DIASOFT
всё по-настоящему

Q.DATABASE, УПРАВЛЕНИЕ БД (СИСТЕМА УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ POSTGRESDB)

Руководство пользователя

Диасофт

2022

Содержание

1. Общие положения

Компонент «Система управления базами данных PostgreDB» – объектно-реляционная система управления базами данных (далее по документу СУБД), разработанная с использованием свободно распространяемой СУБД PostgreSQL. СУБД PostgreDB предназначена для работы с различными системами, а также для работы с банковской линейкой, и обеспечивает значительное повышение производительности.

Система поддерживает базовые функциональные возможности PostgreSQL, а также следующие уникальные возможности:

- Загрузка данных, полученные экспортом из MSSQL при помощи утилиты BCP;
- 64-битный счётчик транзакций;
- Управление планом выполнения запросов с помощью указаний, записываемых в виде простых описаний в SQL-комментариях особого вида;
- Пулер соединений, позволяющий подключиться к СУБД большому числу клиентов без деградации производительности;
- Внешние расширения (модули):
    - nt5srvcall, модуль, выполняющий HTTP-запросы во внешние сервисы;
    - nt5autrans, модуль с возможностью запуска автономных транзакций;
    - nt5auvars, модуль для работы с именованными переменными в общей памяти сервера;
    - nt5utf16, модуль для перекодировки хеш-сумм из формата utf8 в utf16 и обратно.

Настоящий документ содержит описание работы и реализует основной функционал системы.

2. Термины и сокращения

Термин/Сокращение	Определение
БД	База данных
СУБД PostgreDB	Система управления базами данных PostgreDB

3. Начало работы

3.1 Подключение к серверу БД

Для проверки подключения к серверу, от имени суперпользователя в терминале ввести в команду:

sudo -u postgres pg_ctl -D /var/lib/pgsql/pgdata -l /var/lib/pgsql/pgdata/logfile start

В результате выполнения на экран выводится сообщение

 # waiting for server to start.... done

 # server started

Это означает, что сервер запущен.
Проверить соединение с сервером БД можно с помощью команды:

$ pg_isready

Если в результате выполнения на экране появляется сообщение

/tmp:5432 - accepting connections.

Следовательно, подключение произошло успешно, и соединение установлено.

3.2 Psql

Программный компонент psql — это терминальный клиент для работы с PostgreDB. Он позволяет интерактивно вводить запросы, передавать их в PostgreDB и видеть результаты. Также запросы могут быть получены из файла или из аргументов командной строки. Кроме того, psql предоставляет ряд метакоманд и различные возможности, подобные тем, что имеются у командных оболочек, для облегчения написания скриптов и автоматизации широкого спектра задач.

Перейти в консольное приложение psql и зайти под пользователем **postges** можно с помощью команды:

sudo -u postgres psql

3.2.1 Создание БД и подключение к ней

Создав базу данных, можно обратиться к ней:

- Запустив терминальную программу под названием *psql*, в которой можно интерактивно вводить, редактировать и выполнять команды SQL.
- Используя существующие графические инструменты, например, pgAdmin или офисный пакет с поддержкой ODBC или JDBC, позволяющий создавать и управлять базой данных.
- Написав собственное приложение, используя один из множества доступных языковых интерфейсов.

Для создания базы данных **testdb**, необходимо выполнить следующую команду:

$ **createdb testdb**

Если нет никаких сообщений об ошибках, значит операция была выполнена успешно. Если больше нет необходимости использовать базу данных, можно удалить её, выполнив следующую команду:

$ **dropdb testdb**

В результате будут физически удалены все файлы, связанные с базой данных.

Подключиться с помощью psql к базе данных **testdb** можно, введя команду:

$ **psql testdb**

Если имя базы данных не указать, она будет выбрана по имени пользователя.

Как правило, приглашение psql состоит из имени базы данных, к которой psql в данный момент подключён, а затем строки =>. Например:

$ **psql testdb**
psql (14.3)
Type "help" for help.

testdb=>

Последняя строка может выглядеть и так:

testdb =#

Что показывает, что пользователь является суперпользователем, в этом случае не будут распространяться никакие ограничения доступа.

Последняя строка в выводе psql — это приглашение, которое показывает, что psql ждёт ваших команд и вы можете вводить SQL-запросы в рабочей среде psql. Для проверки работы попробуйте команду:

testdb=> **SELECT 1 + 1;**
 ?column?

        2
(1 row)

В программе psql есть множество внутренних команд, которые не являются SQL-операторами. Они начинаются с обратной косой черты, «\». Чтобы выйти из psql, необходимо ввести:

testdb=> **\q**

и psql завершит свою работу, а пользователя вернет в командную оболочку операционной системы.

В командной строке пользователь может вводить команды SQL. Обычно введённые строки отправляются на сервер, когда встречается точка с запятой, завершающая команду. Конец строки не завершает команду. Это позволяет разбивать команды на несколько строк для лучшего понимания. Если команда была отправлена и выполнена без ошибок, то результат команды выводится на экран.

3.3 Описание объектов БД

3.3.1 Создание таблиц

Для того, чтобы создать таблицу, необходимо указать её имя, перечислив все имена столбцов и их типы:

CREATE TABLE *table_name* (
column1 ***datatype,***
column2 ***datatype,***
column3 ***datatype,***
…
);

PostgreDB поддерживает стандартные типы SQL: **int, smallint, real, double precision, char(*N*), varchar(*N*), date, time, timestamp и interval**, а также другие универсальные типы и набор геометрических типов. Кроме того, PostgreDB можно расширять, создавая набор собственных типов данных. Как следствие, имена типов не являются ключевыми словами в данной записи, кроме тех случаев, когда это требуется для реализации особых конструкций стандарта SQL.

Если какая-либо таблица больше не нужна, или необходимо пересоздать её по-другому, ее можно удалить, используя следующую команду:

DROP TABLE *table_name*;

3.3.2 Создание индексов

Индексы — это средство увеличения производительности БД. Используя индекс, сервер баз данных может находить и извлекать нужные строки гораздо быстрее, чем без него. Однако с индексами связана дополнительная нагрузка на СУБД в целом, поэтому применять их следует обдуманно. Обновление таблицы с индексами занимает больше времени, чем обновление таблицы без индексов (поскольку индексы также нуждаются в обновлении). Поэтому следует создавать индексы только для тех столбцов, по которым будет часто выполняться поиск.

Синтаксис создания индекса для таблицы. Допускаются повторяющиеся значения:

CREATE INDEX *index_name*
ON *table_name* (*column1, column2, ...*);

Не допускаются повторяющиеся значения:

CREATE UNIQUE INDEX *index_name*
ON *table_name* (*column1, column2, ...*);

Для удаления индекса используется команда **DROP INDEX**. Добавлять и удалять индексы можно в любое время.

Когда индекс создан, никакие дополнительные действия не требуются: система сама будет обновлять его при изменении данных в таблице и сама будет использовать его в запросах, где, по её мнению, это будет эффективнее, чем сканирование всей таблицы. Индексы могут быть полезны также при выполнении команд **UPDATE** и **DELETE** с условиями поиска. Кроме того, они могут применяться в поиске с соединением. То есть, индекс, определённый для столбца, участвующего в условии соединения, может значительно ускорить запросы с **JOIN.**

Создание индекса для большой таблицы может занимать много времени. По умолчанию PostgreDB позволяет параллельно с созданием индекса выполнять чтение (операторы **SELECT**) таблицы, но операции записи (**INSERT, UPDATE и DELETE)** блокируются до окончания построения индекса. Для производственной среды это ограничение часто бывает неприемлемым.

PostgreDB поддерживает несколько типов индексов: B-дерево, хеш, GiST, SP-GiST, GIN и BRIN. Для разных типов индексов применяются разные алгоритмы, ориентированные на определённые типы запросов. По умолчанию команда **CREATE INDEX** создаёт индексы-B-деревья, эффективные в большинстве случаев. Выбрать другой тип можно, написав название типа индекса после ключевого слова **USING**. Например, создать хеш-индекс можно так:

CREATE INDEX *index_name* ON *table_name* USING HASH (*column1*);

3.3.3 Внешние ключи

Для связи между таблицами применяются внешние ключи. Внешний ключ устанавливается для столбца из зависимой, подчиненной таблицы (*referencing table*), и указывает на один из столбцов из главной таблицы (*referenced table*). Как правило, внешний ключ указывает на первичный ключ из связанной главной таблицы.

Общий синтаксис установки внешнего ключа на уровне столбца:

REFERENCES *main_table* (*column_main_table*)
[ON DELETE {CASCADE|RESTRICT}]
[ON UPDATE {CASCADE|RESTRICT}]

Чтобы установить связь между таблицами, после ключевого слова **REFERENCES** указывается имя связанной таблицы и далее в скобках имя столбца из этой таблицы, на который будет указывать внешний ключ. После выражения **REFERENCES** может идти выражение **ON DELETE** и **ON UPDATE**, которые уточняют поведение при удалении или обновлении данных. - Общий синтаксис установки внешнего ключа на уровне таблицы:

FOREIGN KEY (column1, column2, column3, …)

REFERENCES *main_table* (*column1_main_table, column2_main_table, column3_main_table,* …)

[ON DELETE {CASCADE|RESTRICT}]

[ON UPDATE {CASCADE|RESTRICT}]

4. Модификация данных

4.1 Добавление данных

Сразу после создания таблицы она не содержит никаких данных. Поэтому, чтобы она была полезна, в неё прежде всего нужно добавить данные. Необходимо указать имена столбцов и значения для вставки. Синтаксис команды:

INSERT INTO *table_name* (*column1, column2, column3...*)
VALUES (*value1, value2, value3, ...*);

Если добавляются значения для всех столбцов таблицы, не нужно указывать имена столбцов в SQL-запросе. Однако порядок значений должен соответствовать порядку столбцов в таблице. Здесь синтаксис **INSERT INTO** будет следующим:

INSERT INTO *table_name*
VALUES (*value1, value2, value3, ...*);

Значения данных перечисляются в порядке столбцов в таблице и разделяются запятыми. Одна команда может вставить сразу несколько строк:

INSERT INTO *table_name* (*column1, column2, column3...*) VALUES
(*value1, value2, value3, ...*),
(*value4, value5, value6, ...*),
(*value7, value8, value9, ...*);

Также возможно вставить результат запроса (который может не содержать строк либо содержать одну или несколько):

INSERT INTO *table_name* (*column1, column2, column3...*)
SELECT *value1, value2, value3* FROM *table_name2*

WHERE *condition*;

4.2 Изменение данных

Модификация данных, уже сохранённых в БД, называется изменением. Изменить можно все строки таблицы, либо подмножество всех строк, либо только избранные строки. Каждый столбец при этом можно изменять независимо от других.

Для изменения данных в существующих строках используется команда **UPDATE**. Ей требуется следующая информация:

1. Имя таблицы и изменяемого столбца
2. Новое значение столбца
3. Критерий отбора изменяемых строк

Синтаксис команды будет следующим:

UPDATE *table_name*
SET *column1* = *value1*, *column2* = *value2*,...
WHERE *condition*;

Необходимо указать условия, каким должны соответствовать требуемая строка. Только если в таблице есть первичный ключ (вне зависимости от того, объявляли вы его или нет), можно однозначно адресовать отдельные строки, определив условие по первичному ключу.

Предложение **WHERE** указывает, какие записи необходимо обновить. Если опустить предложение **WHERE**, все записи в таблице будут обновлены.

4.3 Выборка данных

Чтобы получить данные из таблицы, нужно выполнить *запрос*. Для этого предназначен SQL-оператор **SELECT**. Он состоит из нескольких частей: выборки (в которой перечисляются столбцы, которые должны быть получены), списка таблиц (в нём перечисляются таблицы, из которых будут получены данные) и необязательного условия (определяющего ограничения). Возвращенные данные сохраняются в таблице результатов, называемой набором результатов.

Синтаксис команды следующий:

SELECT *column1, column2,...*
FROM *table_name*;

Здесь **column1, column2,** ... это имена полей таблицы, из которой вы хотите выбрать данные. Если необходимо выбрать все поля, доступные в таблице, используйте следующий синтаксис:

SELECT * FROM *table_name*;

Здесь ***** — это краткое обозначение «всех столбцов».

Запрос можно дополнить «условием», добавив предложение **WHERE**, ограничивающее множество возвращаемых строк. В предложении **WHERE** указывается логическое выражение (проверка истинности), которое служит фильтром строк: в результате оказываются только те строки, для которых это выражение истинно. Синтаксис в данном случае будет следующим:

SELECT *column1, column2,...*
FROM *table_name*

***WHERE conditions*;**

Если требуется, можно убрать дублирующиеся строки из результата запроса:

SELECT DISTINCT *column1, column2,...*
FROM *table_name;*

4.4 Удаление данных

Строки также можно удалить из таблицы, используя команду DELETE. Так же, как добавлять данные можно только целыми строками, удалять их можно только по строкам. Удалить избранные строки можно, только сформулировав для них подходящие условия. Но если в таблице есть первичный ключ, с его помощью можно однозначно выделить определённую строку. При этом можно так же удалить группы строк, соответствующие условию, либо сразу все строки таблицы.

Для удаления строк используется команда **DELETE**; её синтаксис очень похож на синтаксис команды **UPDATE**:

DELETE FROM *table_name* WHERE *condition*;

При этом нужно быть осторожными с операторами вида

DELETE FROM *имя_таблицы table_name*;

Без указания условия **DELETE** удалит ***все*** строки данной таблицы, полностью очистит её. При этом система не попросит вас подтвердить операцию.

5. Группировка и сортировка данных

5.1 Условие WHERE

Предложение **WHERE** используется для фильтрации записей. Оно используется для извлечения только тех записей, которые удовлетворяют заданному условию. Синтаксис следующий:

WHERE *condition*;
где ***condition*** — 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**.

Используется не только в операторах **SELECT**, но также в **UPDATE, DELETE** и т.д.

SELECT *column1, column2, ...*
FROM *table_name*
WHERE *condition*;

5.1.1 Логические операторы

Набор логических операторов включает обычные:

boolean AND boolean → boolean
boolean OR boolean → boolean
NOT boolean → boolean

В SQL работает логическая система с тремя состояниями: **TRUE** (истина), **FALSE** (ложь) и **NULL**, «неопределённое» состояние.

Предложение **WHERE** можно комбинировать с операторами **AND, OR** и **NOT**. Операторы **AND** и **OR** используются для фильтрации записей по нескольким условиям:

- Оператор **AND** отображает запись, если все условия, разделенные оператором **AND**, истинны.
- Оператор **OR** отображает запись, если какое-либо из условий, разделенных **OR**, истинно.
- Оператор **NOT** отображает запись, если условие(я) **НЕ** истинно.

Операторы **AND** и **OR** коммутативны, то есть от перемены мест операндов результат не меняется.

SELECT ***column1, column2,...***
FROM ***table_name***
WHERE ***condition1*** **AND** ***condition2*** **AND** ***condition3 ...;***

SELECT ***column1, column2,...***
FROM ***table_name***
WHERE ***condition1*** **OR** ***condition2*** **OR** ***condition3 ...;***

SELECT ***column1, column2,...***
FROM ***table_name***
WHERE NOT ***condition;***

5.2 Предложение GROUP BY

Строки порождённой входной таблицы, прошедшие фильтр **WHERE**, можно сгруппировать с помощью предложения **GROUP BY**.

SELECT ***column_name(s)***
FROM ***table_name***
WHERE ***condition***
GROUP BY ***column_name(s);***

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.

Оператор **GROUP BY** часто используется с агрегатными функциями **(COUNT(), MAX(), MIN(), SUM(), AVG())** для группировки набора результатов по одному или нескольким столбцам.

В группированной таблице столбцы, не включённые в список **GROUP BY**, можно использовать только в агрегатных выражениях. Пример такого агрегатного выражения:

=> **SELECT x, sum(y) FROM test1 GROUP BY x;**
 x | sum
---+-----
 a | 4
 b | 5
 c | 2
(3 rows)
Здесь **sum** — агрегатная функция, вычисляющая единственное значение для всей группы.

Группировка без агрегатных выражений по сути выдаёт набор различающихся значений столбцов. Этот же результат можно получить с помощью предложения **DISTINCT**.

В стандарте SQL **GROUP BY** может группировать только по столбцам исходной таблицы, но расширение PostgreDB позволяет использовать в **GROUP BY** столбцы из списка выборки. Также возможна группировка по выражениям, а не просто именам столбцов.

5.2.1 Агрегатные функции

Как большинство других серверов реляционных баз данных, PostgreDB поддерживает *агрегатные функции*. Агрегатная функция вычисляет единственное значение, обрабатывая множество строк. Они также очень полезны в сочетании с предложением **GROUP BY.**

Функция **MIN()** возвращает наименьшее значение выбранного столбца.

Функция **MAX()** возвращает наибольшее значение выбранного столбца.

SELECT MIN(*column_name*)
FROM *table_name*
WHERE *condition*;

SELECT MAX(*column_name*)
FROM *table_name*
WHERE *condition*;

Функция **COUNT()** возвращает количество строк, соответствующих заданному критерию.

SELECT COUNT(*column_name*)
FROM *table_name*
WHERE *condition*;

Функция **AVG()** возвращает среднее значение числового столбца.

SELECT AVG(*column_name*)
FROM *table_name*
WHERE *condition*;

Функция **SUM()** возвращает общую сумму числового столбца.

SELECT SUM(*column_name*)
FROM *table_name*
WHERE *condition*;

5.3 Предложение ORDER BY

После того как запрос выдал таблицу результатов (после обработки списка выборки), её можно отсортировать. Если сортировка не задана, строки возвращаются в неопределённом порядке. Фактический порядок строк в этом случае будет зависеть от плана соединения и сканирования, а также от порядка данных на диске, поэтому полагаться на него нельзя. Определённый порядок выводимых строк гарантируется, только если этап сортировки задан явно.

Порядок сортировки определяет предложение **ORDER BY**:

SELECT *column1, column2,...*
FROM *table_name*
ORDER BY *column1, column2, ...* ASC|DESC;

Ключевое слово **ORDER BY** по умолчанию сортирует записи в порядке возрастания. Чтобы отсортировать записи в порядке убывания, используйте ключевое слово **DESC**. Выражениями сортировки могут быть любые выражения, допустимые в списке выборки запроса. Например:

SELECT a, b FROM table1 ORDER BY a + b, c;

Когда указывается несколько выражений, последующие значения позволяют отсортировать строки, в которых совпали все предыдущие значения. Для определения места значений **NULL** можно использовать указания **NULLS FIRST** и **NULLS LAST**, которые помещают значения **NULL** соответственно до или после значений не **NULL**. По умолчанию значения **NULL** считаются больше любых других, то есть подразумевается **NULLS FIRST** для порядка **DESC** и **NULLS LAST** в противном случае.

6. Транзакции

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.

Говорят, что транзакция *атомарна*: с точки зрения других транзакций она либо выполняется и фиксируется полностью, либо не фиксируется совсем. Транзакционная база данных гарантирует, что все изменения записываются в постоянное хранилище до того, как транзакция будет считаться завершённой. Изменения, производимые открытой транзакцией, невидимы для других транзакций, пока она не будет завершена, а затем они становятся видны все сразу.

В PostgreDB транзакция определяется набором SQL-команд, окружённым командами **BEGIN** и **COMMIT**. Банковская транзакция может выглядеть так:

```
BEGIN;
UPDATE table_name SET column1 = column2 – 100.000
   WHERE conditions;
-- ...
COMMIT;
```

Если в процессе выполнения транзакции решается, что не нужно фиксировать её изменения, можно выполнить команду **ROLLBACK** вместо **COMMIT**, и все изменения будут отменены.

6.1 Изоляция транзакций

Стандарт SQL определяет четыре уровня изоляции транзакций. Наиболее строгий из них — сериализуемый, определяется одним абзацем, говорящем, что при параллельном выполнении несколько сериализуемых транзакций должны гарантированно выдавать такой же результат, как если бы они запускались по очереди в некотором порядке. Остальные три уровня определяются через описания особых явлений, которые возможны при взаимодействии параллельных

транзакций, но не допускаются на определённом уровне. Как отмечается в стандарте, из определения сериализуемого уровня вытекает, что на этом уровне ни одно из этих явлений невозможно.

Стандарт описывает следующие особые условия, недопустимые для различных уровней изоляции:

«Грязное» чтение

Транзакция читает данные, записанные параллельной незавершённой транзакцией.

Неповторяемое чтение

Транзакция повторно читает те же данные, что и раньше, и обнаруживает, что они были изменены другой транзакцией (которая завершилась после первого чтения).

Фантомное чтение

Транзакция повторно выполняет запрос, возвращающий набор строк для некоторого условия, и обнаруживает, что набор строк, удовлетворяющих условию, изменился из-за транзакции, завершившейся за это время.

Аномалия сериализации

Результат успешной фиксации группы транзакций оказывается несогласованным при всевозможных вариантах исполнения этих транзакций по очереди.

Уровень изоляции	**«Грязное» чтение**	**Неповторяемое чтение**	**Фантомное чтение**	**Аномалия сериализации**
Read uncommited (Чтение незафиксированных данных)	Допускается, но не в PostgreDB	Возможно	Возможно	Возможно
Read committed (Чтение зафиксированных данных)	Невозможно	Возможно	Возможно	Возможно

Repeatable read (Повторяемое чтение)	Невозможно	Невозможно	Допускается, но не в PostgreDB	Возможно
Serializable (Сериализуемость)	Невозможно	Невозможно	Невозможно	Невозможно

В PostgreDB можно запросить любой из четырёх уровней изоляции транзакций, однако внутри реализованы только три различных уровня, то есть режим Read Uncommitted в PostgreDB действует как Read Committed.

Для выбора нужного уровня изоляции транзакций используется команда SET TRANSACTION.

Read Committed — уровень изоляции транзакции, выбираемый в PostgreDB по умолчанию. В транзакции, работающей на этом уровне, запрос SELECT (без предложения FOR UPDATE/SHARE) видит только те данные, которые были зафиксированы до начала запроса, он никогда не увидит незафиксированных данных или изменений, внесённых в процессе выполнения запроса параллельными транзакциями. По сути запрос SELECT видит снимок базы данных в момент начала выполнения запроса. Однако SELECT видит результаты изменений, внесённых ранее в этой же транзакции, даже если они ещё не зафиксированы. Также заметьте, что два последовательных оператора SELECT могут видеть разные данные даже в рамках одной транзакции, если какие-то другие транзакции зафиксируют изменения после запуска первого SELECT, но до запуска второго.

Команды UPDATE, DELETE, SELECT FOR UPDATE и SELECT FOR SHARE ведут себя подобно SELECT при поиске целевых строк: они найдут только те целевые строки, которые были зафиксированы на момент начала команды. Однако к моменту, когда они будут найдены, эти целевые строки могут быть уже изменены (а также удалены или заблокированы) другой параллельной транзакцией. В этом случае запланированное изменение будет отложено до фиксирования или отката первой изменяющей данные транзакции (если она ещё выполняется). Если первая изменяющая транзакция откатывается, её результат отбрасывается, и вторая изменяющая транзакция может продолжить изменение изначально полученной строки. Если

первая транзакция зафиксировалась, но в результате удалила эту строку, вторая будет игнорировать её, а в противном случае попытается выполнить свою операцию с изменённой версией строки. Условие поиска в команде (предложение WHERE) вычисляется повторно для выяснения, соответствует ли по-прежнему этому условию изменённая версия строки. Если да, вторая изменяющая транзакция продолжают свою работу с изменённой версией строки. Применительно к командам SELECT FOR UPDATE и SELECT FOR SHARE это означает, что изменённая версия строки блокируется и возвращается клиенту.

Похожим образом ведёт себя INSERT с предложением ON CONFLICT DO UPDATE. В режиме Read Committed каждая строка, предлагаемая для добавления, будет либо вставлена, либо изменена. Если не возникнет несвязанных ошибок, гарантируется один из этих двух исходов. Если конфликт будет вызван другой транзакцией, результат которой ещё не видим для INSERT, предложение UPDATE подействует на эту строку, даже несмотря на то, что эта команда обычным образом может не видеть никакую версию этой строки.

При выполнении INSERT с предложением ON CONFLICT DO NOTHING строка может не добавиться в результате действия другой транзакции, эффект которой не виден в снимке команды INSERT. Это опять же имеет место только в режиме Read Committed.

Вследствие описанных выше правил, изменяющая команда может увидеть несогласованное состояние: она может видеть результаты параллельных команд, изменяющих те же строки, что пытается изменить она, но при этом она не видит результаты этих команд в других строках таблиц. Из-за этого поведения уровень Read Committed не подходит для команд со сложными условиями поиска; однако он вполне пригоден для простых случаев.

Частичная изоляция транзакций, обеспечиваемая в режиме Read Committed, приемлема для множества приложений. Этот режим быстр и прост в использовании, однако он подходит не для всех случаев. Приложениям, выполняющим сложные запросы и изменения, могут потребоваться более строго согласованное представление данных, чем то, что даёт Read Committed.

В режиме *Repeatable Read* видны только те данные, которые были зафиксированы до начала транзакции, но не видны незафиксированные данные и изменения, произведённые другими транзакциями в процессе выполнения данной транзакции. Как было сказано выше, это не

противоречит стандарту, так как он определяет только минимальную защиту, которая должна обеспечиваться на каждом уровне изоляции.

Этот уровень отличается от Read Committed тем, что запрос в транзакции данного уровня видит снимок данных на момент начала первого оператора в ***транзакции*** (не считая команд управления транзакциями), а не начала текущего оператора. Таким образом, последовательные команды SELECT в ***одной*** транзакции видят одни и те же данные; они не видят изменений, внесённых и зафиксированных другими транзакциями после начала их текущей транзакции.

Команды UPDATE, DELETE, SELECT FOR UPDATE и SELECT FOR SHARE ведут себя подобно SELECT при поиске целевых строк: они найдут только те целевые строки, которые были зафиксированы на момент начала транзакции. Однако к моменту, когда они будут найдены, эти целевые строки могут быть уже изменены (а также удалены или заблокированы) другой параллельной транзакцией. В этом случае транзакция в режиме Repeatable Read будет ожидать фиксирования или отката первой изменяющей данные транзакции (если она ещё выполняется). Если первая изменяющая транзакция откатывается, её результат отбрасывается и текущая транзакция может продолжить изменение изначально полученной строки. Если же первая транзакция зафиксировалась и в результате изменила или удалила эту строку, а не просто заблокировала её, произойдёт откат текущей транзакции с сообщением

ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения

так как транзакция уровня Repeatable Read не может изменять или блокировать строки, изменённые другими транзакциями с момента её начала.

Когда приложение получает это сообщение об ошибке, оно должна прервать текущую транзакцию и попытаться повторить её с самого начала. Во второй раз транзакция увидит внесённое до этого изменение как часть начального снимка базы данных, так что новая версия строки вполне может использоваться в качестве отправной точки для изменения в повторной транзакции. В транзакциях, которые только читают данные, конфликтов сериализации не бывает.

Режим Repeatable Read строго гарантирует, что каждая транзакция видит полностью стабильное представление базы данных. Однако это представление не обязательно будет согласовано с некоторым последовательным выполнением транзакций одного уровня. Например, даже

транзакция, которая только читает данные, в этом режиме может видеть строку, показывающую, что некоторое задание завершено, но ***не*** видеть одну из строк логических частей задания, так как эта транзакция может прочитать более раннюю версию строки задания, чем ту, для которой параллельно добавлялась очередная логическая часть. Строго исполнить бизнес-правила в транзакциях, работающих на этом уровне изоляции, скорее всего не удастся без явных блокировок конфликтующих транзакций.

Для реализации уровня изоляции Repeatable Read применяется подход, который называется в академической литературе по базам данных и в других СУБД *Изоляция снимков* (Snapshot Isolation). По сравнению с системами, использующими традиционный метод блокировок, затрудняющий параллельное выполнение, при этом подходе наблюдается другое поведение и другая производительность. В некоторых СУБД могут существовать даже два отдельных уровня Repeatable Read и Snapshot Isolation с различным поведением.

Уровень *Serializable* обеспечивает самую строгую изоляцию транзакций. На этом уровне моделируется последовательное выполнение всех зафиксированных транзакций, как если бы транзакции выполнялись одна за другой, последовательно, а не параллельно. Однако, как и на уровне Repeatable Read, на этом уровне приложения должны быть готовы повторять транзакции из-за сбоев сериализации. Фактически этот режим изоляции работает так же, как и Repeatable Read, только он дополнительно отслеживает условия, при которых результат параллельно выполняемых сериализуемых транзакций может не согласовываться с результатом этих же транзакций, выполняемых по очереди. Это отслеживание не привносит дополнительных препятствий для выполнения, кроме тех, что присущи режиму Repeatable Read, но тем не менее создаёт некоторую добавочную нагрузку, а при выявлении исключительных условий регистрируется *аномалия сериализации* и происходит *сбой сериализации*.

Любые данные, полученные из постоянной таблицы пользователя, не должны считаться действительными, пока транзакция, прочитавшая их, не будет успешно зафиксирована. Это верно даже для транзакций, не модифицирующих данные, за исключением случая, когда данные считываются в *откладываемой* транзакции такого типа. В этом случае данные могут считаться действительными, так как такая транзакция ждёт, пока не сможет получить снимок, гарантированно предотвращающий подобные проблемы. Во всех остальных случаях приложения не должны полагаться на результаты чтения данных в транзакции, которая не была

зафиксирована; в случае ошибки и отката приложения должны повторять транзакцию, пока она не будет завершена успешно.

Для полной гарантии сериализуемости применяются *предикатные блокировки*, то есть блокировки, позволяющие определить, когда запись могла бы повлиять на результат предыдущего чтения параллельной транзакции, если бы эта запись выполнялась сначала. Эти блокировки не приводят к фактическим блокировкам данных и, следовательно, никоим образом не могут повлечь взаимоблокировки транзакций. Они помогают выявить и отметить зависимости между параллельными транзакциями уровня Serializable, которые в определённых сочетаниях могут приводить к аномалиям сериализации. Транзакции Read Committed или Repeatable Read для обеспечения целостности данных, напротив, должны либо блокировать таблицы целиком, что помешает пользователям обращаться к этим таблицам, либо применять SELECT FOR UPDATE или SELECT FOR SHARE, что не только заблокирует другие транзакции, но и создаст дополнительную нагрузку на диск.

Предикатные блокировки в PostgreDB, как и в большинстве других СУБД, устанавливаются для данных, фактически используемых в транзакции. Они отображаются в системном представлении pg_locks со значением mode равным SIReadLock. Какие именно блокировки будут затребованы при выполнении запроса, зависит от плана запроса, при этом детализированные блокировки (например, блокировки строк) могут объединяться в более общие (например, в блокировки страниц) в процессе транзакции для экономии памяти, расходуемой для отслеживания блокировок. Транзакция READ ONLY может даже освободить свои блокировки SIRead до завершения, если обнаруживается, что конфликты, которые могли бы привести к аномалии сериализации, исключены. На самом деле для транзакций READ ONLY этот факт чаще всего устанавливается в самом начале, так что они обходятся без предикатных блокировок. Если же явно запросить транзакцию SERIALIZABLE READ ONLY DEFERRABLE, она будет заблокирована до тех пор, пока не сможет установить этот факт. (Это ***единственный*** случай, когда транзакции уровня Serializable блокируются, а транзакции Repeatable Read — нет.) С другой стороны, блокировки SIRead часто должны сохраняться и после фиксирования транзакции, пока не будут завершены другие, наложившиеся на неё транзакции.

При правильном использовании сериализуемые транзакции могут значительно упростить разработку приложений. Гарантия того, что любое сочетание успешно зафиксированных

параллельных сериализуемых транзакций даст тот же результат, что и последовательность этих транзакций, выполненных по очереди, означает, что если вы уверены, что единственная транзакция определённого содержания работает правильно, когда она запускается отдельно, вы можете быть уверены, что она будет работать так же правильно в любом сочетании сериализуемых транзакций, вне зависимости от того, что они делают, либо же она не будет зафиксирована успешно. При этом важно, чтобы в среде, где применяется этот подход, была реализована общая обработка сбоев сериализации (которые можно определить по значению SQLSTATE '40001'), так как заведомо определить, какие именно транзакции могут стать жертвами зависимостей чтения/записи и не будут зафиксированы для предотвращения аномалий сериализации, обычно очень сложно. Отслеживание зависимостей чтения-записи неизбежно создаёт дополнительную нагрузку, как и перезапуск транзакций, не зафиксированных из-за сбоев сериализации, но, если на другую чашу весов положить нагрузку и блокирование, связанные с применением явных блокировок и SELECT FOR UPDATE или SELECT FOR SHARE, использовать сериализуемые транзакции в ряде случаев окажется выгоднее.

Тогда как уровень изоляции транзакций Serializable в PostgreDB позволяет фиксировать параллельные транзакции, только если есть уверенность, что тот же результат будет получен при последовательном их выполнении, он не всегда предотвращает ошибки, которые не возникли бы при действительно последовательном выполнении. В частности, можно столкнуться с нарушениями ограничений уникальности, вызванными наложением сериализуемых транзакций, даже после явной проверки отсутствия ключа перед добавлением его. Этого можно избежать, если ***все*** сериализуемые транзакции, добавляющие потенциально конфликтующие ключи, будут предварительно явно проверять, можно ли вставить ключ. Например, приложение, добавляющее новый ключ, может запрашивать его у пользователя и затем проверять, существует ли он, сначала пытаясь найти его, либо генерировать новый ключ, выбирая максимальное существующее значение и увеличивая его на один. Если некоторые сериализуемые транзакции добавляют новые ключи сразу, не следуя этому протоколу, возможны нарушения ограничений уникальности, даже когда они не наблюдались бы при последовательном выполнении этих транзакций.

7. Работа с расширениями и модулями

В PostgreDB реализованы дополнительные модули и расширения, позволяющие выполнять запросы и использовать функции, недоступные в обычном PostgreSQL.

7.1 nt5autrans

nt5autrans- модуль для запуска AUTONOMOUS TRANSACTIONS.

Библиотека для выполнения запросов в дополнительном подключении к БД (автономные транзакции), поддерживаются только синхронные операции, не поддерживаются асинхронные, также нет поддержки персистентных подключений - новый запрос устанавливает повторное подключение.

Библиотека имеет один метод **nt5autrans_exec(text)** и **nt5autrans(text [, text])** с одним необязательным параметром (строкой подключения). Метод возвращает VOID в случае успеха и генерирует исключение в случае какого-либо сбоя.

Для установки расширения необходимо в программе psql выполнить:

CREATE EXTENSION nt5autrans; -- по умолчанию используется SCHEMA pg_catalog

SELECT nt5autrans_exec('select 1;');

Создать расширение можно в любой другой схеме, отличной от pg_catalog, но именно создание расширения в этой схеме позволит избежать необходимости указывать schema.nt5autrans_exec(), или же прописывать права и привилегии пользователям, а также добавлять public-схему в search_path (схема pw_catalog изначально находится в search_path).

Если потребуется удаление модуля, то выполнить:

DROP EXTENSION nt5autrans;

Пример использования №1:

Создать тестовую базу данных:

create database test_db;

Создать тестового пользователя:

create user test_user with login password '****' nosuperuser inherit nocreatedb nocreaterole noreplication;**

Выдать права пользователю **test_user** на пользование тестовой БД:

grant all on database test_db to test_user;

Далее необходимо завершить работу программы **psql** с помощью команды **\q**.

Поскольку расширения можно создавать только с правами суперпользователя, то необходимо подключиться к созданной БД, создать в ней схему и расширение, подключить и выдать права. Выполнить запрос:

psql "host=localhost dbname=test_db user=postgres"

Ввести пароль для пользователя **postgres**:

Password for user postgres: ****

При успешном выполнении команды на экране появляется строчка:

test_db=#

Создать схему:

create schema test_schema authorization test_user;

Создать расширение для схемы:

create extension nt5autrans schema test_schema;

Выдать права на использование схемы для **test_user:**

grant usage on schema test_schema to test_user;

Предоставить расширение на функцию:

grant execute on function test_schema.nt5autrans_exec(text) to test_user;

Далее необходимо завершить работу программы **psql** с помощью команды **\q**.

Ввести команду для подключения к **psql** пользователю **test_user** к **test_db**:

psql "host=localhost dbname=test_db user=test_user"

Ввести пароль для пользователя **test_user:**

Password for user test_user: ****

При успешном выполнении команды на экране появляется строчка:

testdb=>

Далее необходимо завершить работу программы **psql** с помощью команды **\q**.

Создать таблицу, в которую будет писаться данные (пример отладочного журнала), добавить в неё сообщение с помощью INSERT и с помощью функции расширения, отменить транзакцию, чтобы убедиться в сохранности "отладочного журнала. Выполнить запрос:

create table test_schema.log(line text not null, at timestamptz not null default now());

Открыть транзакцию. Выполнить команду:

begin;

При успешном выполнении команды на экране появляется строчка:

BEGIN

test_db=*>

Вставить строчку в таблицу log:

insert into test_schema.log values('This line will dissapear');

Выполнить запрос:

select test_schema.nt5autrans_exec(format('insert into test_schema.log select %L, %L', 'Hello, World!', clock_timestamp()::text),'host=localhost dbname=test_db user=test_user password=*** options=-csearch_path=test_schema');**

При успешном выполнении команды на экране появляется:

nt5autrans_exec

(1 строка)

Просмотреть данные в таблице. Выполнить запрос:

select * from test_schema.log;

При успешном выполнении команды на экране появляется таблица со вставленными данными:

line | at
----------+----------
Hello, World! | 2022-08-21 00:51:12.146678+03
This line will dissapear | 2022-08-21 00:51:19.93458+03

Откатить транзакцию:

rollback;

Просмотреть данные в таблице, выполнить запрос:

select * from test_schema.log;

При успешном выполнении команды на экране появляется таблица с одной строкой:

line | at

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

Hello, World! | 2022-08-21 00:51:12.146678+03

Как видно, одна из строк исчезла из таблицы в результате rollback-операции; а на строку Hello, World! операция rollback не распространяется (ёе добавление было выполнено в другом подключении).

Проверка функции nt5autrans_exec с различным указанием методов поиска объектов. Выполнить запрос:

create table test_schema.t(txt text);

Выполнить запросы:

select test_schema.nt5autrans_exec(format('insert into t select %L', 'first'), 'host=localhost dbname=test_db user=test_user password=**** options=-csearch_path=test_schema');**

select test_schema.nt5autrans_exec(format('insert into test_schema.t select %L', 'second'), 'host=localhost dbname=test_db user=test_user password=****');**

При успешном выполнении команды на экране появляются:

nt5autrans_exec

(1 строка)

Установить search_path для тестовой схемы:

SET search_path TO test_schema;

Выполнить запрос:

select nt5autrans_exec(format('insert into test_shema.t select %L', 'third'), test_db(> 'host=localhost dbname=test_db user=test_user password=****');**

При успешном выполнении команды на экране появляется:

nt5autrans_exec

(1 строка)

Выполнить запрос:

select nt5autrans_exec(format('insert into t1 select %L', 'four'), 'host=localhost dbname=test_db user=test_user password=****');**

В результате выполнения команды на экран выводится сообщение об ошибке:

ОШИБКА: отношение "t1" не существует

Вывести все данные из таблицы **t**. Выполнить запрос:

select * from t;

При успешном выполнении команды на экране появляется таблица:

txt

---

first

second

third

(3 строки)

Пример использования №2:

С помощью параметра **nt5autrans.connection_url** задать строку подключения по умолчанию. Выполнить команду:

psql "host=localhost dbname=test_db user=postgres"

Ввести пароль для пользвоателя **postgres**:

Password for user postgres: ****

При успешном выполнении команды на экране появляется строчка:

testdb=#

Выполнить команду:

set nt5autrans.connection_url to 'host=localhost dbname=test_db user=test_user password=****';**

Просмотреть установленный параметр. Выполнить запрос:

show nt5autrans.connection_url;

В результате выполнения команды на экран выводится таблица:

nt5autrans.connection_url ----------

host=localhost dbname=test_db user=test_user password=****

7.2 nt5auvars

nt5auvars- модуль для работы с именованными переменными в общей памяти сервера (AUTONOMOUS VARIABLES), которые ведут себя как sequences.

Предпосылкой создания этого расширения было:

- невозможность использования SEQUENCE, потому что в названии переменной могут быть любые произвольные символы;
- невозможность использования SEQUENCE, потому что длина имени переменной может превышать длину идентификатора;
- невозможность использования SEQUENCE, потому что у переменных динамическая природа появления/использования/удаления;

Любая конструкция DDL-языка (data definition language), которая передаётся СУБД вместе с DML-конструкциями (data manipulation language) выполнит автоматическое завершение транзакции, т.е. begin; select for update ...; create sequence ...; - завершит транзакцию;

Использование сессионных конфигурационных параметров, т.е. alter session set fa.setting1 to 1; не даст возможность объявить переменную, которая будет существовать в "соседнем" подключении к БД, т.е. в "соседней транзакции".

Кроме того, модуль nt5auvars (autonomous variables) предполагает предоставление такого способа работы с произвольными именованными переменными, чтобы доступ к переменным присутствовал и из сеанса, где переменная была создана, и из "соседнего" сеанса в том числе. Кроме того, значения переменных должны сохраняться и при переподключениях.

Для установки расширения необходимо в программе psql выполнить:

CREATE EXTENSION nt5auvars;

Если потребуется удаление модуля, то выполнить:

DROP EXTENSION nt5autvars;

Интерфейс (методы):

nt5auvars_init(name TEXT, initial INTEGER) INTEGER

Принимает название переменной и её исходное значение. Запоминает переменную. Сохраняет исходное значение. По умолчанию максимальным значением переменной считает -1. Возвращает переданное initial значение.

nt5auvars_init(name TEXT, initial INTEGER, max INTEGER) INTEGER

Принимает название переменной, её исходное и максимальное значения. Запоминает переменную. Сохраняет исходное и максимальное значения. Возвращает переданное initial значение.

nt5auvars_get_list() TEXT[]

Функция без параметров возвращает массив названий сохранённых переменных.

nt5auvars_is_present(name TEXT) BOOL

Принимает название переменной. Проверяет, сохранена ли переменная в памяти. Возвращает признак того, что переменная сохранена в памяти (есть переменная с указанным названием).

nt5auvars_nextval(name TEXT) INTEGER

Принимает название переменной. Проверяет наличие переменной и если переменная ранее была добавлена, то возвращает увеличенное на 1 значение, относительно ранее сохранённого. Возвращает NULL, если переменная с указанным названием ранее не была добавлена. Функция генерирует исключение, если в результате вызовов текущее значение переменной не может быть увеличено больше max-значения.

nt5auvars_nextval(name TEXT, count INTEGER) INTEGER

Возвращает увеличенное на **count** значение, относительно ранее сохранённого, если переменная ранее добавлялась. Функция генерирует исключение, если в результате вызовов текущее значение переменной не может быть увеличено больше max-значения.

nt5auvars_nextval(name TEXT, count INTEGER, save_value BOOL) INTEGER

Возвращает увеличенное на **count** значение, относительно ранее сохранённого, если переменная ранее добавлялась. Сохраняет возвращаемое значение или не сохраняет в зависимости от значения переменной **save_value**. Функция генерирует исключение, если в результате вызовов текущее значение переменной не может быть увеличено больше max-значения.

nt5auvars_nextvalue(name TEXT, initial INTEGER) INTEGER

Принимает название переменной и её исходное значение. Запоминает переменную, если она отсутствует в памяти, вместе с указанным исходным значением, при этом максимальное значение считает за -1. Возвращает переданное initial значение, если переменная не была ранее добавлена. Либо возвращает увеличенное на 1 значение, относительно ранее сохранённого, если переменная ранее добавлялась. Функция генерирует исключение, если в результате вызовов текущее значение переменной не может быть увеличено больше max-значения.

nt5auvars_nextvalue(name TEXT, initial INTEGER, max INTEGER) INTEGER

Принимает название переменной, её исходное и максимальное значения. Запоминает переменную, если она отсутствует в памяти, вместе с указанными исходным и максимальным значениями.

Возвращает переданное initial значение, если переменная не была ранее добавлена. Либо возвращает увеличенное на 1 значение, относительно ранее сохранённого, если переменная ранее

добавлялась. Функция генерирует исключение, если в результате вызовов текущее значение переменной не может быть увеличено больше max-значения.

nt5auvars_remove(name TEXT) BOOL

Принимает название переменной, удаляет переменную по указанному названию. Возвращает признак того, что переменная была удалена из памяти.

Пример использования:

```
select nt5auvars_get_list();
--  nt5auvars_get_list
-- --------------------
--
-- (1 row)

select nt5auvars_init('fa#.document#24', 1, 10);
--  nt5auvars_init
-- ----------------
--              1
-- (1 row)

select nt5auvars_nextval('fa#.document#100', 10, 12);
--  nt5auvars_nextval
-- -------------------
--             10
-- (1 row)

select nt5auvars_get_list();
-- nt5auvars_get_list
-- ----------------------------------
-- {fa#.document#24,fa#.document#100}
-- (1 row)

select unnest(nt5auvars_get_list());
-- unnest
```

```
-- -----------------
-- fa#.document#24
-- fa#.document#100
-- (2 rows)

select nt5auvars_nextval('fa#.document#100');
--  nt5auvars_nextval
-- -------------------
--               11
-- (1 row)

select nt5auvars_nextval('fa#.document#100');
--  nt5auvars_nextval
-- -------------------
--               12
-- (1 row)

select nt5auvars_nextval('fa#.document#100');
-- ERROR:  max value achieved

select nt5auvars_is_present('fa#.document#100');
--  nt5auvars_is_present
-- ----------------------
--  t
-- (1 row)

select nt5auvars_is_present('fa#.document#999');
--  nt5auvars_is_present
-- ----------------------
--  f
-- (1 row)

select nt5auvars_init('fa!', 1, 3);
--  nt5auvars_init
-- ----------------
--               1
-- (1 row)

select nt5auvars_is_present('fa!');
```

```
--  nt5auvars_is_present
-- ----------------------
--  t
-- (1 row)

select nt5auvars_remove('fa!');
--  nt5auvars_remove
-- ------------------
--  t
-- (1 row)

select nt5auvars_is_present('fa!');
--  nt5auvars_is_present
-- ----------------------
--  f
-- (1 row)

select x.var, nt5auvars_is_present(x.var) from (select unnest(nt5auvars_get_list()) as var) as x;
          var     | nt5auvars_is_present
------------------+----------------------
 fa#.document#100 | t
 fa#.document#24  | t
(2 rows)

select x.var, nt5auvars_remove(x.var) from (select unnest(nt5auvars_get_list()) as var) as x;
          var     | nt5auvars_remove
------------------+------------------
 fa#.document#100 | t
 fa#.document#24  | t
(2 rows)

select x.var, nt5auvars_remove(x.var) from (select unnest(nt5auvars_get_list()) as var) as x;
 var | nt5auvars_remove
-----+------------------
(0 rows)
```

7.3 nt5utf16

nt5utf16 - модуль для перекодировки хеш-сумм из utf 8 в utf16 и обратно.

Для установки расширения в программе psql выполнить:

CREATE EXTENSION nt5utf16;

Если потребуется удаление модуля, то выполнить:

DROP EXTENSION nt5utf16;

Расширение имеет 4 функции:

функция nt5utf16_bl16_to_bl8(bytea)

функция nt5utf16_bl16_to_v8(bytea)

функция nt5utf16_bl8_to_bl16(bytea)

функция nt5utf16_v8_to_bl16(character varying)

Пример использования:

select nt5utf16_v8_to_bl16('example');

nt5utf16_v8_to_bl16

\x006500780061006d0070006c0065

(1 строка)

select nt5utf16_bl16_to_bl8('3D37E1');

nt5utf16_bl16_to_bl8

\xe38d84e38cb7

(1 строка)

select nt5utf16_bl16_to_bl8('04.01.2001');

nt5utf16_bl16_to_bl8

\xe380b4e2b8b0e384ae

(1 строка)

7.4 nt5srvcall

Модуль nt5srvcall создан для выполнения запросов к внешним сервисам с использованием http/https.

Для установки расширения необходимо в программе psql выполнить:

CREATE EXTENSION nt5srvcall;

Если потребуется удаление модуля, то выполнить:

DROP EXTENSION nt5srvcall;

Методы, включенные в состав расширения:

tsql_convert(TEXT)

nt5srv_call(TEXT)

7.5 pg_hint_plan

Модуль **pg_hint_plan** позволяет корректировать планы выполнения, применяя так называемые «указания», записываемые в виде простых описаний в SQL-комментариях особого вида.

Чтобы установить расширение, необходимо добавить его в файл **postgresql.conf** с помощью строчки:

shared_preload_libraries = 'pg_hint_plan'

Далее создать расширение с помощью команды:

CREATE EXTENSION IF NOT EXISTS "pg_hint_plan";

Выполнить проверку установленных расширений с помощью команды

SELECT * FROM pg_extension;

В результате на экран выводится таблица с установленными расширениями, где присутствует запись **pg_hint_plan.**

Пример использования:

Создать среду для тестирования (таблицы **htest1, htest2, htest3,** вспомогательные индексы для двух таблиц **(htest1, htest2**), заполнить созданные таблицы тестовыми данными:

```
CREATE TABLE public.htest1 (
  id INT4 NULL,
  htest2Id INT4 NULL,
  value FLOAT8 NULL
);
CREATE INDEX htest1_id_idx ON htest1 (id);
CREATE INDEX htest1_value_idx ON htest1 (value);
CREATE INDEX htest1_htest2Id_idx ON htest1 (htest2Id);
CREATE TABLE public.htest2 (
  id INT4 NULL,
  htest3Id INT4 NULL,
  value FLOAT8 NULL,
```

```
  valuemd VARCHAR NULL
);
CREATE INDEX htest2_id_idx ON htest2 (id);
CREATE INDEX htest2_valuemd_idx ON htest2 (valuemd);
CREATE TABLE public.htest3 (
   id INT4 NULL,
   valuemd VARCHAR NULL
);

INSERT INTO htest1 (id, htest2id, value) VALUES( generate_series(1,1000000),
floor(random()*(100000-1)+1), floor(random()*(100-1)+1));
INSERT INTO htest2 VALUES ( generate_series(1,100000), floor(random()*(10000-1)+1),
floor(random()*(100-1)+1), md5(random()::varchar) );
INSERT INTO htest3 VALUES ( generate_series(1,10000), md5(random()::varchar));
```

Выполнить запрос с отображением плана построения:

```
EXPLAIN ANALYZE
WITH htt1 AS (select * FROM htest1 ht1 WHERE ht1.value < 10),
htt2 AS (select * FROM htest2 ht2 WHERE ht2.valuemd LIKE '%797%'),
htt3 AS (select * FROM htest3 ht3 WHERE ht3.valuemd LIKE '%016%'),
main AS (select * FROM htt1, htt2, htt3 WHERE htt2.id = htt1.htest2id)
SELECT *
FROM main;
```

В результате эксперимента на экран выводится план запроса, в котором представлено время выполнения **- 1946.203 мс.**

QUERY PLAN

Nested Loop (cost=4.51..2973.33 rows=909 width=102) (actual time=26.060..1933.977 rows=47250 loops=1)

```
  ->  Seq Scan on htest3 ht3  (cost=0.00..209.00 rows=101 width=37) (actual time=3.330..25.717
rows=75 loops=1)
        Filter: ((valuemd)::text ~~ '%016%'::text)
        Rows Removed by Filter: 9925
  ->  Materialize  (cost=4.51..2752.99 rows=9 width=65) (actual time=0.282..25.013 rows=630
loops=75)
        ->  Nested Loop  (cost=4.51..2752.94 rows=9 width=65) (actual time=21.119..1859.704 rows=630
loops=1)
              ->  Seq Scan on htest2 ht2  (cost=0.00..2281.00 rows=10 width=49) (actual time=1.963..156.939
rows=707
loops=1)
                    Filter: ((valuemd)::text ~~ '%797%'::text)
                    Rows Removed by Filter: 99293
              ->  Bitmap Heap Scan on htest1 ht1  (cost=4.51..47.18 rows=1 width=16) (actual
time=2.111..2.395 rows=1
 loops=707)
                    Recheck Cond: (htest2id = ht2.id)
                    Filter: (value < '10'::double precision)
                    Rows Removed by Filter: 9
                    Heap Blocks: exact=6995
                    ->  Bitmap Index Scan on htest1_htest2id_idx  (cost=0.00..4.51 rows=11 width=0) (actual
time=1.61
5..1.615 rows=10 loops=707)
                          Index Cond: (htest2id = ht2.id)
 Planning Time: 37.851 ms
 Execution Time: 1946.203 ms
(18 rows)
```

Выполнить запрос с использованием комментария-указателя «**IndexScan»,** который принудительно выбирает сканирование таблицы по индексу, с отображением плана построения:

```
LOAD 'pg_hint_plan';
EXPLAIN ANALYZE
/*+ IndexScan(ht1) IndexScan(ht2) */
WITH htt1 AS (select * FROM htest1 ht1 WHERE ht1.value < 10),
htt2 AS (select * FROM htest2 ht2 WHERE ht2.valuemd LIKE '%797%'),
htt3 AS (select * FROM htest3 ht3 WHERE ht3.valuemd LIKE '%016%'),
main AS (select * FROM htt1, htt2, htt3 WHERE htt2.id = htt1.htest2id)
SELECT *
FROM main;
```

В результате эксперимента на экран выводится план запроса, в котором представлено время выполнения **- 311.990 мс.**

```
                                QUERY PLAN

-----------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=4.51..2973.33 rows=909 width=102) (actual time=1.805..305.260 rows=47250
loops=1)
   ->  Seq Scan on htest3 ht3  (cost=0.00..209.00 rows=101 width=37) (actual time=0.051..7.268 rows=75
loops=1)
         Filter: ((valuemd)::text ~~ '%016%'::text)
         Rows Removed by Filter: 9925
   ->  Materialize  (cost=4.51..2752.99 rows=9 width=65) (actual time=0.016..3.693 rows=630 loops=75)
         ->  Nested Loop  (cost=4.51..2752.94 rows=9 width=65) (actual time=1.184..263.593 rows=630
loops=1)
               ->  Seq Scan on htest2 ht2  (cost=0.00..2281.00 rows=10 width=49) (actual time=0.069..113.610
rows=707
loops=1)
                     Filter: ((valuemd)::text ~~ '%797%'::text)
                     Rows Removed by Filter: 99293
               ->  Bitmap Heap Scan on htest1 ht1  (cost=4.51..47.18 rows=1 width=16) (actual
time=0.155..0.206 rows=1
```

loops=707)

Recheck Cond: (htest2id = ht2.id)

Filter: (value < '10'::double precision)

Rows Removed by Filter: 9

Heap Blocks: exact=6995

-> Bitmap Index Scan on htest1_htest2id_idx (cost=0.00..4.51 rows=11 width=0) (actual time=0.03

7..0.037 rows=10 loops=707)

Index Cond: (htest2id = ht2.id)

Planning Time: 7.876 ms

Execution Time: 311.990 ms

(18 rows)

В данном случае время выполнения существенно сократилось.

7.6 BCP (быстрая загрузка данных)

Утилита предназначена для конвертации файлов бэкапов баз данных MSSQL, в которых в качестве разделителя колонок используется знак '|', в бинарные файлы, полученные из утилиты pg_dump при стандартном сжатии данных.

Аргументы для программы могут задаваться при помощи:

- аргументов командной строки (1-ый приоритет);
- файла config.yaml (2-ой приоритет);
- для некоторых аргументов предусмотрено значение по умолчанию.

Перечень параметров для утилиты предоставляется следующим списком:

- file (string) – имя файла, данные из которого необходимо сконвертировать. Значение по умолчанию: file.txt
  (-f или --file для командной строки);

- resultFile (string) – имя бинарного файла, полученного в результате конвертации утилитой bcp2dump. Значение по умолчанию: result.bin
  (-r или --resultFile для командной строки);
- table (строчка) – имя таблицы, в которую необходимо выгрузить данные в postgresql. Значение по умолчанию: newtable_1
  (-t или --table для командной строки);
- encoding (строчка) – кодировка, в которой находится база данных postgresql, в которую будет выгружаться целевой бинарный файл. Значение по умолчанию: UTF-8
  (-e или --encoding для командной строки);
- columns (массив) – массив названия колонок в таблице postgresql, заданной параметром table. Значение по умолчанию: пустой массив
  (-c или --columns для командной строки. Аргументы разделяются через запятую, поддерживаются мультиаргументы.) Пример для конфиг файла: columns: ["column1", "column2", "column3", "column4"]
- encodingDataFrom (строчка) – кодировка, из которой надо перекодировать данные. Значение по умолчанию: WINDOWS-1251
  (-a или --encodingDataFrom для командной строки);
- encodingDataTo (строчка) – кодировка, в которую надо перекодировать данные перед конвертацией в бинарный формат. Значение по умолчанию: UTF-8
  (-d или --encodingDataTo для командной строки);
- debug (bool) - поддержка дебага. Поддерживается только в конфиг файле. Значение по умолчанию: false.

Пример задания аргументов из командной строки:

./bcp2dump –c column1,column2 –f file.txt –c column3

8. Управление доступом к объектам БД, права пользователей

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*.

Набор прав, применимых к определённому объекту, зависит от типа объекта (таблица, функция и т. д.). Право изменять или удалять объект является неотъемлемым правом владельца объекта, его нельзя лишиться или передать другому объекту.

Объекту можно назначить нового владельца с помощью команды **ALTER** для соответствующего типа объекта, например:

ALTER TABLE *table_name* OWNER TO *new_owner;*

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**. Право на изменение таблицы можно дать этой роли так:

GRANT UPDATE ON *new_table* TO *username*;

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

Чтобы лишить пользователей ранее выданных им прав, используйте команду **REVOKE**:

REVOKE ALL ON *new_table* FROM PUBLIC;

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.

SELECT

Позволяет выполнять **SELECT** для любого столбца или перечисленных столбцов в заданной таблице, представлении, матпредставлении или другом объекте табличного вида. Также позволяет выполнять **COPY TO**. Помимо этого, данное право требуется для обращения к существующим значениям столбцов в **UPDATE** или **DELETE**. Для последовательностей это право позволяет пользоваться функцией **currval**. Для больших объектов оно позволяет читать содержимое объекта.

INSERT

Позволяет вставлять с помощью **INSERT** строки в заданную таблицу, представление и т. п. Может назначаться для отдельных столбцов; в этом случае только этим столбцам можно присваивать значения 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, представлении и т. п. (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.) Для **SELECT ... FOR UPDATE и SELECT ... FOR SHARE** также требуется иметь это право как минимум для одного столбца, помимо права **SELECT**. Для последовательностей это право позволяет пользоваться функциями **nextval** и **setval**. Для больших объектов это право позволяет записывать данные в объект или обрезать его.

DELETE

Позволяет удалять с помощью команды **DELETE** строки из таблицы, представления и т. п. (На практике для любой нетривиальной команды **DELETE** потребуется также право **SELECT**, так как она должна обратиться к колонкам таблицы, чтобы определить, какие строки подлежат удалению.)

TRUNCATE

Позволяет опустошать таблицу с помощью **TRUNCATE.**

REFERENCES

Позволяет создавать ограничение внешнего ключа, обращающееся к таблице или определённым столбцам таблицы.

CREATE

Для баз данных это право позволяет создавать схемы и публикации, а также устанавливать доверенные расширения в конкретной базе. Для схем это право позволяет создавать новые объекты в заданной схеме. Чтобы переименовать существующий объект, необходимо быть его владельцем ***и*** иметь это право для схемы, содержащей его. Для табличных пространств это право позволяет создавать таблицы, индексы и временные файлы в определённом табличном пространстве, а также создавать базы данных, для которых это пространство будет основным.

CONNECT

Позволяет подключаться к базе данных. Это право проверяется при установлении соединения (в дополнение к условиям, определённым в конфигурации **pg_hba.conf**). Для схем это право даёт доступ к содержащимся в них объектам (предполагается, что при этом имеются права, необходимые для доступа к самим объектам). По сути это право позволяет субъекту «просматривать» объекты внутри схемы. Без этого разрешения имена объектов всё же можно будет узнать, например, обратившись к системным каталогам. Кроме того, если отозвать это право, в существующих сеансах могут оказаться операторы, для которых просмотр имён объектов был выполнен ранее, так что это право не позволяет абсолютно надёжно перекрыть доступ к объектам. Для последовательностей это право позволяет использовать функции **currval** и **nextval.** Для типов и доменов это право позволяет использовать заданный тип или домен при создании таблиц, функций или других объектов схемы. Для обёрток сторонних данных это право позволяет создавать использующие их определения сторонних серверов. Для сторонних серверов это право позволяет создавать использующие их сторонние таблицы.

Наделённые этим правом могут также создавать, модифицировать или удалять собственные сопоставления пользователей, связанные с определённым сервером.