Аннотация

Данное приложение служит основой для методических указаний к проведению лабораторного практикума по тестированию программного кода. В лабораторный практикум вошли модульное тестирование, тестирование графического пользовательского интерфейса и нагрузочное тестирование. Приложение содержит пять лабораторных работ, в которых приведен теоретический материал для выполнения предложенных заданий.

СОДЕРЖАНИЕ

БрГТУ.150089-06 95 00

Учебно-методическое пособие №1 по теме:

«Разработка структуры БД в 3 нормально форме в соответствии с заданной предметной областью.»

Цель: Разработка структуры БД в 3 нормально форме в соответствии с заданной предметной областью.

Содержание:

Нормализация:

Цель нормализации – это исключить избыточное дублирование данных, которое является причиной аномалий(Аномалия – ситуация в таблице БД, которая приводит к противоречию в БД и усложняет обработку БД), возникших при добавлении, редактировании и удалении кортежей(строк таблицы).

Денормализация – это процесс осознанного приведения базы данных к виду, в котором она не будет соответствовать правилам нормализации. Обычно это необходимо для повышения производительности и скорости извлечения данных, за счет увеличения избыточности данных.

Если приложению необходимо часто выполнять выборки, которые занимают слишком много времени (например, объединение данных из множества таблиц), то следует рассмотреть возможность проведения денормализации.

Возможное решение следующее: вынести результаты выборки в отдельную таблицу. Это позволит увеличить скорость выполнения запросов, но также означает появление необходимости в постоянном обслуживании этой новой таблицы.

Прежде чем приступать к денормализации, необходимо убедится, что ожидаемые результаты оправдывают издержки, с которыми придется столкнуться.

Для того, чтобы понять как же должна выглядеть БД в 3 нормальной форме, нам необходимо узнать про первые две.

Первая нормальная форма: Отношение находится в 1НФ, если все его атрибуты являются простыми, все используемые домены должны содержать только скалярные значения. Не должно быть повторений строк в таблице.

Например, есть таблица 1 «Автомобили»:

Таблица 1 – автомобили

Фирма	Модели
BMW	M1, M5, X6
Mercedes-Benz	C190

Нарушение нормализации 1НФ происходит в моделях BMW, т.к. в одной ячейке содержится список из 3 элементов: M1, M5, X6 т.е. он не является атомарным. Преобразуем таблицу 2 к 1НФ :

Таблица 2 – первая нормальная форма

Фирма	Модели
BMW	M1
BMW	M5
BMW	X6
Mercedes-Benz	C190

Следует познакомиться с нормализацией:

Нормальная форма – требование, предъявляемое к структуре таблиц в теории реляционных баз данных для устранения из базы избыточных функциональных зависимостей между атрибутами(полями таблиц).

Вторая нормальная форма
Отношение находится во 2НФ, если оно находится в 1НФ и каждый не ключевой атрибут неприводимо зависит от Первичного Ключа(ПК). Неприводимость означает, что в составе потенциального ключа отсутствует меньшее подмножество атрибутов, от которого можно также вывести данную функциональную зависимость.

Например, дана таблица 3:

Таблица 3 – первая нормальная форма

Модель	Фирма	Цена	Скидка
M1	BMW	550000	5%
M5	BMW	625000	5%
X6	BMW	300000	5%
C190	Mercedes-Benz	700000	10%

Таблица находится в первой нормальной форме, но не во второй. Цену машины зависит от модели и фирмы. Скидка зависят от фирмы, то есть зависимость от первичного ключа неполная. Исправляется это путем декомпозиции на два отношения, в которых не ключевые атрибуты зависят от ПК.

Таблица 4 – вторая нормальная форма

Модель	Фирма	Цена
M1	BMW	550000
M5	BMW	625000
X6	BMW	300000
C190	Mercedes-Benz	700000

Таблица 5 – вторая нормальная форма

Фирма	Скидка
BMW	5%
Mercedes-Benz	10%

3НФ:

Отношение находится в 3НФ, когда находится во 2НФ и каждый не ключевой атрибут не транзитивно зависит от первичного ключа. Проще говоря, второе правило требует выносить все не ключевые поля, содержимое которых может относиться к нескольким записям таблицы в отдельные таблицы.

Третья нормальная форма:

Таблица 6 – третья нормальная форма

Модель	Магазин	Телефон
BMW	Риал-авто	87-33-99
Audi	Риал-авто	87-33-99
Mercedes-Benz	Топ-авто	88-44-65

Таблица 6 находится во 2НФ, но не в 3НФ.

В отношении атрибут «Модель» является первичным ключом. Личных телефонов у автомобилей нет, и телефон зависит исключительно от магазина. Таким образом, в отношении существуют следующие функциональные зависимости: Модель → Магазин, Магазин → Телефон, Модель → Телефон. Зависимость Модель → Телефон является транзитивной, следовательно, отношение не находится в 3НФ.

В результате разделения исходного отношения получаются два отношения, находящиеся в 3НФ:

Таблица 7 – отношения в третьей нормальной форме

Риал-авто	87-33-99
Риал-авто	87-33-99
Топ-авто	88-44-65

Таблица 8 – отношение в третьей нормальной форме

Модель	Магазин
BMW	Риал-авто
Audi	Риал-авто
Mercedes-Benz	Топ-авто

Нормальная форма Бойса-Кодда.

BCNF - нормальная форма Бойса-Кодда. Иногда нормальную форму Бойса-Кодда называют усиленной третьей нормальной формой, поскольку она во всех отношениях сильнее (строже) по сравнению с 3НФ. Эта нормальная форма вводит дополнительное ограничение по сравнению с 3НФ.

Определение нормальной формы Бойса-Кодда:

Отношение находится в BCNF, если оно находится во 3НФ и в ней отсутствуют зависимости атрибутов первичного ключа от не ключевых атрибутов.

Ситуация, когда отношение будет находится в 3NF, но не в BCNF, возникает при условии, что отношение имеет два (или более) возможных ключа, которые являются составными и имеют общий атрибут. Заметим, что на практике такая ситуация встречается достаточно редко, для всех прочих отношений 3NF и BCNF эквивалентны.

Вывод: в данном лабораторном практикуме раскрываются понятия Третьей нормальной формы и нормализации.

Контрольные вопросы:

1. Что такое нормализация?
2. Что такое денормализация?
3. В чём отличие 2НФ от 3НФ?
4. Определение НФ Бойса-Кодда.

Задания для выполнения:

Привести таблицу к Третьей Нормальной форме по заданным предметным областям:

1. Машины (сущности: Марка, год выпуска, пробег, габариты, принадлежность организациям, техосмотр, год прохождения техосмотра, водитель, гос. знак(id) цена машины)
2. Книжный магазин (сущности: автор, id книги, год издания, издатель, язык написания оригинала, читатель, продавец, цена, дата продажи)
3. Музыка (сущности: исполнитель, дата выхода песни, автор слов песни, патент, менеджер, сборы, сцена, название песни)
4. Кинотеатр (сущности: прокат, сборы, дата премьеры, кинотеатр, автор сценария, продюсер, актёр, id фильма, название фильма)
5. Аэропорт (сущности: страна нахождения, id рейса, номер самолёта, время прибытия, время отправления, количество пассажиров, заработная плата)
6. Ограбление банка (сущности:id ограбления, название банка, запас банка, кличка грабителей, уровень грабителей, вероятность провала)

7. Морской порт (сущности: название порта, вместимость, адрес, начальник порта, начальник смены, смена, работник порта, время прибытия, время отправления, судно, заработная плата)
8. Зоопарк (сущности: смотритель зоопарка, наименование, вольер, отдел, работник зоопарка, id зверя, заработная плата)
9. Университет (сущности: преподаватель, студент, заведующий кафедрой, декан, ректор, предмет, оценки, заработная плата, экзамен, зачет, номер зачётной книги)
10. Поликлиника (сущности: главврач, доктор, медсестра, пациент, болезнь, кабинет, номер карты, отделение, страховой полис, заработная плата)
11. Модельное агентство (сущности: номер модели, рост, вес, возраст, пол, название агентства, фотограф, директор агентства, кличка, спонсоры, фотосессия, заработная плата)
12. Магазин компьютерной техники (сущности: продавец, покупатель, ОЗУ, ПЗУ, цена, акции, скидки, дата покупки, номер чека)
13. Ломбард (сущности: номер чека, продавец, покупатель, ставка, залог, цена, время продаж, наименование товара)
14. Туристическое агентство (сущности:id тура, название тура, страна отправления, страна пребывания, время пребывания, покупатель, тур-агент, название фирмы, транспорт, отель, место посещения, цена)
15. Грузовые перевозки (сущности: перевозчик, отправитель, цена груза, id груза, место отправления, место пребывания, водитель, номер машины, вид груза, имя компании грузоперевозок)
16. Банк (сущности: имя банка, отделение банка, номер счёта, клиент, заработная плата, размер вклада, кредит, банкир, операция, процентная ставка)
17. Театр (сущности: актёр, посетитель, адрес театра, вид представления, имя представления, цена билета, номер билета, место, автор сценария, директор-постановщик, декорации)
18. Парикмахерская (сущности: парикмахер, клиент, цена услуги, вид стрижки, номер чека, имя парикмахерской,)
19. Химчистка (сущности: клиент, номер чека, цена, работник, заработная плата, номер машины, объём машины, акции, карта постоянного клиента, абонемент)
20. Почта (сущности: работник, номер операции, номер чека, посетитель, посылка, отправитель, получатель, место отправления, место доставки, дата отправления, дата доставки, id посылки, заработная плата,)

21. Заправка (сущности: работник, посетитель, цена, заработная плата, вид топлива, дополнительные услуги, номер чека, машина, акции, скидки, вместительность бака машины, дата оказания услуги)
22. Служба доставки (сущности: оператор, курьер, получатель, номер заказа, номер чека, место доставки, название службы доставки, адрес отправления доставки, дополнительные услуги, цена, заработная плата, скидки, акции.)
23. Оператор сотовой связи (сущности: имя оператора, услуги, цена, заработная плата, работник, получатель услуги, регион оказания услуги, дата подключения, дата отключения, номер договора)
24. Автовокзал (сущности: водитель, транспорт, номер транспорта, пассажир, кассир, место отправления, место прибытия, дата отправления, дата прибытия, номер чека, заработная плата, цена)
25. Железнодорожная станция (сущности: машинист, транспорт, номер транспорта, пассажир, кассир, место отправления, место прибытия, дата отправления, дата прибытия, номер чека, заработная плата, цена)
26. Аптека (сущности: фармацевт, провизор, препарат, цена, заработная плата, номер чека, посетитель, рецепт, врач, дата покупки)
27. Зоомагазин (сущности: посетитель, продавец, заработная плата, цена, товар, животное, дата покупки, номер чека)
28. Жилой дом (сущности: жилец, номер квартиры, номер договора, компания ЖЭК, консьерж, уборщик, дворник, дата ввода в эксплуатацию дома, заработная плата, цена коммунальных услуг)
29. Автосалон (сущности: работник, заработная плата, цена, посетитель, продавец, директор, id машины, марка машины, год выпуска, вид машины)
30. Охранное агентство (сущности: охранник, администратор, директор, номер договора, получатель услуг, время начала оказания услуг, время окончания оказания услуг, заработная плата, цена)

БрГТУ.150089-06 95 00

Учебно-методическое пособие № 2 по теме:

«Реализация запросов и представлений по заданным параметрам в PostgreSQL»

Цель: Рассмотрение реализации запросов к базе данных.

Содержание:

Предложение `WHERE`

Предложение `WHERE` записывается так:

```
WHERE  условие_ограничения
```

где условие_ограничения — любое выражение значения, выдающее результат типа boolean.

После обработки предложения FROM каждая строка полученной виртуальной таблицы проходит проверку по условию ограничения. Если результат условия равен true, эта строка остаётся в выходной таблице, а иначе (если результат равен false или NULL) отбрасывается. В условии ограничения, как правило, задействуется минимум один столбец из таблицы, полученной на выходе FROM. Хотя строго говоря, это не требуется, но в противном случае предложение WHERE будет бессмысленным.

Пример запроса с WHERE:

```
SELECT … FROM tabl1 WHERE name = Alex
```

tabl1 – название таблицы откуда происходит выборка, т.е. строки где колонка name не равна Alex исключаются из blabla и не выводятся на экран.

Метки столбцов:

Элементам в списке выборки можно назначить имена для последующей обработки, например, для указания в предложении ORDER BY.

Рассмотрим на примере:

```
SELECT A AS value, b + c AS sum FROM …
```

Если выходное имя столбца не определено (с помощью AS), система назначает имя сама. Для простых ссылок на столбцы этим именем становится имя целевого столбца, а для вызовов функций это имя функции. Для сложных выражений система генерирует некоторое подходящее имя.

Слово AS можно опустить, но только если имя нового столбца не является ключевым словом PostgreSQL . Во избежание случайного совпадения имени с ключевым словом это имя можно заключить в кавычки. Например, VALUE — ключевое слово, поэтому такой вариант не будет работать:

```
SELECT a  value, b + c AS sum FROM …
```

А такой будет:

```
SELECT a  “value”, b + c AS sum FROM …
```

Для предотвращения конфликта с ключевыми словами, которые могут появиться в будущем, рекомендуется всегда писать AS или заключать метки выходных столбцов в кавычки.

DISTINCT

После обработки списка выборки в результирующей таблице можно дополнительно исключить дублирующиеся строки. Для этого сразу после SELECT добавляется ключевое слово DISTINCT:

```
SELECT DISTINCT список_выборки …
```

(Чтобы явно включить поведение по умолчанию, когда возвращаются все строки, вместо DISTINCT можно указать ключевое слово ALL.)

Две строки считаются разными, если они содержат различные значения минимум в одном столбце. При этом значения NULL полагаются равными.

Предложение DISTINCT ON не описано в стандарте SQL и иногда его применение считается плохим стилем из-за возможной неопределённости в результатах. При разумном использовании GROUP BY и подзапросов во FROMможно обойтись без этой конструкции, но часто она бывает удобнее.

ПРЕДСТАВЛЕНИЕ (VIEW) объект данных который не содержит никаких данных его владельца. Это - тип таблицы, чье содержание выбирается из других таблиц с помощью выполнения запроса. Поскольку значения в этих таблицах меняются, то автоматически, их значения могут быть показаны представлением. Использование представлений основанных на улучшенных средствах запросов,

таких как объединение и подзапрос, разработанных очень тщательно, в некоторых случаях даст больший выигрыш по сравнению с запросами.

Представление - это фактически запрос, который выполняется всякий раз, когда представление становится темой команды. Вывод запроса при этом в каждый момент становится содержанием представления.

КОМАНДА CREATE VIEW:

Вы создаете представление командой CREATE VIEW. Она состоит из слов CREATE VIEW (создать представление), имени представления которое нужно создать, слова AS (как), и далее запроса, как в следующем примере:

```
CREATE VIEW Londonstaff
          AS SELECT *
          FROM Salespeople
          WHERE city = 'London';
```

Теперь мы имеем представление, называемое Londonstaff. Вы можете использовать это представление точно так же как и любую другую таблицу. Она может быть запрошена, модифицирована, вставлена в, удалена из, и соединена с, другими таблицами и представлениями.

```
               SELECT *
               FROM Londonstaff;


         ===============  SQL Execution Log ============
        |                                               |
        | SELECT *                                      |
        | FROM  Londonstaff;                            |
        |                                               |
        | ==============================================|
        |   snum      sname         city         comm   |
        | ------    ----------   -----------   -------  |
        |   1001      Peel         London       0.1200  |
        |   1004      Motika       London       0.1100  |
        |                                               |
         ===============================================
```

Когда вы приказываете SQL выбрать(SELECT) все строки (*) из представления, он выполняет запрос содержащий в определении - Loncfonstaff, и возвращает все из его вывода.

Представления значительно расширяют управление вашими данными. Это - превосходный способ дать публичный доступ к некоторой, но не всей информации в таблице.

ГРУППОВЫЕ ПРЕДСТАВЛЕНИЯ:

Групповые представления - это представления, который содержит предложение GROUP BY, или который основывается на других групповых представлениях. Групповые представления могут стать превосходным способом обрабатывать полученную информацию непрерывно. Предположим, что каждый день вы должны следить за порядком номеров заказчиков, номерами продавцов, принимающих порядки, номерами порядков, средним от порядков, и общей суммой приобретений в порядках.

Чем конструировать каждый раз сложный запрос, можно просто создать следующее представление:

```
CREATE VIEW Totalforday
       AS SELECT odate, COUNT (DISTINCT cnum), COUNT
                (DISTINCT snum), COUNT (onum), AVG
                (amt), SUM (amt)
                FROM Orders
       GROUP BY odate;
```

Теперь можно увидеть всю эту информацию с помощью простого запроса:

```
SELECT *
   FROM Totalforday;
```

SQL запросы могут дать полный комплекс возможностей, так что представления обеспечивают чрезвычайно гибким и мощным инструментом чтобы определить точно, как наши данные могут быть использованы.[1]

Предложение GROUP BY группирует строки таблицы, объединяя их в одну группу при совпадении значений во всех перечисленных столбцах. Порядок, в котором указаны столбцы, не имеет значения. В результате наборы строк с одинаковыми значениями преобразуются в отдельные строки, представляющие все строки группы. Это может быть полезно для устранения избыточности выходных данных и/или для вычисления агрегатных функций, применённых к этим группам.

Например:

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
```

Во втором запросе мы не могли написать SELECT * FROM test1 GROUP BY x, так как для столбца y нет единого значения, связанного с каждой группой. Однако столбцы, по которым выполняется группировка, можно использовать в списке выборки, так как они имеют единственное значение в каждой группе.

INNER JOIN

Это наиболее часто используемое в SQL соединение. Оно возвращает пересечение двух множеств. В терминах таблиц, оно возвращает только записи из обеих таблиц, отвечающие указанному критерию. Результат операции – закрашенная область. (Рисунок 1)

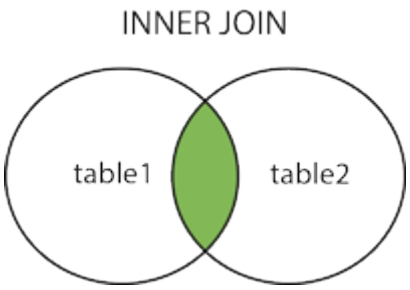

Рисунок 1

FULL JOIN

Полностью это соединение называется FULL OUTER JOIN (зарезервированное слово OUTER необязательно). FULL JOIN работает как объединение двух множеств. (Рисунок 2)

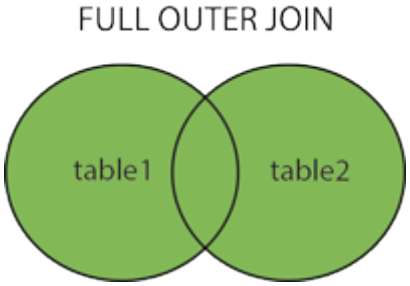

Рисунок 2

LEFT JOIN

Также известен как LEFT OUTER JOIN, и является частным случаем FULL JOIN. Дает все запрошенные данные из таблицы в левой части JOIN плюс данные из правой таблицы, пересекающиеся с первой таблицей. (Рисунок 3)

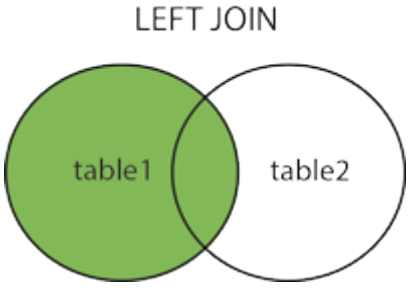

Рисунок 3

RIGHT JOIN

Также известен как RIGHT OUTER JOIN, и является еще одним частным случаем FULL JOIN. Он выдает все запрошенные данные из таблицы, стоящей

в правой части оператора JOIN, плюс данные из левой таблицы, пересекающиеся с правой. (Рисунок 4) [2]

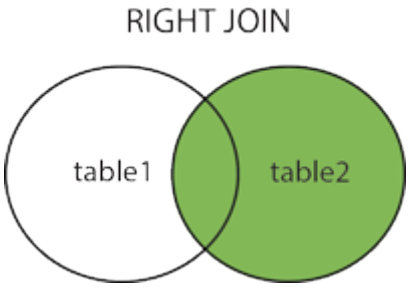

Рисунок 4

Агрегатные выражения

Агрегатное выражение представляет собой применение агрегатной функции к строкам, выбранным запросом. Агрегатная функция сводит множество входных значений к одному выходному, как например, сумма или среднее. Агрегатное выражение может записываться следующим образом:

```
агрегатная_функция (выражение [ , ... ] [ предложение_order_by ] ) [ FILTER
( WHERE условие_фильтра ) ]

агрегатная_функция (ALL выражение [ , ... ] [ предложение_order_by ] ) [ FI
LTER ( WHERE условие_фильтра ) ]

агрегатная_функция (DISTINCT выражение [ , ... ] [ предложение_order_by ] )
[ FILTER ( WHERE условие_фильтра ) ]

агрегатная_функция ( * ) [ FILTER ( WHERE условие_фильтра ) ]

агрегатная_функция ( [ выражение [ , ... ] ] ) WITHIN GROUP ( предложение_о
rder_by ) [ FILTER ( WHERE условие_фильтра ) ]
```

Здесь агрегатная_функция — имя ранее определённой агрегатной функции (возможно, дополненное именем схемы), выражение — любое выражение значения, не содержащее в себе агрегатного выражения или вызова оконной функции.

В первой форме агрегатного выражения агрегатная функция вызывается для каждой строки. Вторая форма эквивалентна первой, так как указание ALL подразумевается по умолчанию. В третьей форме агрегатная функция вызывается для всех различных значений выражения (или набора различных значений, для нескольких выражений), выделенных во входных данных. В четвёртой форме агрегатная функция вызывается для каждой строки, так как никакого конкретного значения не указано (обычно это имеет смысл только для функции count(*)). В последней форме используются сортирующие агрегатные функции, которые будут описаны ниже.

Большинство агрегатных функций игнорируют значения NULL, так что строки, для которых выражения выдают одно или несколько значений NULL, отбрасываются. Это можно считать истинным для всех встроенных операторов, если явно не говорится об обратном.

Например, count(*) подсчитает общее количество строк, а count(f1) только количество строк, в которых f1 не NULL (так как count игнорирует NULL), а count(distinct f1) подсчитает число различных и отличных от NULL значений колонки f1.[3]

Вывод: в данной лабораторной работе рассмотрены основные виды запросов и представлений.

Задания для выполнения:

Предметная область – машины
Выборки:

- Выбрать машины у которых закончился срок действия техосмотра.
- Выбрать водителей, у которых машина моложе двух лет и расположить в порядке убывания.
- Выбрать группу из 10 негабаритных машин разных марок и годов выпуска.
- Выбрать машины, у которых не истёк срок прохождения техосмотра на 2019 год..

Представления:

- Марка, пробег, габариты, техосмотр, водителей, машин 2010 года выпуска.
- Определить среднюю цену машины 1995 года выпуска, пробег, габариты, марка, принадлежность организации.
- Год, распределение машин по пройденному техосмотру в этом году, если техосмотр закончился ранее, то не учитывать эти машины.
- Выписать все машины марки «BMW» выпущенных с 2002 по 2019 год в порядке возрастания.

Предметная область – Книжный магазин

Выборки:

- Определить для каждого автора те случаи, когда произведения не переводились на другие языки.
- Вывести автора книги и жанр произведения которые разошлись тиражом более 7000 экземпляров.
- Определить количество произведений «Война и мир» их общую стоимость и имя издательства.
- Вывести список автором, количество их произведений, язык написания оригинала .

Представления

- Творческое направление, количество человек, количество книг, стоимость которых превышает 200 рублей.
- Тираж направление автор страна – только для произведений, которые переведены на 10 языков.
- Страна, количество издательств, которые опубликовались в этой стране общая стоимость всего литературного наследия.
- Направление, количество авторов, для которых это направление не было основным.

Предметная область – музыка

Выборки

- Определить количество исполнителей, которые заказывали текст песен у одного автора.
- Определить количество музыкальных произведений, исполненных группой «дзидзьо» и общие сборы за период существования группы.
- Выбрать исполнителей, которые выпустили свои последние песни в один день.
- Выбрать названия групп, в которых исполнителем был «Merlin Manson».

Представления

- Группы, количество исполнителей работающий в жанре «рок» сборы которых превышали 100000 долларов.
- Выбрать группу из 10 самых успешных менеджеров (менеджеров с группой у которой самые большие сборы).
- Выбрать группу которая выступила более чем на 30 сценах в одной стране.
- Выбрать названия групп которые выступили на сцене «Славянский базар» в период с 2017 по 2018.

БрГТУ.150089-06 95 00

Методическое пособие № 3 по теме:

«Изучение постреляционных особенностей на примере наследования таблиц и многомерных полей»

Цель: изучение постреляционных особенностей на примере наследования таблиц и многомерных полей.

Содержание:

Наследование

PostgreSQL реализует наследование таблиц, что может быть полезно для проектировщиков баз данных.

Начнём со следующего примера: предположим, что мы создаём модель данных для городов. В каждом штате есть множество городов, но лишь одна столица. Мы хотим иметь возможность быстро получать город-столицу для любого штата. Это можно сделать, создав две таблицы: одну для столиц штатов, а другую для городов, не являющихся столицами. Однако, что делать, если нам нужно получить информацию о любом городе, будь то столица штата или нет? В решении этой проблемы может помочь наследование. Мы определим таблицу capitalskак наследника cities:

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- в футах
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

В этом случае таблица capitals наследует все столбцы своей родительской таблицы, cities. Столицы штатов также имеют дополнительный столбец state, в котором будет указан штат.

В PostgreSQL таблица может наследоваться от нуля или нескольких других таблиц, а запросы могут выбирать все строки родительской таблицы или все строки родительской и всех дочерних таблиц. По умолчанию принят последний вариант. Например, следующий запрос найдёт названия всех городов, включая столицы штатов, расположенных выше 500 футов:

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;
```

Для данных из введения он выдаст:

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

А следующий запрос находит все города, которые не являются столицами штатов, но также находятся на высоте выше 500 футов:

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;

   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
```

Здесь ключевое слово ONLY указывает, что запрос должен применяться только к таблице cities, но не к таблицам, расположенным ниже cities в иерархии наследования. Многие операторы, которые мы уже обсудили, – SELECT, UPDATE и DELETE – поддерживают ключевое слово ONLY.

Вы также можете добавить после имени таблицы *, чтобы обрабатывались и все дочерние таблицы:

```
SELECT name, altitude
    FROM cities*
```

```
    WHERE altitude > 500;
```

Указывать * не обязательно, так как теперь это поведение подразумевается по умолчанию (если только вы не измените параметр конфигурации sql_inheritance). Однако такая запись может быть полезна тем, что подчеркнёт использование дополнительных таблиц.

В некоторых ситуациях бывает необходимо узнать, из какой таблицы выбрана конкретная строка. Для этого вы можете воспользоваться системным столбцом tableoid, присутствующим в каждой таблице:

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

Этот запрос выдаст:

```
 tableoid |   name    | altitude
----------+-----------+----------
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

Механизм наследования не способен автоматически распределять данные команд INSERT или COPY по таблицам в иерархии наследования. Поэтому в нашем примере этот оператор INSERT не выполнится:

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('Albany', NULL, NULL, 'NY');
```

Мы могли надеяться на то, что данные каким-то образом попадут в таблицу capitals, но этого не происходит: INSERT всегда вставляет данные непосредственно в указанную таблицу.

Дочерние таблицы автоматически наследуют от родительской таблицы ограничения-проверки и ограничения NOT NULL (если только для них не задано явно NO INHERIT). Все остальные ограничения (уникальности, первичный ключ и внешние ключи) не наследуются.

Таблица может наследоваться от нескольких родительских таблиц, в этом случае она будет объединять в себе все столбцы этих таблиц, а также столбцы,

описанные непосредственно в её определении. Если в определениях родительских и дочерней таблиц встретятся столбцы с одним именем, эти столбцы будут «объединены», так что в дочерней таблице окажется только один столбец. Чтобы такое объединение было возможно, столбцы должны иметь одинаковый тип данных, в противном случае произойдёт ошибка. Наследуемые ограничения-проверки и ограничения NOT NULL объединяются подобным образом. Так, например, объединяемый столбец получит свойство NOT NULL, если какое-либо из порождающих его определений имеет свойство NOT NULL. Ограничения-проверки объединяются, если они имеют одинаковые имена; но если их условия различаются, происходит ошибка.

Отношение наследования между таблицами обычно устанавливается при создании дочерней таблицы с использованием предложения INHERITS оператора CREATE TABLE. Другой способ добавить такое отношение для таблицы, определённой подходящим образом — использовать INHERIT с оператором ALTER TABLE. Для этого будущая дочерняя таблица должна уже включать те же столбцы (с совпадающими именами и типами), что и родительская таблица. Также она должна включать аналогичные ограничения-проверки (с теми же именами и выражениями). Удалить отношение наследования можно с помощью указания NO INHERIT оператора ALTER TABLE.

Для создания таблицы, которая затем может стать наследником другой, удобно воспользоваться предложением LIKE оператора CREATE TABLE. Такая команда создаст новую таблицу с теми же столбцами, что имеются в исходной. Если в исходной таблицы определены ограничения CHECK, для создания полностью совместимой таблицы их тоже нужно скопировать, и это можно сделать, добавив к предложению LIKE параметр INCLUDING CONSTRAINTS.

Родительскую таблицу нельзя удалить, пока существуют унаследованные от неё. Так же как в дочерних таблицах нельзя удалять или модифицировать столбцы или ограничения-проверки, унаследованные от родительских таблиц. Если вы хотите удалить таблицу вместе со всеми её потомками, это легко сделать, добавив в команду удаления родительской таблицы параметр CASCADE.

При изменениях определений и ограничений столбцов команда ALTER TABLE распространяет эти изменения вниз в иерархии наследования. Однако удалить столбцы, унаследованные дочерними таблицами, можно только с помощью параметра CASCADE. При создании отношений наследования команда ALTER

TABLE следует тем же правилам объединения дублирующихся столбцов, что и CREATE TABLE.

Гиперкубическая модель данных:

Многомерные базы данных рассматривают данные как кубы, которые являются обобщением электронных таблиц на любое число измерений. Кроме того, кубы поддерживают иерархию измерений и формул без дублирования их определений. Набор соответствующих кубов составляет многомерную базу данных (или хранилище данных).

Кубами легко управлять, добавляя новые значения измерений. В обычном обиходе этим термином обозначают фигуру с тремя измерениями, однако теоретически куб может иметь любое число измерений. На практике чаще всего кубы данных имеют от 4 до 12 измерений. Современный инструментарий часто сталкивается с нехваткой производительности, когда так называемый гиперкуб имеет свыше 10-15 измерений.

Комбинации значений измерений определяют ячейки куба. В зависимости от конкретного приложения ячейки в кубе могут располагаться как разрозненно, так и плотно. Кубы, как правило, становятся разрозненными по мере увеличения числа размерностей и степени детализации значений измерений.

На рис. 1 показан куб, содержащий данные по продажам в двух датских городах, указанных в таблице 1 с дополнительным измерением — «Время». В соответствующих ячейках хранятся данные об объеме продаж. В примере можно обнаружить «факт» — непустую ячейку, содержащую соответствующие числовые параметры — для каждой комбинации время, продукт и город, где была совершена, по крайней мере, одна продажа. В ячейке размещаются числовые значения, связанные с фактом — в данном случае, это объем продаж — единственный параметр.

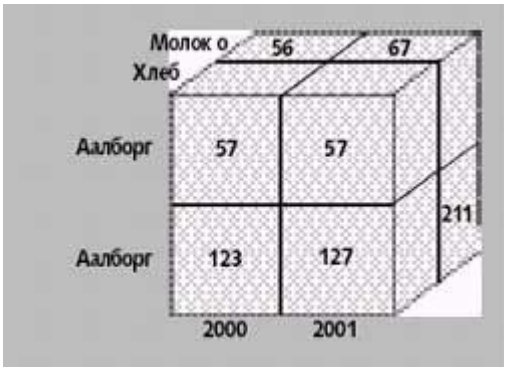

Рис. 5. Пример куба, содержащего данные о продажах.

В этом случае куб обобщает электронную таблицу из Таблицы 1, добавляя к ней третье измерение — время

В общем случае куб позволяет представить только два или три измерения одновременно, но можно показывать и больше за счет вложения одного измерения в другое. Таким образом, путем проецирования куба на двух- или трехмерное пространство можно уменьшить размерность куба, агрегировав некоторые размерности, что ведет к работе с более комплексными значениями параметров. К примеру, рассматривая продажи по городам и времени, мы агрегируем информацию для каждого сочетания город и время. Так, на рис. 5, сложив поля 127 и 211, получаем общий объем продаж для Копенгагена в 2001 году.

БрГТУ.150089-06 95 00

Учебно-методическое пособие № 4 по теме:

«Разделение прав доступа и проведения транзакций в соответствии с уровнем пользователей»

Цель: разделение прав доступа и проведения транзакций в соответствии с уровнем пользователей.

Содержание:

ПРАВА ДОСТУПА

Когда в базе данных создаётся объект, ему назначается владелец. Владельцем обычно становится роль, с которой был выполнен оператор создания. Для большинства типов объектов в исходном состоянии только владелец (или суперпользователь) может делать с объектом всё, что угодно. Чтобы разрешить использовать его другим ролям, нужно дать им права.

Существует несколько типов прав: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE и USAGE. Набор прав, применимых к определённому объекту, зависит от типа объекта (таблица, функция и т. д.)

Неотъемлемое право изменять или удалять объект имеет только владелец объекта.

Объекту можно назначить нового владельца с помощью команды ALTER для соответствующего типа объекта, например ALTER TABLE(ALTER TABLE — изменить определение таблицы. ALTER TABLE меняет определение существующей таблицы. Несколько её разновидностей описаны ниже. Заметьте, что для разных разновидностей могут требоваться разные уровни блокировок. Если явно не отмечено другое, требуется блокировка ACCESS EXCLUSIVE. При перечислении нескольких подкоманд будет запрашиваться самая сильная блокировка из требуемых ими.). Суперпользователь может делать это без ограничений, а обычный

пользователь, только если он является одновременно текущим владельцем объекта (или членом роли владельца) и членом новой роли.

Для назначения прав применяется команда GRANT. Например, если в базе данных есть роль joe и таблица accounts, право на изменение таблицы можно дать этой роли так:

```
GRANT UPDATE ON accounts TO joe;
```

Если вместо конкретного права написать ALL, роль получит все права, применимые для объекта этого типа.

Для назначения права всем ролям в системе можно использовать специальное имя «роли»: PUBLIC.

Чтобы лишить пользователей прав, используется команда REVOKE:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

Особые права владельца объекта (то есть права на выполнение DROP, GRANT, REVOKE и т. д.) всегда неявно закреплены за владельцем и их нельзя назначить или отобрать. Но владелец объекта может лишить себя обычных прав, например, разрешить всем, включая себя, только чтение таблицы.

Обычно распоряжаться правами может только владелец объекта (или суперпользователь). Однако возможно дать право доступа к объекту «с правом передачи», что позволит получившему такое право назначать его другим. Если такое право передачи впоследствии будет отозвано, то все, кто получил данное право доступа (непосредственно или по цепочке передачи), потеряют его.

Для того, чтобы создать нового пользователя используется команда: createuser.

```
Синтаксис: createuser [параметр-подключения...][параметр...][имя_пользователя]
```

Createuser — это обёртка для SQL-команды CREATE ROLE. Создание пользователей с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Createuser принимает следующие аргументы:

имя_пользователя

Задаёт имя создаваемого пользователя PostgreSQL. Это имя должно отличаться от имён всех существующих ролей в данной инсталляции PostgreSQL.

-c номер
--connection-limit=номер

Устанавливает максимальное допустимое количество соединений для создаваемого пользователя. По умолчанию ограничение в количестве соединений отсутствует.

d
--createdb

Разрешает новому пользователю создавать базы данных.

-D
--no-createdb

Запрещает новому пользователю создавать базы данных. Это поведение по умолчанию.

-e
--echo

Вывести команды к серверу, генерируемые при выполнении createuser.

-E
--encrypted

Шифровать пароль пользователя, хранимый в базе. Если флаг не указан, то для пароля используется поведение по умолчанию.

-g role
--role=role

Указывает роль, к которой будет добавлена текущая роль в качестве члена группы. Допускается множественное использование флага -g.

-i
--inherit

Создаваемая роль автоматически унаследует права ролей, в которые она включается. Это поведение по умолчанию.

-I
--no-inherit

Роль не будет наследовать права ролей, в которые она включается.

-l
--login

Новый пользователь сможет подключаться к серверу (то есть его имя может быть идентификатором начального пользователя сеанса). Это свойство по умолчанию.

-L
--no-login

Новый пользователь не сможет подключаться к серверу.

-r
--createrole

Разрешает новому пользователю создавать другие роли, что означает наделение привилегией CREATEROLE.

-s
--superuser

Создаваемая роль будет иметь права суперпользователя.

-S
--no-superuser

Новый пользователь не будет суперпользователем. Это поведение по умолчанию.

--replication

Создаваемый пользователь будет наделён правом REPLICATION.

--no-replication

Создаваемый пользователь не будет иметь привилегии REPLICATION.

-?
--help

Вывести помощь по команде createuser.

createuser также принимает из командной строки параметры подключения:

-h сервер
--host=сервер

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

-p порт
--port=порт

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

-w
--no-password

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся.

-W
--password

Принудительно запрашивать пароль перед подключением к базе данных.

Чтобы создать роль joe на сервере, используемом по умолчанию:

```
$ createuser username
```

Чтобы создать роль username с правами суперпользователя и предустановленным паролем:

```
$ createuser -P -s -e username
Введите пароль для новой роли: xyzzy
Подтвердите ввод пароля: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERU
SER CREATEDB CREATEROLE INHERIT LOGIN;
```

В приведённом примере введённый пароль отображается лишь для отражения сути, на деле же он не выводится на экран. Как видно в выводе журнала команд, пароль зашифрован. Если же указан флаг --unencrypted, то он отобразиться в этом журнале неизменным, а также, возможно, и в других журналах сервера. По этой причине в этой ситуации стоит использовать флаг -e с особой осторожностью.

GRANT

GRANT — определить права доступа

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] имя_таблицы [, ...]
         | ALL TABLES IN SCHEMA имя_схемы [, ...] }
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( имя_столбца [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( имя_столбца [, ...] ) }
    ON [ TABLE ] имя_таблицы [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE имя_последовательности [, ...]
         | ALL SEQUENCES IN SCHEMA имя_схемы [, ...] }
    TO указание_роли [, ...] [ WITH GRANT OPTION
```

Команда GRANT имеет две основные разновидности: первая назначает права для доступа к объектам баз данных (таблицам, столбцам, представлениям, сторонним таблицам, последовательностям, базам данных, обёрткам сторонних данных, сторонним серверам, функциям, процедурным языкам, схемам или табличным пространствам), а вторая назначает одни роли членами других. Эти разновидности во многом похожи, но имеют достаточно отличий, чтобы рассматривать их отдельно.

GRANT ДЛЯ ОБЪЕКТОВ БАЗ ДАННЫХ

Эта разновидность команды GRANT даёт одной или нескольким ролям определённые права для доступа к объекту базы данных. Эти права добавляются к списку имеющихся, если роль уже наделена какими-то правами.

Также можно дать роли некоторое право для всех объектов одного типа в одной или нескольких схемах. Эта функциональность в настоящее время поддерживается только для таблиц, последовательностей и функций (но заметьте,

что указание ALL TABLES распространяется также на представления и сторонние таблицы).

Ключевое слово PUBLIC означает, что права даются всем ролям, включая те, что могут быть созданы позже. PUBLIC можно воспринимать как неявно определённую группу, в которую входят все роли. Любая конкретная роль получит в сумме все права, данные непосредственно ей и ролям, членом которых она является, а также права, данные роли PUBLIC.

Если указано WITH GRANT OPTION, получатель права, в свою очередь, может давать его другим. Без этого указания распоряжаться своим правом он не сможет. Группе PUBLIC право передачи права дать нельзя.

Нет необходимости явно давать права для доступа к объекту его владельцу (обычно это пользователь, создавший объект), так как по умолчанию он имеет все права. (Однако владелец может лишить себя прав в целях безопасности.)

Право удалять объект или изменять его определение произвольным образом не считается назначаемым; оно неотъемлемо связано с владельцем, так что отозвать это право или дать его кому-то другому нельзя. (Однако похожий эффект можно получить, управляя членством в роли, владеющей объектом; см. ниже.) Владелец также неявно получает право распоряжения всеми правами для своего объекта.

PostgreSQL по умолчанию назначает группе PUBLIC определённые права для некоторых типов объектов. Для таблиц, столбцов, последовательностей, обёрток сторонних данных, сторонних серверов, больших объектов, схем или табличных пространств PUBLIC по умолчанию никаких прав не имеет.

Все возможные права перечислены ниже:

SELECT

Позволяет выполнять SELECT для любого столбца или перечисленных столбцов в заданной таблице, представлении или последовательности. Также позволяет выполнять COPY TO. Помимо того, это право требуется для обращения к существующим значениям столбцов в UPDATE или DELETE.

INSERT

Позволяет вставлять строки в заданную таблицу с помощью INSERT. Если право ограничивается несколькими столбцами, только их значение можно будет задать в команде INSERT (другие столбцы получат значения по умолчанию). Также позволяет выполнять COPY FROM.

UPDATE

Позволяет изменять (с помощью UPDATE) данные во всех, либо только перечисленных, столбцах в заданной таблице. (На практике для любой нетривиальной команды UPDATE потребуется и право SELECT, так как она должна обратиться к столбцам таблицы, чтобы определить, какие строки подлежат изменению, и/или вычислить новые значения столбцов.)

DELETE

Позволяет удалять строки из заданной таблицы с помощью DELETE.

TRUNCATE

Позволяет опустошить заданную таблицу с помощью TRUNCATE (Команда TRUNCATE быстро удаляет все строки из набора таблиц. Она действует так же, как безусловная команда DELETE для каждой таблицы, но гораздо быстрее, так как она фактически не сканирует таблицы. Более того, она немедленно высвобождает дисковое пространство, так что выполнять операцию VACUUM после неё не требуется. Наиболее полезна она для больших таблиц.).

REFERENCES

Позволяет создавать ограничение внешнего ключа, ссылающееся на определённую таблицу либо на определённые столбцы таблицы.

TRIGGER

Позволяет создавать триггеры в заданной таблице.

CREATE

Для баз данных это право позволяет создавать схемы и публикации в заданной базе.

CONNECT

Позволяет пользователю подключаться к указанной базе данных. Это право проверяется при установлении соединения (в дополнение к условиям, определённым в конфигурации pg_hba.conf).

TEMP

Позволяет создавать временные таблицы в заданной базе данных.

EXECUTE

Позволяет выполнять заданную функцию и применять любые определённые поверх неё операторы. Это единственный тип прав, применимый к функциям.

ALL PRIVILEGES

Даёт целевой роли все права сразу. Ключевое слово PRIVILEGES является необязательным в PostgreSQL, хотя в строгом SQL оно требуется.

GRANT для ролей

Эта разновидность команды `GRANT` включает роль в члены одной или нескольких других ролей. Членство в ролях играет важную роль, так как права, данные роли, распространяются и на всех её членов.

С указанием WITH ADMIN OPTION новоиспечённый член роли сможет, в свою очередь, включать в члены этой роли, а также исключать из неё другие роли. Без этого указания обычные пользователи не могут это делать. Считается, что роль не имеет права WITH ADMIN OPTION для самой себя, но ей позволено управлять своими членами из сеанса, в котором пользователь сеанса соответствует данной роли. Суперпользователи баз данных могут включать или исключать любые роли из любых ролей. Роли с правом CREATEROLE могут управлять членством в любых ролях, кроме ролей суперпользователей.

В отличие от прав, членство в ролях нельзя назначить группе PUBLIC. Заметьте также, что эта форма команды не принимает избыточное слово GROUP.

REVOKE — отозвать права доступа

Команда REVOKE лишает одну или несколько ролей прав, назначенных ранее. Ключевое слово PUBLIC обозначает неявно определённую группу всех ролей.

Любая конкретная роль получает в сумме права, данные непосредственно ей, права, данные любой роли, в которую она включена, а также права, данные группе PUBLIC. Поэтому, например, лишение PUBLIC права SELECT не обязательно будет означать, что все роли лишатся права SELECT для данного объекта: оно сохранится у тех ролей, которым оно дано непосредственно или косвенно, через другую роль. Подобным образом, лишение права SELECT какого-либо пользователя может не повлиять на его возможность пользоваться правом SELECT, если это право дано группе PUBLIC или другой роли, в которую он включён.

Если указано GRANT OPTION FOR, отзывается только право передачи права, но не само право. Без этого указания отзывается и право, и право распоряжаться им.

Если пользователь обладает правом с правом передачи, и он дал его другим пользователям, последнее право считается зависимым. Когда первый пользователь

лишается самого права или права передачи и существуют зависимые права, эти зависимые права также отзываются, если дополнительно указано CASCADE; в противном случае операция завершается ошибкой. Это рекурсивное лишение прав затрагивает только права, полученные через цепочку пользователей, которую можно проследить до пользователя, являющегося субъектом команды REVOKE. Таким образом, пользователи могут в итоге сохранить это право, если оно было также получено через других пользователей.

Когда отзывается право доступа к таблице, с ним вместе автоматически отзываются соответствующие права для каждого столбца таблицы (если такие права заданы). С другой стороны, если роли были даны права для таблицы, лишение роли таких же прав на уровне отдельных столбцов ни на что не влияет.

Замечания по совместимости, приведённые для команды GRANT, справедливы и для REVOKE. Стандарт требует обязательного указания ключевого слова RESTRICT или CASCADE, но PostgreSQL подразумевает RESTRICT по умолчанию.

Транзакции — это фундаментальное понятие во всех СУБД. Суть транзакции в том, что она объединяет последовательность действий в одну операцию "всё или ничего". Промежуточные состояния внутри последовательности не видны другим транзакциям, и, если что-то помешает успешно завершить транзакцию, ни один из результатов этих действий не сохранится в базе данных.

Например, рассмотрим базу данных банка, в которой содержится информация о счетах клиентов, а также общие суммы по отделениям банка. Предположим, что мы хотим перевести 100 долларов со счёта Алисы на счёт Боба. Простоты ради, соответствующие SQL-команды можно записать так:

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

Точное содержание команд здесь не важно, важно лишь то, что для выполнения этой довольно простой операции потребовалось несколько отдельных действий. При этом с точки зрения банка необходимо, чтобы все эти действия выполнились вместе, либо не выполнились совсем. Если Боб получит 100 долларов, но они не будут списаны со счёта Алисы, объяснить это сбоем системы определённо

не удастся. И наоборот, Алиса вряд ли будет довольна, если она переведёт деньги, а до Боба они не дойдут. Нам нужна гарантия, что если что-то помешает выполнить операцию до конца, ни одно из действий не оставит следа в базе данных. И мы получаем эту гарантию, объединяя действия в одну транзакцию. Говорят, что транзакция атомарна: с точки зрения других транзакций она либо выполняется и фиксируется полностью, либо не фиксируется совсем.

Нам также нужна гарантия, что после завершения и подтверждения транзакции системой баз данных, её результаты в самом деле сохраняются и не будут потеряны, даже если вскоре произойдёт авария. Например, если мы списали сумму и выдали её Бобу, мы должны исключить возможность того, что сумма на его счёте восстановится, как только он выйдет за двери банка. Транзакционная база данных гарантирует, что все изменения записываются в постоянное хранилище (например, на диск) до того, как транзакция будет считаться завершённой.

Другая важная характеристика транзакционных баз данных тесно связана с атомарностью изменений: когда одновременно выполняется множество транзакций, каждая из них не видит незавершённые изменения, произведённые другими. Например, если одна транзакция подсчитывает баланс по отделениям, будет неправильно, если она посчитает расход в отделении Алисы, но не учтёт приход
в отделении Боба, или наоборот. Поэтому свойство транзакций "всё или ничего" должно определять не только, как изменения сохраняются в базе данных, но и как они видны в процессе работы. Изменения, производимые открытой транзакцией, невидимы для других транзакций, пока она не будет завершена, а затем они становятся видны все сразу.

В PostgreSQL транзакция определяется набором SQL-команд, окружённым командами BEGIN и COMMIT. Таким образом, наша банковская транзакция должна была бы выглядеть так:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
COMMIT;
```

Если в процессе выполнения транзакции мы решим, что не хотим фиксировать её изменения (например, потому что оказалось, что баланс Алисы стал отрицательным), мы можем выполнить команду ROLLBACK вместо COMMIT, и все наши изменения будут отменены.

PostgreSQL на самом деле отрабатывает каждый SQL-оператор как транзакцию. Если вы не вставите команду BEGIN, то каждый отдельный оператор

будет неявно окружён командами BEGIN и COMMIT (в случае успешного завершения). Группу операторов, окружённых командами BEGIN и COMMIT иногда называют блоком транзакции.

Операторами в транзакции можно также управлять на более детальном уровне, используя точки сохранения. Точки сохранения позволяют выборочно отменять некоторые части транзакции и фиксировать все остальные. Определив точку сохранения с помощью SAVEPOINT, при необходимости вы можете вернуться к ней с помощью команды ROLLBACK TO. Все изменения в базе данных, произошедшие после точки сохранения и до момента отката, отменяются, но изменения, произведённые ранее, сохраняются.

Всё это происходит в блоке транзакции, так что в других сеансах работы с базой данных этого не видно. Совершённые действия становятся видны для других сеансов все сразу, только когда вы фиксируете транзакцию, а отменённые действия не видны вообще никогда.

Вернувшись к банковской базе данных, предположим, что мы списываем 100 долларов со счёта Алисы, добавляем их на счёт Боба, и вдруг оказывается, что деньги нужно было перевести Уолли. В данном случае мы можем применить точки сохранения:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- ошибочное действие... забыть его и использовать счёт Уолли
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

ROLLBACK TO — это единственный способ вернуть контроль над блоком транзакций, оказавшимся в прерванном состоянии из-за ошибки системы, не считая возможности полностью отменить её и начать снова.

Вывод: в данном методическом пособии рассмотрены понятия прав доступа и транзакции. Приведены примеры с описанием переменных и частями кода.

Контрольные вопросы:

1 В чем отличие обычного пользователя от суперпользователя?

2 Для чего нужны транзакции и точки сохранения в них?

3 Что такое права доступа и как их назначить и(или) отозвать?

Задания для выполнения:

1 Создать новую учетную запись с одним параметром-подключения и тремя параметрами.

2 Определить права доступа для нового пользователя.

3 Отозвать права доступа для нового пользователя.

4 Произвести транзакцию как при пополнении счёта мобильного телефона (снятие определённой суммы и перевод на другой счёт).

Методическое пособие № 5 по теме:

«Проверка скорости работы запросов и их оптимизация с помощью внешней составной искусственной индексации»

Цель: Проверка скорости работы запросов и их оптимизация с помощью внешней составной искусственной индексации.

Содержание:

Pgbench:

Команда pgbench — запустить тест производительности PostgreSQL

Пример:

```
pgbench -i [параметр...] [имя_бд]
pgbench [параметр...] [имя_бд]
```

Команда pgbench — это простая программа для запуска тестов производительности PostgreSQL. Она многократно выполняет одну последовательность команд, возможно в параллельных сеансах базы данных, а затем вычисляет среднюю скорость транзакций (число транзакций в секунду). По умолчанию pgbench тестирует сценарий, примерно соответствующий TPC-B, который состоит из пяти команд SELECT, UPDATE и INSERT в одной транзакции.

Для запускаемого по умолчанию теста типа TPC-B требуется предварительно подготовить определённые таблицы. Чтобы создать и наполнить эти таблицы, следует запустить pgbench с ключом -i (инициализировать).

Запуск инициализации выглядит так:

```
pgbench -i [ другие-параметры ] имя_базы
```

Где ***имя_базы*** — имя уже существующей базы, в которой будет проводиться тест. (Чтобы указать, как подключиться к серверу баз данных, вы также можете добавить параметры ***-h***, ***-p*** и/или ***-U***.)

pgbench -i

Создаёт четыре таблицы: pgbench_accounts, pgbench_branches, pgbench_history и pgbench_tellers, предварительно уничтожая существующие таблицы с этими именами.

С «коэффициентом масштаба», по умолчанию равным 1, эти таблицы изначально содержат такое количество строк:

```
table                   # of rows
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

Эти числа можно (и в большинстве случаев даже нужно) увеличить, воспользовавшись параметром -s(коэффициент масштаба). При этом также может быть полезен ключ -F (фактор заполнения).

Подготовив требуемую конфигурацию, можно запустить тест производительности командой без -i, то есть:

```
pgbench [ параметры ] имя_базы
```

Практически во всех случаях, чтобы получить полезные результаты, необходимо передать какие-либо дополнительные параметры. Наиболее важные параметры: -c (число клиентов), -t (число транзакций), -T(длительность) и -f (файл со скриптом). pgbench принимает следующие аргументы командной строки для тестирования производительности:

-c клиенты
--client=клиенты

Число имитируемых клиентов, то есть число одновременных сеансов базы данных. Значение по умолчанию — 1.

-C
--connect

Устанавливать новое подключение для каждой транзакции вместо одного для каждого клиента. Это полезно для оценивания издержек подключений.

-d
--debug

Выводить отладочные сообщения.

-j потоки
--jobs=потоки

Число рабочих потоков в pgbench. Использовать нескольких потоков может быть полезно в многоядерных системах. Число клиентов должно быть кратно числу потоков, так как каждый поток должен управлять одинаковым числом клиентских сеансов. Значение по умолчанию — 1.

-l
--log

Записать время выполнения каждой транзакции в файл протокола.

CREATE INDEX

Индекс (англ. index) – объект базы данных, создаваемый с целью повышения производительности поиска данных. Таблицы в базе данных могут иметь большое количество строк, которые хранятся в произвольном порядке, и их поиск по заданному критерию путём последовательного просмотра таблицы строка за строкой может занимать много времени. Индекс формируется из значений одного или нескольких столбцов таблицы и указателей на соответствующие строки таблицы и, таким образом, позволяет искать строки, удовлетворяющие критерию поиска. Ускорение работы с использованием индексов достигается в первую очередь за счёт того, что индекс имеет структуру, оптимизированную под поиск – например, сбалансированного дерева.

CREATE INDEX — создать индекс

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] имя ] ON имя_таб
лицы [ USING метод ]
    ( { имя_столбца | ( выражение ) } [ COLLATE правило_сортировки ] [ класс_
операторов ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ INCLUDING ( имя_столбца [, ...] ) ]
    [ WITH ( параметр_хранения = значение [, ... ] ) ]
    [ TABLESPACE табл_пространство ]
    [ WHERE предикат ]
```

CREATE INDEX создаёт индексы по указанному столбцу(ам) заданного отношения, которым может быть таблица или материализованное представление. Индексы применяются в первую очередь для оптимизации производительности базы данных (хотя при неправильном использовании возможен и противоположный эффект).

Выражение в предложении WHERE может ссылаться только на столбцы нижележащей таблицы, но не обязательно ограничиваться теми, по которым

строится индекс. В настоящее время в WHERE также нельзя использовать подзапросы и агрегатные выражения. Это же ограничение распространяется и на выражения в полях индексов.

Параметры расписаны ниже

UNIQUE

Указывает, что система должна контролировать повторяющиеся значения в таблице при создании индекса (если в таблице уже есть данные) и при каждом добавлении данных. Попытки вставить или изменить данные, при которых будет нарушена уникальность индекса, будут завершаться ошибкой.

CONCURRENTLY

С этим указанием Postgres Pro построит индекс, не устанавливая никаких блокировок, которые бы предотвращали добавление, изменение или удаление записей в таблице; тогда как по умолчанию операция построения индекса блокирует запись (но не чтение) в таблице до своего завершения.

IF_NOT_EXISTS

Не считать ошибкой, если индекс с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующий индекс как-то соотносится с тем, который мог бы быть создан. Имя индекса является обязательным, когда указывается IF NOT EXISTS.

NCLUDING

Необязательное предложение INCLUDING позволяет указать список столбцов, которые будут включены в неключевую часть индекса. Столбцы, указанные в этой части, не могут одновременно входить в ключевую часть, и наоборот. Выражение INCLUDING предназначено только для того, чтобы больше запросов могли быть ускорены путём использования сканирования только индекса.

ИМЯ

Имя создаваемого индекса.

имя_таблицы

Имя индексируемой таблицы

метод

Имя применяемого метода индекса. Возможные варианты: btree, hash, gist, spgist, gin и brin. По умолчанию подразумевается метод btree.

имя_столбца

Имя столбца таблицы.

выражение

Выражение с одним или несколькими столбцами таблицы.

класс_операторов

Имя класса операторов.

ASC

Указывает порядок сортировки по возрастанию

DESC

Указывает порядок сортировки по убыванию.

NULLS FIRST

Указывает, что значения NULL после сортировки оказываются перед остальными. Это поведение по умолчанию с порядком сортировки DESC.

NULLS LAST

Указывает, что значения NULL после сортировки оказываются после остальных. Это поведение по умолчанию с порядком сортировки ASC.

параметр_хранения

Имя специфичного для индекса параметра хранения

табл_пространство

Табличное пространство, в котором будет создан индекс. Если не определено, выбирается default_tablespace, либо temp_tablespaces, при создании индекса временной таблицы.

предикат

Выражение ограничения для частичного индекса.

CREATE INDEX является языковым расширением Postgres Pro. Средства обеспечения индексов в стандарте SQL не описаны.

Вывод: в данной лабораторной работе рассмотрены индексация и проверка скорости работы запросов в базе данных.

Контрольные вопросы:

1. Что такое pgbench ?
2. Что такое индекс?

Задания для выполнения:

1. Запустить тест производительности для базы данных без индексов.
2. Добавить в таблицы базы данных 10000 записей
3. Запустить тест производительности для базы данных с большим количеством записей в таблицах.
4. Проиндексировать таблицы и запустить тест производительности.

БрГТУ.150089-06 95 00

Методическое пособие № 6 по теме:

«Использование триггеров и создание пользовательских процедур»

Цель: Использование триггеров и создание пользовательских процедур, знакомство с PL/pgSQL .

Содержание:

Процедурный язык PL/pgSQL :

- добавляет управляющие конструкции к стандарту SQL;
- допускает сложные вычисления;
- может использовать все объекты БД, определенные пользователем;
- прост в использовании.

Преимущества использования PL/pgSQL:

Стандартный SQL используется в PostgreSQL и других реляционных БД как основной язык для создания запросов. Он переносим и прост, как для изучения, так и для использования. Однако слабое его место — в том, что каждая конструкция языка выполняется сервером отдельно. Это значит, что клиентское приложение должно отправлять каждый запрос серверу, получить его результат, определенным образом согласно логике приложения, обработать его, посылать следующий запрос и т. д. В случае, если клиент и сервер БД расположены на разных машинах, это может привести к нежелательному увеличению задержек и объема пересылаемых от клиента серверу и наоборот данных.

При использовании PL/pgSQL все становится проще. Появляется возможность сгруппировать запросы и вычислительные блоки в единую конструкцию, которая будет размещаться и выполняться на сервере, а клиент будет отправлять запрос на её выполнение и получать результат, минуя все промежуточные пересылки данных назад—вперед, что в большинстве случаев очень позитивно сказывается на производительности.

Так же существует функциональность анонимных блоков, позволяющий писать запросы не на SQL, а прямо на любом существующем процедурном языке сервера, в том числе pl/pgSQL, без создания хранимых функций на сервере СУБД.

Триггерные функции

В PL/pgSQL можно создавать триггерные функции, которые будут вызываться при изменениях данных или событиях в базе данных. Триггерная функция создаётся командой CREATE FUNCTION, при этом у функции не должно быть аргументов, а типом возвращаемого значения должен быть trigger (для триггеров, срабатывающих при изменениях данных) или event_trigger (для триггеров, срабатывающих при событиях в базе). Для триггеров автоматически определяются специальные локальные переменные с именами вида TG_имя, описывающие условие, повлёкшее вызов триггера.

Триггеры при изменении данных

Триггер (англ. trigger) — хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено действием по модификации данных: добавлением INSERT, удалением DELETE строки в заданной таблице, или изменением UPDATE данных в определённом столбце заданной таблицы реляционной базы данных. Триггеры применяются для обеспечения целостности данных и реализации сложной бизнес-логики. Триггер запускается сервером автоматически при попытке изменения данных в таблице, с которой он связан. Все производимые им модификации данных рассматриваются как выполняемые в транзакции, в которой выполнено действие, вызвавшее срабатывание триггера. Соответственно, в случае обнаружения ошибки или нарушения целостности данных может произойти откат этой транзакции.

Триггер при изменении данных объявляется как функция без аргументов и с типом результата trigger. Заметьте, что эта функция должна объявляться без аргументов, даже если ожидается, что она будет получать аргументы, заданные в команде CREATE TRIGGER — такие аргументы передаются через TG_ARGV, как описано ниже.

Когда функция на PL/pgSQL срабатывает как триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

NEW

Тип данных RECORD. Переменная содержит новую строку базы данных для команд INSERT/UPDATE в триггерах уровня строки. В триггерах уровня оператора и для команды DELETE эта переменная имеет значение null.

OLD

Тип данных RECORD. Переменная содержит старую строку базы данных для команд UPDATE/DELETE в триггерах уровня строки. В триггерах уровня оператора и для команды INSERT эта переменная имеет значение null.

TG_NAME

Тип данных name. Переменная содержит имя сработавшего триггера.

TG_WHEN

Тип данных text. Строка, содержащая BEFORE, AFTER или INSTEAD OF, в зависимости от определения триггера.

TG_LEVEL

Тип данных text. Строка, содержащая ROW или STATEMENT, в зависимости от определения триггера.

TG_OP

Тип данных text.
Строка, содержащая INSERT, UPDATE, DELETE или TRUNCATE, в зависимости от того, для какой операции сработал триггер.

TG_RELID

Тип данных oid. OID таблицы, для которой сработал триггер.

TG_RELNAME

Тип данных name. Имя таблицы, для которой сработал триггер. Эта переменная устарела и может стать недоступной в будущих релизах. Вместо неё нужно использовать TG_TABLE_NAME.

Триггерная функция должна вернуть либо NULL, либо запись/строку, соответствующую структуре таблице, для которой сработал триггер.

Возвращаемое значение для строчного триггера AFTER и триггеров уровня оператора (BEFORE или AFTER) всегда игнорируется. Это может быть и NULL.

Однако, в этих триггерах по-прежнему можно прервать вызвавшую их команду, для этого нужно явно вызвать ошибку.

Триггерная функция на PL/pgSQL

Триггер, показанный в этом примере, при любом добавлении или изменении строки в таблице сохраняет в этой строке информацию о текущем пользователе и отметку времени. Кроме того, он требует, чтобы было указано имя сотрудника, и зарплата задавалась положительным числом.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Проверить, что указаны имя сотрудника и зарплата
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Кто будет работать, если за это надо будет платить?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;
        -- Запомнить, кто и когда изменил запись
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

ТРИГГЕРЫ СОБЫТИЙ

Три́ггер (англ. trigger) — хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено действием по модификации данных: добавлением INSERT, удалением DELETE строки в заданной таблице, или изменением UPDATE данных в определённом столбце заданной таблицы реляционной базы данных.

В PL/pgSQL можно создавать событийные триггеры. Postgres Pro требует, чтобы функция, которая вызывается как событийный триггер, объявлялась без аргументов и типом возвращаемого значения был event_trigger.

Когда функция на PL/pgSQL вызывается как событийный триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

TG_EVENT

Тип данных text. Строка, содержащая событие, для которого сработал триггер.

TG_TAG

Тип данных text. Переменная, содержащая тег команды, для которой сработал триггер.

Функция событийного триггера на **PL/pgSQL**

Триггер в этом примере просто выдаёт сообщение NOTICE каждый раз, когда выполняется поддерживаемая команда.

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE FUNCTION snitch();
```

Процедуры

Процедура представляет собой объект базы данных, подобный функции. Отличие состоит в том, что процедура не возвращает значение, и поэтому для неё не определяется возвращаемый тип. Тогда как функция вызывается в составе запроса или команды DML, процедура вызывается явно, оператором CALL.

CREATE PROCEDURE — создать процедуру

Синтаксис:

```
CREATE [ OR REPLACE ] PROCEDURE
    имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ { DEFAULT |
= } выражение_по_умолчанию ] [, ...] ] )
  { LANGUAGE имя_языка
    | TRANSFORM { FOR TYPE имя_типа } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET параметр_конфигурации { TO значение | = значение | FROM CURRENT }
    | AS 'определение'
    | AS 'объектный_файл', 'объектный_символ'
  } ...
```

Команда CREATE PROCEDURE определяет новую процедуру. CREATE OR REPLACE PROCEDURE создаёт новую процедуру либо заменяет определение уже существующей. Чтобы определить процедуру, необходимо иметь право USAGE для соответствующего языка.

Если указано имя схемы, процедура создаётся в заданной схеме, в противном случае — в текущей. Имя новой процедуры должно отличаться от имён существующих процедур и функций с такими же типами аргументов в этой схеме. Однако процедуры и функции с аргументами разных типов могут иметь одно имя (это называется перегрузкой).

Команда CREATE OR REPLACE PROCEDURE предназначена для изменения текущего определения существующей процедуры. С её помощью нельзя изменить имя или типы аргументов (если попытаться сделать это, будет создана новая отдельная процедура).

Когда команда CREATE OR REPLACE PROCEDURE заменяет существующую процедуру, владелец и права доступа к этой процедуре не меняются. Все другие свойства процедуры получают значения, задаваемые командой явно или по умолчанию. Чтобы заменить процедуру, необходимо быть её владельцем (или быть членом роли-владельца).

Владельцем процедуры становится создавший её пользователь.

Чтобы создать процедуру, необходимо иметь право USAGE для типов её аргументов.

Параметры:

имя

Имя создаваемой процедуры

режим_аргумента

Режим аргумента: IN, INOUT или VARIADIC. По умолчанию подразумевается IN. (Режим OUT для процедур в настоящее время не поддерживается. Используйте вместо него INOUT.)

имя_аргумента

Имя аргумента.

тип_аргумента

Тип данных аргумента процедуры (возможно, дополненный схемой), при наличии аргументов. Тип аргументов может быть базовым, составным или доменным, либо это может быть ссылка на столбец таблицы.

В зависимости от языка реализации также может допускаться указание «псевдотипов», например, cstring. Псевдотипы показывают, что фактический тип аргумента либо определён не полностью, либо существует вне множества обычных типов SQL.

Ссылка на тип столбца записывается в виде имя_таблицы.имя_столбца %TYPE. Иногда такое указание бывает полезно, так как позволяет создать процедуру, независящую от изменений в определении таблицы.

выражение_по_умолчанию

Выражение, используемое для вычисления значения по умолчанию, если параметр не задан явно. Результат выражения должен сводиться к типу соответствующего параметра. Для всех входных параметров, следующих за параметром с определённым значением по умолчанию, также должны быть определены значения по умолчанию.

имя_языка

Имя языка, на котором реализована функция. Это может быть sql, c, internal либо имя процедурного языка, определённого пользователем, например, plpgsql. Стиль написания этого имени в апострофах считается устаревшим и требует точного совпадения регистра.

определение

Строковая константа, определяющая реализацию процедуры; её значение зависит от языка. Это может быть имя внутренней процедуры, путь к объектному файлу, команда SQL или код на процедурном языке.

объектный_файл, объектный_символ

Эта форма предложения AS применяется для динамически загружаемых процедур на языке C, когда имя процедуры в коде C не совпадает с именем процедуры в SQL. Строка объектный_файл задаёт имя файла, содержащего скомпилированную процедуру на C (данная команда воспринимает эту строку так же, как и LOAD).

Вывод: в данном лабораторном практикуме были рассмотрены триггеры и пользовательские функции

Контрольные вопросы:

1 Что такое PL/pgSQL?

2 Зачем нужны триггеры?

3 Что такое процедура в бд?

Задания для выполнения:

1 Создать триггерную функцию на PL/pgSQL.

2 Создать процедуру с тремя параметрами.

БрГТУ.150089-06 95 00

Методическое пособие № 7 по теме:

«Установка и первичная настройка Percona XtraDB Cluster»

Цель: Установка и первичная настройка Percona XtraDB Cluster.

Содержание:

Описание:

Кластер – это специализированный объект базы данных, используемый для физически совместного хранения одной или нескольких таблиц, которые часто соединяются вместе в SQL-запросах. Кластеры хранят взаимосвязанные строки разных таблиц вместе в одних и тех же блоках данных, что позволяет сократить количество операций дискового ввода-вывода и улучшить время доступа для соединений таблиц, входящих в кластер.

Percona XtraDB Cluster - кластерная СУБД, предоставляющая решение для создания кластеров с синхронной репликаций между узлами, работающими в режиме multi-master. Percona XtraDB Cluster обеспечивает высокую производительность, быстрое восстановление узла кластера после падения и полный контроль состояния кластера. Исходные тексты проекта распространяются под лицензией GPLv2.

Основные преимущества:

- Синхронная репликация. Транзакция проходит либо на всех узлах, либо ни на каком.
- Режим multi-master. Писать данные можно на любой узел.
- Параллельная репликация.
- Высокая консистентность данных.
- Полная совместимость с базами данных MySQL.
- Полная совместимость с приложениями, работающими с MySQL.
- Балансировщик нагрузки ProxySQL.
- Легко настраиваемое зашифрованное общение между узлами.
- Высокая масштабируемость.

Общая схема работы:

Кластер состоит из узлов. Рекомендуется использовать хотя бы 3 узла. В каждом узле находится обычный сервер MySQL или Precona Server. Таким образом, уже существующий сервер MySQL или Precona Server можно преобразовать в узел и добавить в кластер, и наоборот: любой узел можно отсоединить от кластера и использовать как обычный сервер. В каждом узле находится полная копия данных.

Схема кластера показана на рисунке 1:

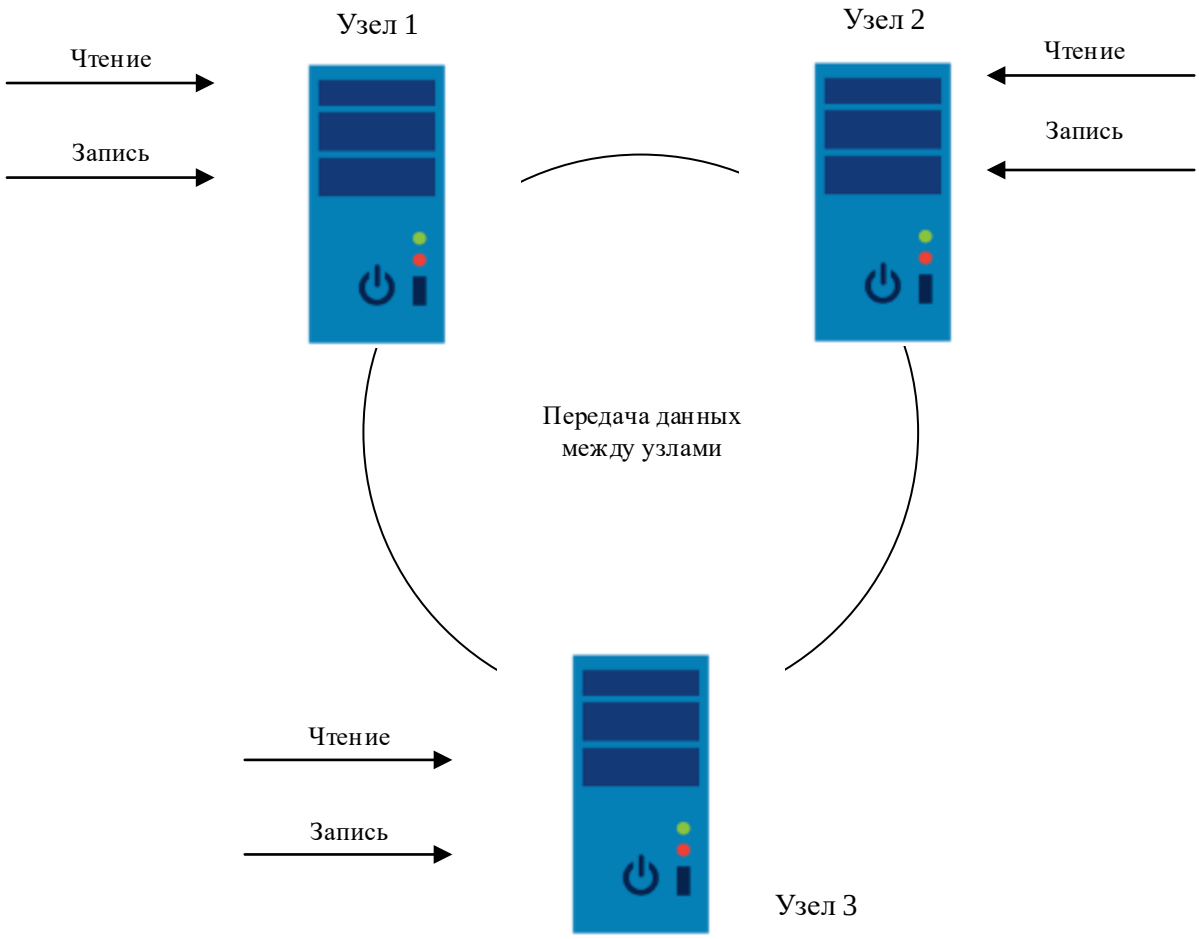

Рисунок 1 – схема кластера.

В кластере 3 узла, используется прозрачный коннект к одному виртуальному IP-адресу, разделяемому между всеми 3 серверами при помощи keepalived. Для балансировки нагрузки на узел используется HAProxy, конечно же установленный на каждом сервере, чтобы в случае отказа одного из них, нагрузку благодаря VIP продолжал балансировать другой. Узел А используется в качестве Reference (Backup) Node, которую не будут загружать запросами наши приложения. При этом она будет полноправным членом кластера, и участвовать в репликации. Это делается в связи с тем, что в случае сбоя в кластере или нарушения целостности

данных мы будем иметь узел, который почти наверняка содержит консистентные данные, которые приложения просто не могли порушить из-за отсутствия доступа. Возможно, это выглядит расточительным расходованием ресурсов, но 99% надежность данных все же важнее чем доступность 24/7. Именно этот узел мы будем использовать для SST — State Snapshot Transfer —Кроме того, Node A — отличный кандидат для сервера, откуда можно будем снимать стандартные периодические резервные копии.

Узел B и Узел C держат нагрузку на себе, при этом все операции записи происходят на одном из узлов.

Особенности такой схемы:

Преимущества:

- При исполнении запроса, он исполняется локально на узле. Все данные можно найти на локальной машине, без необходимости удаленного доступа.
- Отсутствует централизация. Любой узел можно отсоединить в любой момент, и кластер продолжит работать.
- Отличное решения для большого количества запросов на чтение. Их можно направлять к любому узлу.

Недостатки:

- Присоединение нового узла требует значительных ресурсов. Новый узел должен получить полную копию данных базы от одного из существующих узлов. Если база составляет 100 GB, то придется копировать все 100 GB.
- Неоптимальное масштабирование для большого количества запросов записи. Есть возможность увеличить производительность за счет направления трафика на несколько узлов, но суть проблемы от этого не меняется - все записи должны попадать на все узлы.

Виды синхронизации:

- Инкрементальный перенос состояния – это функциональность, которая вместо всего снимка состояния получить отсутствующие наборы данных, но только если эти данные находятся в кэше.
- Перенос снимка состояния – это полная копия данных с одного узла передаётся на другой узел. Используется, когда новый узел присоединяется к кластеру и должен передавать данные из существующего узла.

Первичная настройка кластера

Создадим Percona XtraDB Cluster с трех серверов под управлением операционной системы Ubuntu.

- Node 1
  - Host name: pxc1
  - IP address: 192.168.70.61
- Node 2
  - Host name: pxc2
  - IP address: 192.168.70.62
- Node 3
  - Host name: pxc3
  - IP address: 192.168.70.63

При установке Debian / Ubuntu запрашивается пароль root. После установки пакетов mysqld запустится автоматически. Остановим mysqld на всех трех узлах, используя следующую команду:

```
/etc/init.d/mysql stop
```

Следует убедиться, что файл конфигурации расположенный по адресу: /etc/mysql/my.cnf для первого узла (pxc1) содержит в себе следующее:

```
[mysqld]
datadir=/var/lib/mysql
user=mysql
# Path to Galera library
wsrep_provider=/usr/lib/libgalera_smm.so
# Cluster connection URL contains the IPs of node#1, node#2 and node#3
wsrep_cluster_address=gcomm://192.168.70.61,192.168.70.62,192.168.70.63
# In order for Galera to work correctly binlog format should be ROW
binlog_format=ROW
# MyISAM storage engine has only experimental support
default_storage_engine=InnoDB
# This InnoDB autoincrement locking mode is a requirement for Galera
innodb_autoinc_lock_mode=2
# Node #1 address
```

```
wsrep_node_address=192.168.70.61
# SST method
wsrep_sst_method=xtrabackup-v2
# Cluster name
wsrep_cluster_name=my_ubuntu_cluster
# Authentication for SST method
wsrep_sst_auth="sstuser:s3cretPass"
```

Далее надо запустить первый узел с помощью следующей команды:

```
[root@pxc1 ~]# /etc/init.d/mysql bootstrap-pxc
```

Эта команда запустит первый узел и загрузит кластер.

После запуска первого узла можно проверить состояние кластера с помощью следующей команды:

```
mysql> show status like 'wsrep%';
+----------------------------+--------------------------------------+
| Variable_name              | Value                                |
+----------------------------+--------------------------------------+
| wsrep_local_state_uuid     | b598af3e-ace3-11e2-0800-3e90eb9cd5d3 |
...
| wsrep_local_state          | 4                                    |
| wsrep_local_state_comment  | Synced                               |
...
| wsrep_cluster_size         | 1                                    |
| wsrep_cluster_status       | Primary                              |
| wsrep_connected            | ON                                   |
...
| wsrep_ready                | ON                                   |
+----------------------------+--------------------------------------+
40 rows in set (0.01 sec)
```

Эти выходные данные показывают, что кластер был успешно загружен.

Создадим нового пользователя со следующими привилегиями:

```
mysql@pxc1> CREATE USER 'sstuser'@'localhost' IDENTIFIED BY 's3cretPass';
mysql@pxc1> GRANT PROCESS, RELOAD, LOCK TABLES, REPLICATION CLIENT ON *.* TO
'sstuser'@'localhost';
mysql@pxc1> FLUSH PRIVILEGES;
```

Далее настроим второй узел. Файл /etc/mysql/my.cnf для второго узла (pxc2) должен содержать в себе следующее:

```
[mysqld]
datadir=/var/lib/mysql
user=mysql
# Path to Galera library
wsrep_provider=/usr/lib/libgalera_smm.so
connection URL contains IPs of node#1, node#2 and node#3
wsrep_cluster_address=gcomm://192.168.70.61,192.168.70.62,192.168.70.63
# In order for Galera to work correctly binlog format should be ROW
binlog_format=ROW
# MyISAM storage engine has only experimental support
default_storage_engine=InnoDB
# This InnoDB autoincrement locking mode is a requirement for Galera
innodb_autoinc_lock_mode=2
# Node #2 address
wsrep_node_address=192.168.70.62
# Cluster name
wsrep_cluster_name=my_ubuntu_cluster
# SST method
wsrep_sst_method=xtrabackup-v2
#Authentication for SST method
wsrep_sst_auth="sstuser:s3cretPass"
```

Запустим второй узел при помощи команды:

```
[root@pxc2 ~]# /etc/init.d/mysql start
```

После запуска сервера он должен автоматически получать SST (Передача моментального снимка состояния - это полная копия данных с одного узла на другой.). Состояние кластера теперь можно проверить на обоих узлах. Ниже приведен пример состояния со второго узла (pxc2):

```
mysql> show status like 'wsrep%';
+----------------------------+--------------------------------------+
| Variable_name              | Value                                |
+----------------------------+--------------------------------------+
| wsrep_local_state_uuid     | b598af3e-ace3-11e2-0800-3e90eb9cd5d3 |
...

| wsrep_local_state          | 4                                    |
| wsrep_local_state_comment  | Synced                               |
...
| wsrep_cluster_size         | 2                                    |
| wsrep_cluster_status       | Primary                              |
| wsrep_connected            | ON                                   |
...
| wsrep_ready                | ON                                   |
+----------------------------+--------------------------------------+
40 rows in set (0.01 sec)
```

Эти выходные данные показывают, что новый узел был успешно добавлен в кластер.

Настроим третий узел. Файл /etc/mysql/my.cnf для третьего узла (pxc3) должен содержать в себе следующее:

```
 [mysqld]
datadir=/var/lib/mysql
user=mysql
# Path to Galera library
wsrep_provider=/usr/lib/libgalera_smm.so
# Cluster connection URL contains IPs of node#1, node#2 and node#3
wsrep_cluster_address=gcomm://192.168.70.61,192.168.70.62,192.168.70.63
# In order for Galera to work correctly binlog format should be ROW
binlog_format=ROW
# MyISAM storage engine has only experimental support
default_storage_engine=InnoDB

# This InnoDB autoincrement locking mode is a requirement for Galera
innodb_autoinc_lock_mode=2
# Node #3 address
```

```
wsrep_node_address=192.168.70.63
# Cluster name
wsrep_cluster_name=my_ubuntu_cluster
# SST method
wsrep_sst_method=xtrabackup-v2
#Authentication for SST method
wsrep_sst_auth="sstuser:s3cretPass"
```

Запустим третий узел при помощи команды:

```
[root@pxc3 ~]# /etc/init.d/mysql start
```

После запуска сервера он должен автоматически получать SST. Состояние кластера можно проверить на всех узлах. Ниже приведен пример состояния от третьего узла (pxc3):

```
mysql> show status like 'wsrep%';
+----------------------------+--------------------------------------+
| Variable_name              | Value                                |
+----------------------------+--------------------------------------+
| wsrep_local_state_uuid     | b598af3e-ace3-11e2-0800-3e90eb9cd5d3 |
| wsrep_local_state          | 4                                    |
| wsrep_local_state_comment  | Synced                               |
...
| wsrep_cluster_size         | 3                                    |
| wsrep_cluster_status       | Primary                              |
| wsrep_connected            | ON                                   |
...
| wsrep_ready                | ON                                   |
+----------------------------+--------------------------------------+
40 rows in set (0.01 sec)
```

Тестирование репликации:

Чтобы проверить репликацию, создадим новую базу данных на втором узле, создадим таблицу для этой базы данных на третьем узле и добавим несколько записей в таблицу на первом узле.

Создадим новую базу данных на втором узле:

```
mysql@pxc2> CREATE DATABASE percona;
```

```
Query OK, 1 row affected (0.01 sec)
```

Создадим таблицу на третьем узле:

```
mysql@pxc3> USE percona;

Database changed

mysql@pxc3> CREATE TABLE example (node_id INT PRIMARY KEY, node_name VARCHAR(
30));

Query OK, 0 rows affected (0.05 sec)
```

Вставление записи в первый узел:

```
mysql@pxc1> INSERT INTO percona.example VALUES (1, 'percona1');

Query OK, 1 row affected (0.02 sec)
```

Теперь получим все строки из данной таблицы на втором узле при помощи команды:

```
mysql@pxc2> SELECT * FROM percona.example;

+---------+-----------+

| node_id | node_name |

+---------+-----------+

|       1 | percona1  |

+---------+-----------+

1 row in set (0.00 sec)
```

Эта процедура должна гарантировать, что все узлы в кластере синхронизированы и работают как и было задуманно.

Вывод: В данном лабораторном практикуме рассмотрен кластер для управления бд - Percona XtraDB Cluster. Рассмотрена его установка и первичная настройка.

Контрольные вопросы:

1. Что такое кластер?
2. Для чего нужен кластер?
3. Какие виды синхронизации существуют?

Задание для выполнения:

1 Установить Percona XtraDB Cluster, запустить кластер, настроить три узла кластера, провести тестовую репликацию.