Базы данных

Лекция 12 – Управление транзакциями

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2024.04.23

Оглавление

Транзакции

Явным образом транзакции создаются командой **START TRANSACTION** и завершаются командой **COMMIT** (фиксация, регистрация) или **ROLLBACK** (откат, прерывание).

Выполнять START TRANSACTION, чтобы начать блок транзакции, необязательно – блок неявно начинает любая команда SQL. Поведение Postgres можно представить как неявное выполнение COMMIT после каждой команды, которой не предшествует START TRANSACTION (или BEGIN), и поэтому такое поведение часто называется «автофиксацией». Другие реляционные СУБД тоже могут обеспечивать автофиксацию.

Транзакции могут разбиваться на части – субтранзакции.

Субтранзакции можно также запускать в других субтранзакциях.

Транзакция верхнего уровня и её дочерние подтранзакции формируют иерархию или дерево, поэтому по отношению к основной транзакции используется термин «транзакция верхнего уровня».

Субтранзакции запускаются внутри транзакций и позволяют разбивать более крупные транзакции на компоненты, которые могут фиксироваться или прерываться, не влияя на родительские транзакции, которые, соответственно, могут продолжать выполняться.

Такое решение позволяет обрабатывать ошибки проще и часто используется при разработке приложений. В английском языке слово субтранзакция зачастую сокращается до subxact.

Субтранзакции могут явным образом запускаться при помощи команды **SAVEPOINT**.

Транзакции и идентификаторы

С каждой транзакцией связаны уникальные идентификаторы.

Идентификаторы используются в качестве основы механизма многоверсионного управления конкурентным доступом MVCC.

Есть два типа идентификаторов транзакций – виртуальный и невиртуальный.

Виртуальный идентификатор `VirtualTransactionId` (также именуемый `virtualXID` или `vxid`) состоит из идентификатора обслуживающего процесса (`backendID`) и последовательно назначаемого внутреннего номера для данного процесса (`localXID`).

Например, виртуальный идентификатор 4/12532 состоит из следующих компонентов: `backendID` со значением 4 и `localXID` со значением 12532.

Невиртуальные идентификаторы `TransactionId` или `xid`, например 278394, последовательно выбираются для транзакций из глобального счётчика, который используется всеми базами данных в рамках кластера PostgreSQL (верхний уровень).

Значение невиртуальному идентификатору присваивается при первой операции записи транзакции в базу данных. Это означает, что транзакции с меньшими `xid` начинают запись раньше транзакций с большими `xid`.

Порядок, в котором транзакции выполняют впервые запись в базу данных, может отличаться от порядка, в котором они запускаются, особенно если транзакции начинаются с операторов, выполняющих только операции чтения.

Внутренний тип идентификаторов транзакций `xid` имеет размер 32 бита, и значения в нём повторяются через каждые 4 миллиарда транзакций. После каждого цикла 32-битная эпоха увеличивается на 1. Существует 64-битный тип `xid8`, который включает эту эпоху и поэтому не повторяется на протяжении жизни сервера. Его можно преобразовать в `xid` посредством операции приведения типа.

Субтранзакции, выполняющейся в режиме записи, присваивается невиртуальный идентификатор, его называют «subxid».

Субтранзакциям, выполняющимся в режиме только чтения, subxid не присваиваются, однако при первой попытке записи такой идентификатор будет присвоен.

При этом всем родительским транзакциям субтранзакции, вплоть до транзакции верхнего уровня включительно, присваивается невиртуальный идентификатор.

Значение xid родительской транзакции всегда будет меньше значений subxid субтранзакций.

При фиксации субтранзакции все зафиксированные дочерние субтранзакции с идентификатором `subxid` также считаются зафиксированными в рамках этой подтранзакции.

При прерывании подтранзакции все дочерние подтранзакции также считаются прерванными. При этом в случае прерывании транзакции верхнего уровня все её подтранзакции также прерываются, даже если они были зафиксированы.

Чем больше подтранзакций остаётся открытыми в каждой транзакции (в отношении которых не выполнен откат или освобождение), тем выше будут издержки, связанные с управлением транзакциями.

Для каждого сервера в общей памяти кешируется до 64 открытых subxid. После этого этапа издержки хранения для операций ввода/вывода существенно возрастают.

Изоляция транзакций

Стандарт SQL определяет четыре уровня изоляции транзакций.

Наиболее строгий из них — *сериализуемый* (Serializable) – 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 могут иметь место при взаимодействии параллельных транзакций, но не допускаются на определённом уровне.

Из определения сериализуемого уровня вытекает, что на этом уровне ни одно из этих явлений не возможно – если эффект транзакций должен быть тем же, что и при их выполнении по очереди, кто увидеть особые явления, связанные с другими транзакциями, 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.

Уровни изоляции транзакций, описанные в стандарте SQL и реализованные в Postgres

Уровень изоляции	**«Грязное» чтение**	**Неповторяемое чтение**	**Фантомное чтение**	**Аномалия сериализации**
Read uncommited (Чтение незафиксированных данных)	Допускается, но не в PG	Возможно	Возможно	Возможно
Read committed (Чтение зафиксированных данных)	Невозможно	Возможно	Возможно	Возможно
Repeatable read (Повторяемое чтение)	Невозможно	Невозможно	Допускается, но не в PG	Возможно
Serializable (Сериализуемость)	Невозможно	Невозможно	Невозможно	Невозможно

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

Причина этого в том, что только так можно сопоставить стандартные уровни изоляции с реализованной в Postgres архитектурой многоверсионного управления конкурентным доступом. Также реализация Repeatable Read в Postgres не допускает фантомного чтения.

Для выбора нужного уровня изоляции транзакций используется команда SET TRANSACTION.

Важно

Поведение некоторых функций и типов данных PostgreSQL в транзакциях подчиняется особым правилам. В частности, изменения последовательностей (и следовательно, счётчика в столбце, объявленному как serial) немедленно видны во всех остальных транзакциях и не откатываются назад, если выполнившая их транзакция прерывается.

Уровень изоляции Read Committed

Read Committed — уровень изоляции транзакции, выбираемый в PostgreSQL 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.

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 при поиске целевых строк ведут себя подобно SELECT – 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).

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.

При выполнении INSERT с предложением ON CONFLICT DO NOTHING строка в результате действия другой транзакции может не добавиться.

Вследствие описанного выше , 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.

Так как в режиме Read Committed каждая команда начинается с нового снимка состояния, который включает результаты всех транзакций, зафиксированных к этому моменту, последующие команды в одной транзакции будут в любом случае видеть эффекты всех параллельных зафиксированных транзакций. Вопрос здесь состоит в том, видит ли одна команда абсолютно согласованное состояние базы данных.

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.

Уровень изоляции Repeatable Read

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.

При этом каждый запрос будет видеть эффекты предыдущих изменений в своей транзакции, несмотря на то, что они не зафиксированы.

Это самое строгое требование, которое стандарт SQL вводит для этого уровня изоляции, и при его выполнении предотвращаются все явления, описанные в таблице, за исключением аномалий сериализации.

Этот уровень отличается от Read Committed тем, что запрос в транзакции данного уровня видит снимок данных на момент начала первого оператора в транзакции, 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.

Приложения, использующие этот уровень, должны быть готовы повторить транзакции в случае сбоев сериализации.

Команды UPDATE, DELETE, MERGE, 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, удалены или заблокированы другой параллельной транзакцией. В этом случае транзакция в режиме Repeatable Read будет ожидать фиксирования или отката первой изменяющей данные транзакции. В случае отката первой транзакции все завершится успешно.

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.

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).

Уровень изоляции Serializable

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.

SET TRANSACTION — характеристики текущей транзакции

Синтаксис

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

Команда SET TRANSACTION устанавливает характеристики текущей транзакции.

На последующие транзакции она не влияет. Однако, в рамках сеанса можно установить характеристики транзакции по умолчанию, используя команду **SET SESSION CHARACTERISTICS**. Заданные по умолчанию характеристики для отдельных транзакций командой SET TRANSACTION можно переопределить.

transaction_mode принимает следующие значения:

ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }

READ WRITE | READ ONLY

[NOT] DEFERRABLE

и определяет следующие характеристики транзакции:

- уровень изоляции транзакции (ISOLATION LEVEL);
- режим доступа транзакции (чтение/запись или только чтение);
- допустимость откладывания транзакции.

В дополнение к ним можно выбрать снимок, но только для текущей транзакции, не для сеанса по умолчанию.

Уровень изоляции транзакции

```
ISOLATION LEVEL
{ READ COMMITTED | REPEATABLE READ | SERIALIZABLE | READ UNCOMMITTED }
```

Уровень изоляции транзакции определяет, какие данные может видеть транзакция, когда *параллельно* с ней выполняются другие транзакции:

READ COMMITTED – оператор видит только те строки, которые были зафиксированы до начала его выполнения. Этот уровень устанавливается по умолчанию.

REPEATABLE READ – все операторы текущей транзакции видят только те строки, которые были зафиксированы перед первым запросом на выборку или изменение данных, выполненным в этой транзакции.

SERIALIZABLE – все операторы текущей транзакции видят только те строки, которые были зафиксированы перед первым запросом на выборку или изменение данных, выполненным в этой транзакции.

Наложение операций чтения и записи параллельных сериализуемых транзакций может привести к ситуации, которая невозможна при последовательном их выполнении (когда одна транзакция выполняется за другой). В этом случае произойдёт откат одной из транзакций с ошибкой `serialization_failure` (сбой сериализации).

READ UNCOMMITTED – в Postgres данный уровень обрабатывается как READ COMMITTED.

Уровень изоляции транзакции нельзя изменить после выполнения первого запроса на выборку или изменение данных в текущей транзакции, используя SELECT, INSERT, DELETE, UPDATE, MERGE, FETCH или COPY.

Режим доступа транзакции

```
READ WRITE | READ ONLY
```

Режим доступа транзакции определяет, будет ли транзакция только читать данные или будет и читать, и писать. По умолчанию подразумевается READ WRITE.

В транзакции READ ONLY запрещаются:

INSERT, UPDATE, DELETE, MERGE и COPY FROM (если только целевая таблица не временная;

любые команды CREATE, ALTER и DROP, COMMENT, GRANT, REVOKE, TRUNCATE;

а также EXPLAIN ANALYZE и EXECUTE, если команда, которую они должны выполнить, относится к вышеперечисленным.

Это высокоуровневое определение режима только для чтения, которое в принципе не исключает запись на диск.

Допустимость откладывания транзакции

```
[ NOT ] DEFERRABLE
```

Свойство DEFERRABLE оказывает влияние, только если транзакция находится также в режимах SERIALIZABLE и READ ONLY.

Если для транзакции установлены все три этих свойства, транзакция может быть заблокирована при первой попытке получить свой снимок данных, после чего она сможет выполняться без дополнительных усилий, обычных для режима SERIALIZABLE, и без риска привести к сбою сериализации или пострадать от него.

Этот режим подходит для длительных операций, например для построения отчётов или резервного копирования.

Значение DEFERRABLE параметра `transaction_mode` является языковым расширением PostgreSQL.

SET TRANSACTION SNAPSHOT

```
SET TRANSACTION SNAPSHOT snapshot_id
```

Команда SET TRANSACTION SNAPSHOT позволяет начать новую транзакцию со снимком данных, который получила уже существующая транзакция.

Эта ранее созданная транзакция должна экспортировать снимок с помощью функции `pg_export_snapshot ()`, которая возвращает идентификатор снимка, который передается команде SET TRANSACTION SNAPSHOT в качестве идентификатора импортируемого снимка.

Команду SET TRANSACTION SNAPSHOT можно выполнить только в начале транзакции, до первого запроса на выборку или изменение данных (SELECT, INSERT, DELETE, UPDATE, MERGE, FETCH или COPY) в текущей транзакции.

Для транзакции при этом уже должен быть установлен уровень изоляции SERIALIZABLE или REPEATABLE READ. В противном случае снимок будет сразу же потерян, так как на уровне READ COMMITTED для каждой команды делается новый снимок.

Если импортирующая транзакция работает на уровне изоляции SERIALIZABLE, то транзакция, экспортирующая снимок, также должна работать на этом уровне.

Транзакции в режиме READ WRITE не могут импортировать снимок из транзакции в режиме READ ONLY.

START TRANSACTION — начать блок транзакции

```
START TRANSACTION [ transaction_mode [, ...] ]
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
```

transaction_mode может быть следующим:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED |
READ UNCOMMITTED }
READ WRITE | READ ONLY
[ NOT ] DEFERRABLE
```

Эта команда начинает новый блок транзакции, то есть обозначает, что все операторы после команды START TRANSACTION и до явной команды COMMIT или ROLLBACK будут выполняться в одной транзакции.

Если указан уровень изоляции, режим READ WRITE или допустимость откладывания транзакции, новая транзакция получит эти характеристики, как при выполнении команды **SET TRANSACTION**.

В блоке транзакции операторы выполняются быстрее, так как для запуска/фиксации транзакции производится масса операций, нагружающих процессор и диск. Кроме того, выполнение нескольких операторов в одной транзакции позволяет обеспечить целостность при внесении серии связанных изменений – другие сеансы не видят промежуточное состояние, когда произошли ещё не все связанные изменения.

Стандарт SQL требует, чтобы последовательные `transaction_mode` разделялись запятыми, но Postgres позволяет запятые опустить.

Пример

Чтобы начать новую транзакцию со снимком данных, который получила уже существующая транзакция, его нужно сначала экспортировать из первой транзакции.

При этом будет получен идентификатор снимка:

```
START TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
 00000003-0000001B-1
(1 row)
```

Затем этот идентификатор нужно передать команде SET TRANSACTION SNAPSHOT в начале новой транзакции как строковый литерал:

```
START TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

COMMIT — зафиксировать текущую транзакцию

```
COMMIT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

COMMIT фиксирует текущую транзакцию.

Все изменения, произведённые транзакцией, становятся видимыми для других и гарантированно сохранятся в случае сбоя.

WORK, **TRANSACTION** – необязательные ключевые слова, не оказывают никакого влияния.

AND CHAIN – если добавлено указание AND CHAIN, сразу после окончания текущей транзакции начинается новая с такими же характеристиками.

В противном случае новая транзакция не начинается.

При попытке выполнить COMMIT вне транзакции ничего не произойдёт, но будет выдано предупреждение.

COMMIT AND CHAIN вне транзакции вызовет ошибку.

ROLLBACK — прервать текущую транзакцию

```
ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
ABORT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

ROLLBACK откатывает текущую транзакцию и приводит к аннулированию всех изменений, произведённых транзакцией.

WORK, **TRANSACTION** – необязательные ключевые слова, не оказывают никакого влияния.

AND CHAIN – если добавлено указание AND CHAIN, сразу после окончания текущей транзакции начинается новая (не прерванная) с такими же характеристиками.
В противном случае новая транзакция не начинается.

При выполнении команды ROLLBACK вне блока транзакции выдаётся предупреждение и больше ничего не происходит.
ROLLBACK AND CHAIN вне блока транзакции вызывает ошибку.

SAVEPOINT — определить новую точку сохранения в текущей транзакции

```
SAVEPOINT savepoint_name
```

SAVEPOINT устанавливает новую точку сохранения в текущей транзакции.

Точка сохранения — это специальная отметка внутри транзакции, которая позволяет откатить все команды, выполненные после неё, и восстановить таким образом состояние на момент установки этой точки.

savepoint_name – Имя, назначаемое новой точке сохранения.

Ранее существующая точка с таким именем оказывается недоступной, пока не будут освобождены установленные позже одноимённые точки.

Для отката к установленной точке сохранения предназначена команда **ROLLBACK TO**.

Чтобы уничтожить точку сохранения, сохраняя изменения, произведённые после того, как она была установлена, применяется команда **RELEASE SAVEPOINT**.

Точки сохранения могут быть установлены только внутри блока транзакции. В одной транзакции можно определить несколько точек сохранения.

Пример 1

Установление точки сохранения и затем отмена действия всех команд, выполненных после установленной точки:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

Показанная транзакция вставит в таблицу значения 1 и 3, но не 2.

Пример 2

Этот пример показывает, как установить и затем уничтожить точку сохранения:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

Данная транзакция вставит значения 3 и 4.

Пример 3

Этот пример показывает, как использовать точки сохранения с одним именем:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);

    -- откат ко второй точке сохранения
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;                -- вывод строк 1 и 2

    -- освобождение второй точки сохранения
    RELEASE SAVEPOINT my_savepoint;

    -- откат к первой точке сохранения
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;                -- вывод только строки 1
COMMIT;
```

Данная транзакция сначала откатит строку 3, затем строку 2.

Совместимость

Стандарт SQL требует, чтобы точка сохранения уничтожалась автоматически, когда устанавливается другая точка сохранения с тем же именем.

В Postgres старая точка сохранения остаётся, хотя при откате или уничтожении будет выбираться только самая последняя – после уничтожения последней точки командой RELEASE SAVEPOINT доступной для команд ROLLBACK TO SAVEPOINT и RELEASE SAVEPOINT становится следующая точка сохранения.

В остальном оператор SAVEPOINT полностью соответствует стандарту.

ROLLBACK TO SAVEPOINT — откатиться к точке сохранения

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

Откатывает все команды, выполненные после установления точки сохранения, и начинает новую субтранзакцию на том же уровне транзакции.

Точка сохранения остаётся действующей и при необходимости можно снова откатиться к ней позже.

ROLLBACK TO SAVEPOINT неявно уничтожает все точки сохранения, установленные после заданной точки.

Чтобы уничтожить точку сохранения, не отменяя действия команд, выполненных после неё, применяется команда RELEASE SAVEPOINT.

Указание имени точки сохранения, не установленной ранее, считается ошибкой.

Совместимость

Стандарт SQL утверждает, что ключевое слово SAVEPOINT является обязательным, но Postgres и Oracle позволяют его опускать.

Стандарт SQL в качестве необязательного допускает WORK, но не TRANSACTION после ROLLBACK.

В стандарте SQL есть дополнительное предложение AND [NO] CHAIN, которое в настоящее время в Postgres не поддерживается.

RELEASE SAVEPOINT — освободить ранее определённую точку сохранения

```
RELEASE [ SAVEPOINT ] savepoint_name
```

Команда RELEASE SAVEPOINT освобождает указанную точку сохранения и все активные точки сохранения, созданные после указанной, а также освобождает соответствующие ресурсы.

Все изменения, сделанные с момента создания точки сохранения и не отменённые, на момент создания указанной точки сохранения командой RELEASE SAVEPOINT объединяются в активную транзакцию или в точку сохранения.

Изменения, сделанные после RELEASE SAVEPOINT, также включаются в активную транзакцию или точку сохранения.

Указание имени точки сохранения, не определённой ранее, считается ошибкой.

Если одно имя дано нескольким ранее определённым точкам сохранения, освобождена будет только последняя из них. Повторные команды будут последовательно освобождать более ранние точки сохранения.

SET CONSTRAINTS — установить время проверки ограничений для текущей транзакции

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

SET CONSTRAINTS определяет, когда будут проверяться ограничения в текущей транзакции.

Ограничения IMMEDIATE проверяются в конце каждого оператора, а ограничения DEFERRED откладываются до фиксации транзакции.

Режим IMMEDIATE или DEFERRED задаётся для каждого ограничения независимо.

При создании ограничение получает одну из следующих характеристик:

DEFERRABLE INITIALLY DEFERRED – откладываемое, изначально отложенное;

DEFERRABLE INITIALLY IMMEDIATE – откладываемое, изначально немедленное;

NOT DEFERRABLE – неоткладываемое.

Третий вариант всегда подразумевает IMMEDIATE и на него команда SET CONSTRAINTS не влияет.

Первые два варианта запускаются в каждой транзакции в указанном режиме, но их поведение можно изменить в рамках транзакции командой SET CONSTRAINTS.

SET CONSTRAINTS со списком имён ограничений меняет режим только этих ограничений, при этом все они должны быть откладываемыми.

Имя любого ограничения можно дополнить схемой. Если имя схемы не указано, в поисках первого подходящего имени будет просматриваться текущий путь поиска схем.

SET CONSTRAINTS ALL меняет режим всех откладываемых ограничений.

Когда SET CONSTRAINTS меняет режим ограничения с DEFERRED на IMMEDIATE, все изменения данных, ожидающие проверки в конце транзакции, вместо этого проверяются в момент выполнения команды SET CONSTRAINTS.

Если какое-либо ограничение нарушается, при выполнении SET CONSTRAINTS происходит ошибка (и режим проверки не меняется).

Таким образом, с помощью SET CONSTRAINTS можно принудительно проверить ограничения в определённом месте транзакции.

В настоящее время это распространяется только на ограничения UNIQUE, PRIMARY KEY, REFERENCES (внешний ключ) и EXCLUDE.

Ограничения NOT NULL и CHECK всегда проверяются немедленно в момент добавления или изменения строки (не в конце оператора).

Ограничения уникальности и ограничения-исключения, объявленные без указания DEFERRABLE, так же проверяются немедленно.

Двухфазные транзакции

Postgres поддерживает протокол двухфазной фиксации (2PC), позволяющий нескольким распределённым системам работать в транзакционной манере.

Каждым ресурсом, используемым в транзакции, управляет диспетчер ресурсов, действия которого координируются диспетчером транзакций. В конце транзакции приложение запрашивает ее фиксацию или откат. Диспетчер транзакций обрабатывает ситуации, в которых один из диспетчеров ресурсов голосует за фиксацию, а другие – за откат транзакции.

Протокол двухфазной фиксации состоит из двух фаз – фаза подготовки и фаза фиксации, и гарантирует, что при завершении транзакции все изменения, произведенные над всеми ресурсами, либо полностью фиксируются, либо полностью откатываются.

После завершения транзакции результат сообщается всем участникам.

В первой фазе транзакции диспетчер транзакций опрашивает каждый ресурс, чтобы определить, следует ли выполнить фиксацию или откат транзакции.

Во второй фазе транзакции диспетчер транзакций уведомляет каждый ресурс о результате опроса, позволяя ресурсам выполнить необходимые операции очистки.

В рамках этого протокола используются следующие команды:

PREPARE TRANSACTION;

COMMIT PREPARED;

ROLLBACK PREPARED.

Двухфазные транзакции предназначены для использования внешними системами управления транзакциями. Когда пользователь выполняет команду PREPARE TRANSACTION для подготовки транзакции, следующим шагом он может выполнить только команду COMMIT PREPARED или ROLLBACK PREPARED.

Управление конкурентным доступом

Postgres предоставляет богатый набор средств для управления конкурентным доступом к данным, основанным на модели MVCC (Multiversion Concurrency Control[1]).

Это означает, что каждый SQL-оператор видит снимок данных (версию базы данных) на определённый момент времени, вне зависимости от текущего состояния данных.

Такой подход защищает операторы от несогласованности данных, которая возможна, если другие конкурирующие транзакции внесут изменения в те же строки данных, и обеспечивает тем самым изоляцию транзакций для каждого сеанса баз данных. MVCC отличается от традиционных методик блокирования, что позволяет снизить уровень конфликтов блокировок и повысить производительность в многопользовательской среде.

Преимущество использования MVCC по сравнению с блокированием заключается в том, что блокировки MVCC, полученные для чтения данных, не конфликтуют с блокировками, полученными для записи, и поэтому чтение никогда не мешает записи, а запись чтению даже для самого строгого уровня изоляции транзакций SSI (Serializable Snapshot Isolation[2]).

Для приложений, которым в принципе не нужна полная изоляция транзакций и которые предпочитают явно определять точки конфликтов, в PostgreSQL также есть средства блокировки на уровне таблиц и строк. Однако при правильном использовании MVCC обычно обеспечивает лучшую производительность, чем блокировки. Кроме этого, приложения могут использовать рекомендательные блокировки, не привязанные к какой-либо одной транзакции.

---

1) Многоверсионное управление конкурентным доступом
2) Сериализуемая изоляция снимков