующие форматы ввода и вывода даты и времени. Вот пример с учетом часового пояса:

```
SELECT timestamp with time zone '2016-09-21 22:25:35';
```

Обратите внимание, что хотя мы не указали явно значение часового пояса при вводе данных, при выводе это значение +03 было добавлено. А это пример без учета часового пояса:

```
SELECT timestamp '2016-09-21 22:25:35';
```

В рассмотренных примерах мы использовали синтаксис тип 'строка' для указания конкретного типа простой литеральной константы. Имя типа мы указывали не после преобразуемого литерала, а перед ним, например, timestamp '2016-09-21 22:25:35'. Строго говоря, это не является операцией приведения типа, хотя и похоже на нее. Для получения значения текущей временной отметки служит функция **current_timestamp**. Приведем пример ее использования. SELECT current_timestamp; Здесь в выводе присутствует и часовой пояс: +03. Оба типа — timestamp и timestamptz — занимают один и тот же объем 8 байтов, но значения типа timestamptz хранятся, будучи приведенными к нулевому часовому поясу (UTC), а перед выводом приводятся к часовому поясу пользователя. На практике при принятии решения о том, какой из этих двух типов — timestamp или timestamptz — использовать, необходимо учитывать, требуется ли значения, хранящиеся в таблице, приводить к местному часовому поясу или не требуется.

Последним типом является interval, который представляет продолжительность отрезка времени между двумя моментами времени. Значение этого типа можно получить при вычитании одной временной отметки из другой.

```
SELECT ('2016-09-16'::timestamp - '2017-10-15'::timestamp)::interval;
```

© ВТБ совместно с GeekBrains

Значения временных отметок можно усекать с той или иной точностью с помощью функции date_trunc. Например, для получения текущей временной отметки с точностью до одного часа нужно сделать так:

```
SELECT (date_trunc('hour', current_timestamp));
```

Из значений временных отметок можно с помощью функции extract извлекать отдельные поля, т. е. год, месяц, день, число часов, минут или секунд и т. д. Например, вот запрос для извлечения номера месяца:

```
SELECT extract('mon' FROM timestamp '2017-10-25 12:34:56.123459');
```

Логический тип

Логический (boolean) тип может принимать три состояния: истина и ложь, а также неопределенное состояние, которое можно представить значением NULL. Таким образом, тип boolean реализует трехзначную логику. В качестве истинного состояния могут служить следующие значения: TRUE, 't', 'true', 'y', 'yes', 'on', '1'. В качестве ложного состояния могут служить следующие значения: FALSE, 'f', 'false', 'n', 'no', 'off', '0'.

Массивы

PostgreSQL позволяет создавать в таблицах такие столбцы, в которых будут содержаться не скалярные значения, а массивы переменной длины. Эти массивы могут быть многомерными и могут содержать значения любого из встроенных типов, а также типов данных, определенных пользователем. (два тире -- в примере ниже означают начало строкового комментария).

```
CREATE TABLE users (id serial, name text, arr integer[]);
INSERT INTO users (name, arr) VALUES ('Bob', '{1, 2, 3, 4, 5, 6, 7, 8, 9,
10}'::integer[]); -- 1
UPDATE users SET arr = arr || 11 WHERE name = 'Bob'; -- 2
UPDATE users SET arr = array_append(arr, 12) WHERE name = 'Bob'; -- 3
UPDATE users SET arr = array_prepend(13, arr) WHERE name = 'Bob';-- 4
UPDATE users SET arr = array_remove(arr, 2) WHERE name = 'Bob'; -- 5
UPDATE users SET arr[4] = 1, arr[7] = 10 WHERE name = 'Bob'; -- 6
SELECT * FROM users WHERE array_position(arr, 12) IS NOT NULL; --7
SELECT * FROM users WHERE arr @> '{5, 6}'::integer[];
```

В строке (1) добавляем запись, содержащую массив. В (2) с помощью конкатенации добавляем в массив число 11. В (3) добавили запись в конец массива, в (4) в начало. В 5 удалили элемент **по значению**, а не по индексу. В (6) изменили значения массива, ячейки выбираись по индексу, индексация начинается **с 1, а не с 0**. В (7) проверили что в массиве присутствует число 12, array_position вернул индекс ячейки, в которой лежит 12, если бы число отсутствовало, то вернулся бы NULL. В (8) проверили что arr есть все элементы из массива {5, 6}.

© ВТБ совместно с GeekBrains

Работа с таблицами

Модификация таблиц

Модифицировать таблицы приходится по различным причинам. Например, при необходимости добавить к какому-нибудь атрибуту ограничение DEFAULT, т. е. значение «по умолчанию». Конечно, если в таблицах еще нет данных, то их можно просто пересоздать, внеся изменения в их определения. Но если таблицы содержат большое количество строк, то пересоздать их не всегда возможно, в этом случае на помощь приходит команда ALTER TABLE. Эта команда предоставляет большое количество возможностей.

Для **добавления столбца** используется **ADD COLUMN**. Если в таблице есть записи и мы выполним следующий запрос.

```
ALTER TABLE students ADD COLUMN mark int2 NOT NULL CHECK( mark < 6);
```

В таком случае СУБД выдает сообщение об ошибке: «ОШИБКА: столбец "mark" содержит значения NULL». Дело в том, что в таблице students уже есть строки и при добавлении нового столбца, значения в этом столбце окажутся равными NULL, а наложенное ограничение NOT NULL запрещает это.

Чтобы выйти из этой ситуации, придется сначала добавить столбец, не накладывая на его значения никаких ограничений, затем ввести значения нового атрибута в уже существующие строки, причем эти значения должны удовлетворять тем ограничениям, которые мы собираемся наложить. После этого накладываем все необходимые ограничения.

```
ALTER TABLE students ADD COLUMN mark int2;
UPDATE students SET mark = 4 WHERE id = 1;
UPDATE students SET mark = 3 WHERE id = 2;
UPDATE students SET mark = 5 WHERE id = 3;
ALTER TABLE students ALTER COLUMN mark SET NOT NULL;
ALTER TABLE students ADD CHECK (mark < 6);
```

Для **удаления ограничения** используется **ALTER COLUMN DROP** или **DROP CONSTRAINT**.

```
ALTER TABLE students ALTER COLUMN group_name DROP NOT NULL;
ALTER TABLE students DROP CONSTRAINT mark_check;
```

Для **удаления столбца** используется **DROP COLUMN**. При этом не обязательно предварительно удалять ограничения, наложенные на этот столбец.

```
ALTER TABLE students DROP COLUMN score;
```

Для изменения типа столбца используется **SET DATA TYPE**.

```
ALTER TABLE students ALTER COLUMN serial_number SET DATA TYPE bigint;
```

© ВТБ совместно с GeekBrains

В том случае, когда один тип данных изменяется на другой тип данных в пределах одной группы, например, оба типа — числовые, то проблем обычно не возникает. Если исходный и целевой типы данных относятся к разным группам, то потребуются дополнительные действия. Например, если столбец имел тип text, а мы хотим преобразовать его в integer. Для реализации такой задачи служит фраза **USING** команды **ALTER TABLE**.

```
ALTER TABLE students ALTER COLUMN mark SET DATA TYPE integer USING ( CASE WHEN
mark = 'A' THEN 5 WHEN mark = 'B' THEN 3 ELSE 2 END);
```

Однако такой вариант команды не сработает, если на столбце mark были наложены ограничения по набору возможных значений (CHECK (mark >= 2 AND mark <= 5)). В этом случае придется их убрать, заменить тип данных, и потом вернуть ограничения обратно с учетом нового типа данных.

Представления

При работе с базами данных зачастую приходится многократно выполнять одни и те же запросы, которые могут быть весьма сложными и требовать обращения к нескольким таблицам. Чтобы избежать необходимости многократного формирования таких запросов, можно использовать так называемые представления (views).

Если речь идет о выборке данных, то представления практически неотличимы от таблиц с точки зрения обращения к ним в командах SELECT. Упрощенный синтаксис команды CREATE VIEW, предназначенной для создания представлений, таков:

```
CREATE [OR REPLACE] VIEW имя-представления [ ( имя-столбца [, ...] ) ] AS
запрос;
```

Для удаления существующего представления можно воспользоваться

```
DROP VIEW [IF EXISTS] имя_таблицы;
```

В этой команде обязательными элементами являются имя представления и запрос к базе данных, который и формирует выборку из нее. Если список имен столбцов не указан, тогда их имена формируются на основании текста запроса. Теперь мы можем вместо написания сложного первоначального запроса обращаться непосредственно к представлению, как будто это обычная таблица.

```
SELECT * FROM имя_представления;
```

В отличие от таблиц, представления не содержат данных. При каждом обращении к представлению в команде SELECT данные выбираются из таблиц, на основе которых это представление создано.

СУБД PostgreSQL предлагает свое расширение команды CREATE VIEW, а именно — фразу OR REPLACE. Если представление уже существует, то можно его не удалять, а просто заменить новой версией. Однако нужно помнить о том, что при создании новой версии представления (без явного удаления старой с помощью команды DROP VIEW) должны оставаться неизменными имена столбцов представления, в противном случае надо удалить представление.

© ВТБ совместно с GeekBrains

При создании представлений существует возможность задавать имена столбцов можно явно указать в команде, но если они не указаны, то СУБД сама «вычислит» эти имена. Это можно сделать двумя способами: первый заключается в том, чтобы создать псевдоним для этого столбца с помощью ключевого слова AS, а второй — в указании списка имен столбцов в начале команды CREATE VIEW.

Представления позволяют облегчить развитие и модификацию базы данных, потому что они могут позволить сохранить интерфейс неизменным, но сам запрос, который лежит в основе конкретного представления, может измениться. При этом для прикладного программиста представление останется неизменным, поэтому не потребуется переделывать запросы к этому представлению в прикладной программе.

PostgreSQL предлагает свое расширение — так называемое материализованное представление. Упрощенный синтаксис команды CREATE MATERIALIZED VIEW, предназначенной для создания материализованных представлений, таков:

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] имя-мат-представления [ ( имя-столбца [, ...] ) ] AS запрос [ WITH [ NO ] DATA ];
```

В момент выполнения команды создания материализованного представления оно заполняется данными, но только если в команде не было фразы WITH NO DATA. Если же она была включена в команду, тогда в момент своего создания представление остается пустым, а для заполнения его данными нужно использовать команду **REFRESH MATERIALIZED VIEW**. Материализованное представление очень похоже на обычную таблицу. Однако оно отличается от таблицы тем, что не только сохраняет данные, но также запоминает запрос, с помощью которого эти данные были собраны. Как и любой другой объект базы данных, материализованное представление можно удалить.

```
DROP MATERIALIZED VIEW mat_view;
```

Положительные стороны использования представлений:

1. Упрощение разграничения полномочий пользователей на доступ к хранимым данным. Разным типам пользователей могут требоваться различные данные, хранящиеся в одних и тех же таблицах. Это касается как столбцов, так и строк таблиц. Создание различных представлений для разных пользователей избавляет от необходимости создавать дополнительные таблицы, дублируя данные, и упрощает организацию системы управления доступом к данным.

2. Упрощение запросов к базе данных. Запросы к базе данных могут включать несколько таблиц и быть весьма сложными и громоздкими, при этом такие запросы могут выполняться часто. Использование представлений позволяет скрыть эти сложности от прикладного программиста и сделать запросы более простыми и наглядными.

3. Снижение зависимости прикладных программ от изменений структуры таблиц базы данных. В процессе развития информационной системы структура таблиц базы данных может изменяться. Столбцы представления, т. е. их имена, типы данных и порядок следования, — это, образно говоря, интерфейс к запросу, который реализуется данным представлением. Если этот интерфейс остается неизменным, то SQL-запросы, в которых используется данное представление, корректировать не потребуется. Нужно будет лишь в ответ на изменение структуры базовых таблиц, на основе которых представление сконструировано, соответствующим образом перестроить запрос, выполняемый данным представлением.

4. Снижение времени выполнения сложных запросов за счет использования материализованных представлений. В материализованных представлениях можно сохранять результаты выполнения запросов, которые формируются длительное время, но при этом допускают их формирование заранее, а не обязательно в момент возникновения потребности в результатах этого запроса. Одним из недостатков является то, что их необходимо своевременно обновлять с помощью команды REFRESH, чтобы они содержали актуальные данные.

© ВТБ совместно с GeekBrains

Схемы базы данных

Схема — это логический фрагмент базы данных, в котором могут содержаться различные объекты: таблицы, представления, индексы и др. В базе данных обязательно есть хотя бы одна схема. При создании базы данных в ней автоматически создается схема с именем public.

В каждой базе данных может содержаться более одной схемы. Их имена должны быть уникальными в пределах конкретной базы данных. Имена объектов базы данных (таблиц, представлений, последовательностей и др.) должны быть уникальными в пределах конкретной схемы, но в разных схемах имена объектов могут повторяться. Таким образом, можно сказать, что схема образует так называемое пространство имен.

Если в базе данных присутствует две схемы: test и public, то доступ к объектам в этих схемах можно организовать как имя_схемы_имя_объекта.

```
SELECT * FROM test.students;
```

Однако такой способ не очень удобен. Другой способ заключается в том, чтобы одну из схем сделать текущей. Среди параметров времени исполнения, которые предусмотрены в конфигурации сервера PostgreSQL, есть параметр search_path. Его значение по умолчанию можно изменить в конфигурационном файле postgresql.conf. Он содержит имена схем, которые PostgreSQL просматривает при поиске конкретного объекта базы данных, когда имя схемы в команде не указано. Посмотреть значение этого параметра можно с помощью команды SHOW:
SHOW search_path;

Чтобы указать search_path таким образом, чтобы PostgreSQL искал объекты первым делом в указанной схеме необходимо выполнить:

```
SET search_path = test;
```

Схема test в таком случае и станет текущей.

Транзакции

Транзакция — это совокупность операций над базой данных, которые вместе образуют логически целостную процедуру, и могут быть либо выполнены все вместе, либо не будет выполнена ни одна из них. В простейшем случае транзакция состоит из одной операции. Транзакции являются одним из средств обеспечения согласованности базы данных, наряду с ограничениями целостности (constraints), накладываемыми на таблицы.

Транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние. Транзакция может иметь два исхода: первый — изменения данных, произведенные в ходе ее выполнения, успешно зафиксированы в базе данных, а второй — транзакция отменяется, и отменяются все изменения, выполненные в ее рамках.

Отмена транзакции называется откатом (rollback). Сложные информационные системы, как правило, предполагают одновременную работу многих пользователей с базой данных, поэтому современные СУБД предлагают специальные механизмы для организации параллельного выполнения транзакций.

Реализация транзакций в СУБД PostgreSQL основана на многоверсионной модели (Multiversion Concurrency Control, MVCC). Эта модель предполагает, что каждый SQL оператор видит

© ВТБ совместно с GeekBrains

так называемый снимок данных (snapshot), т. е. то согласованное состояние (версию) базы данных, которое она имела на определенный момент времени. При этом параллельно исполняемые транзакции, даже вносящие изменения в базу данных, не нарушают согласованности данных этого снимка. Такой результат в PostgreSQL достигается за счет того, что когда параллельные транзакции изменяют одни и те же строки таблиц, тогда создаются отдельные версии этих строк, доступные соответствующим транзакциям. Это позволяет ускорить работу с базой данных, однако требует больше дискового пространства и оперативной памяти. И еще одно важное следствие применения MVCC — операции чтения никогда не блокируются операциями записи, а операции записи никогда не блокируются операциями чтения.

Транзакции должны обладать следующими свойствами (аббревиатура ACID):

1. **Атомарность** (atomicity) – транзакция будет зафиксирована в базе данных полностью, либо не будет зафиксирована ни одна операция транзакции.

2. **Согласованность** (consistency) – в результате успешного выполнения транзакции база данных была переведена из одного согласованного состояния в другое согласованное состояние.

3. **Изолированность** (isolation) – во время выполнения транзакции другие транзакции должны оказывать по возможности минимальное влияние на нее.

4. **Долговечность** (durability) – После успешной фиксации транзакции пользователь должен быть уверен, что данные надежно сохранены в базе данных и впоследствии могут быть извлечены из нее, независимо от последующих возможных сбоев в работе системы.

При параллельном выполнении транзакций возможны следующие феномены:

1. **Потерянное обновление** (lost update). Когда разные транзакции одновременно изменяют одни и те же данные, то после фиксации изменений может оказаться, что одна транзакция перезаписала данные, обновленные и зафиксированные другой транзакцией.

2. **«Грязное» чтение** (dirty read). Транзакция читает данные, измененные параллельной транзакцией, которая еще не завершилась. Если эта параллельная транзакция в итоге будет отменена, тогда окажется, что первая транзакция прочитала данные, которых нет в системе.

3. **Неповторяющееся чтение** (non-repeatable read). При повторном чтении тех же самых данных в рамках одной транзакции оказывается, что другая транзакция успела изменить и зафиксировать эти данные. В результате тот же самый запрос выдает другой результат.

4. **Фантомное чтение** (phantom read). Транзакция повторно выбирает множество строк в соответствии с одним и тем же критерием. В интервале времени между выполнением этих выборок другая транзакция добавляет новые строки и успешно фиксирует изменения. В результате при выполнении повторной выборки в первой транзакции может быть получено другое множество строк.

5. **Аномалия сериализации** (serialization anomaly). Результат успешной фиксации группы транзакций, выполняющихся параллельно, не совпадает с результатом ни одного из возможных вариантов упорядочения этих транзакций, если бы они выполнялись последовательно.

Перечисленные феномены, а также ситуации, в которых они имеют место, будут рассмотрены подробно и проиллюстрированы примерами. Поясним кратко, в чем состоит смысл концепции сериализации. Для двух транзакций, скажем, A и B, возможны только два варианта упорядочения при их последовательном выполнении: сначала A, затем B или сначала B, затем A. Причем результаты реализации двух вариантов могут в общем случае не совпадать. Например, при выполнении двух банковских операций — внесения некоторой суммы денег на какой-то счет и начисления процентов по этому счету — важен порядок выполнения операций. Если первой операцией будет увеличение суммы на счете, а второй — начисление процентов, тогда итоговая сумма будет больше, чем при противоположном порядке выполнения этих операций. Если описанные операции выполняются в рамках двух различных транзакций, то оказываются возможными различные итоговые результаты, зависящие от порядка их выполнения. Сериализация двух транзакций при их параллельном выполнении означает, что полученный результат будет соответствовать одному из двух возможных вариантов упорядочения транзакций при их последовательном выполнении. При этом нельзя сказать точно, какой из вариантов будет реализован. Если распространить эти рассуждения на случай, когда параллельно выполняется более двух транзакций, тогда результат их параллельного выполнения

© ВТБ совместно с GeekBrains

также должен быть таким, каким он был бы в случае выбора некоторого варианта упорядочения транзакций, если бы они выполнялись последовательно, одна за другой. Конечно, чем больше транзакций, тем больше вариантов их упорядочения. Концепция сериализации не предписывает выбора какого-то определенного варианта. Речь идет лишь об одном из них. В том случае, если СУБД не сможет гарантировать успешную сериализацию группы параллельных транзакций, тогда некоторые из них могут быть завершены с ошибкой. Эти транзакции придется выполнить повторно. Для конкретизации степени независимости параллельных транзакций вводится понятие уровня изоляции транзакций. Каждый уровень характеризуется перечнем тех феноменов, которые на данном уровне не допускаются.

Всего в стандарте SQL предусмотрено четыре уровня. Каждый более высокий уровень включает в себя все возможности предыдущего.

1. **Read Uncommitted**. Это самый низкий уровень изоляции. Согласно стандарту SQL на этом уровне допускается чтение «грязных» (незафиксированных) данных. Однако в PostgreSQL требования, предъявляемые к этому уровню, более строгие, чем в стандарте: чтение «грязных» данных на этом уровне не допускается.

2. **Read Committed**. Не допускается чтение «грязных» (незафиксированных) данных. Таким образом, в PostgreSQL уровень Read Uncommitted совпадает с уровнем Read Committed. Транзакция может видеть только те незафиксированные изменения данных, которые произведены в ходе выполнения ее самой.

3. **Repeatable Read**. Не допускается чтение «грязных» (незафиксированных) данных и неповторяющееся чтение. В PostgreSQL на этом уровне не допускается также фантомное чтение. Таким образом, реализация этого уровня является более строгой, чем того требует стандарт SQL. Это не противоречит стандарту.

4. **Serializable**. Не допускается ни один из феноменов, перечисленных выше, в том числе и аномалии сериализации.

Конкретный уровень изоляции обеспечивает сама СУБД с помощью своих внутренних механизмов. Его достаточно указать в команде при старте транзакции. Однако программист может дополнительно использовать некоторые операторы и приемы программирования, например, устанавливать блокировки на уровне отдельных строк или всей таблицы.

По умолчанию PostgreSQL использует уровень изоляции Read Committed, для проверки можно воспользоваться следующей командой.

```
SHOW default_transaction_isolation;
```

Для старта транзакции используется команда **BEGIN**. Для указания уровня изоляции **SET TRASACTION ISOLATION LEVEL [READ UNCOMMITTED, READ COMMITTED, REPETABLE READ, SERIALIZABLE]**. Для получения текущего уровня изоляции **SHOW transaction_isolation**. Для отмены транзакции можно воспользоваться **ROLLBACK**. Для завершения транзакции **COMMIT (или END для PostgreSQL)**. Для запуска транзакции с определенным уровнем изоляции можно воспользоваться **BEGIN ISOLATION LEVEL [READ COMMITTED]**.

В PostgreSQL реализация уровня изоляции Read Uncommitted более строгая, чем того требует стандарт языка SQL. Фактически этот уровень тождественен уровню изоляции Read Committed.

В случае **READ COMMITED** если две параллельные транзакции попытаются изменить одну и ту же строку, то первая транзакция ее изменит внутри себя, а вторая при попытке апдейста перейдет в состояние ожидания, пока первая не сделать ROLLBACK или COMMIT.

В случае **REPEATABLE READ** транзакция создает снимок данных перед выполнением первого запроса транзакции, и если возникнет необходимость изменить строки, измененные другой транзакцией, то данная транзакция будет отменена. Важно помнить, что повторный запуск может потребоваться только для транзакций, которые вносят изменения в данные. Для транзакций, которые только читают данные, повторный запуск никогда не требуется.

© ВТБ совместно с GeekBrains

В случае **SERIALIZABLE** транзакции могут работать параллельно точно так же, как если бы они выполнялись последовательно одна за другой. Однако, как и при использовании уровня Repeatable Read, приложение должно быть готово к тому, что придется перезапускать транзакцию, которая была прервана системой из-за обнаружения зависимостей чтения/записи между транзакциями. Группа транзакций может быть параллельно выполнена и успешно зафиксирована в том случае, когда результат их параллельного выполнения был бы эквивалентен результату выполнения этих транзакций при выборе одного из возможных вариантов их упорядочения, если бы они выполнялись последовательно, одна за другой.

Блокировки

Кроме поддержки уровней изоляции транзакций, PostgreSQL позволяет также создавать явные блокировки данных как на уровне отдельных строк, так и на уровне целых таблиц. Блокировки могут быть востребованы при проектировании транзакций с уровнем изоляции, как правило, Read Committed, когда требуется более детальное управление параллельным выполнением транзакций.

PostgreSQL предлагает много различных видов блокировок, но мы ограничимся рассмотрением только двух из них. Команда **SELECT** имеет предложение **FOR UPDATE**, которое позволяет заблокировать отдельные строки таблицы с целью их последующего обновления. Если одна транзакция заблокировала строки с помощью этой команды, тогда параллельные транзакции не смогут заблокировать эти же строки до тех пор, пока первая транзакция не завершится, и тем самым блокировка не будет снята.

```
SELECT * FROM students WHERE score > 80 FOR UPDATE;
```

Для организации блокировки на уровне таблицы можно выполнить команду блокировки всей таблицы в самом строгом режиме, в котором другим транзакциям доступ к этой таблице запрещен полностью:

```
LOCK TABLE students IN ACCESS EXCLUSIVE MODE;
```

Практическое задание

- Создайте таблицу студенты (students): id, имя, серия паспорта, номер паспорта;
- Создайте таблицу Предметы (subjects): id, название предмета;
- Создайте таблицу Успеваемость (progress): id, студент, предмет, оценка;
- Оценка может находиться в пределах от 2 до 5;
- Вывести список студентов, сдавших определенный предмет, на оценку выше 3;
- При удалении студента из таблицы, вся его успеваемость тоже должна быть удалена;
- Посчитать средний бал по определенному предмету;
- Посчитать средний балл по определенному студенту;
- Пара серия-номер пасорта должны быть уникальны в таблице Студенты;
- Найти три премета, которые сдали наибольшее количество студентов;

© ВТБ совместно с GeekBrains

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://postgrespro.ru/docs/
2. Е. П. Моргунов PostgreSQL Основы языка SQL, БХВ-Петербург, Санкт-Петербург, 2018. - 336 стр.

© ВТБ совместно с GeekBrains