Практические работы по дисциплине «Технологии обработки транзакций клиент-серверных приложений» направления подготовки бакалавриата 09.03.04 «Программная инженерия»

Практическая работа №4

Транзакции. Снимки данных в PostgreSQL.

Теория для понимания практики, желательна к выполнению:

Что такое снимок данных

Физически в страницах данных могут находиться несколько версий одной и той же строки, хотя каждая транзакция должна видеть максимум одну из них. Все вместе версии разных строк, наблюдаемые транзакцией, образуют снимок данных. Снимок обеспечивает согласованную в ACID смысле картину данных на определенный момент времени и содержит только самые актуальные данные, зафиксированные к моменту его создания.

Чтобы обеспечить изоляцию, каждая транзакция работает со своим собственным снимком. При этом разные транзакции видят разные, но тем не менее согласованные (на разные моменты времени) данные.

На уровне изоляции Read Committed снимок создается в начале каждого оператора транзакции и остается активным все время работы этого оператора.

На уровнях Repeatable Read и Serializable снимок создается один раз в начале первого оператора транзакции и остается активным до самого конца транзакции.

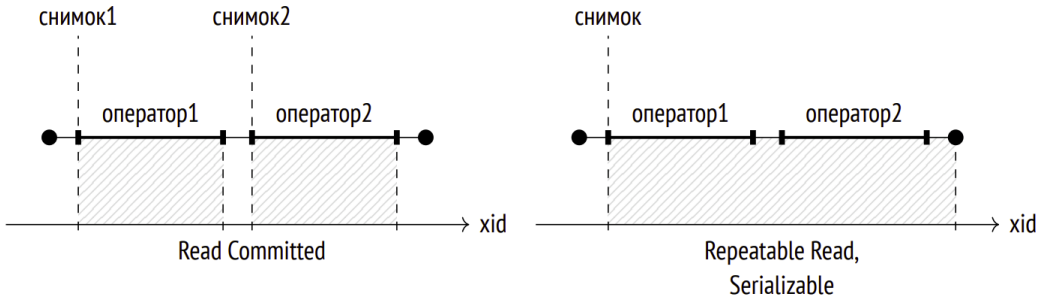

Рисунок 1 – снимки данных на уровнях изоляции в PostgreSQL

Видимость версий строк в снимке

Конечно, снимок данных не является физической копией всех

необходимых версий строк. Фактически снимок задается несколькими числами, а видимость версий строк в снимке определяется правилами.

Будет данная версия строки видна в снимке или нет — зависит от полей xmin и xmax ее заголовка (то есть от номеров создавшей и удалившей транзакций) и от соответствующих этим полям информационных битов. Интервалы xmin–xmax не пересекаются, поэтому каждая строка представлена в любом снимке максимум одной своей версией.

Точные правила видимости довольно сложны и учитывают множество различных ситуаций и крайних случаев. Упрощая, можно сказать, что версия строки видна, когда в снимке видны изменения, сделанные транзакцией xmin, и не видны изменения, сделанные транзакцией xmax (иными словами, если версия строки уже появилась, но еще не удалена).

В свою очередь, изменения транзакции видны в снимке, если эта транзакция была зафиксирована до момента создания снимка. И еще, в качестве исключения из общего правила, транзакция видит в снимке свои собственные еще не зафиксированные изменения. Изменения оборванных транзакций, разумеется, ни в одном снимке не видны.

Вот простой пример. Транзакции изображены в виде отрезков (от момента начала до момента фиксации):

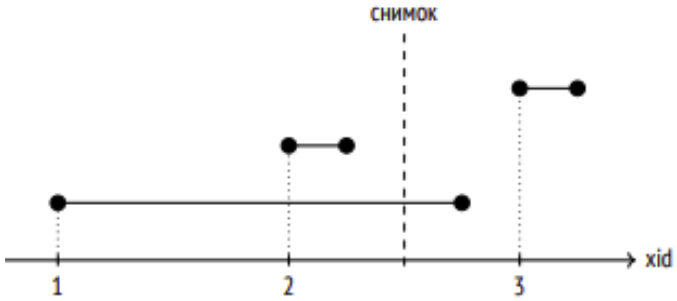

Рисунок 2 – транзакции от момента начала до момента фиксации

Здесь:

- изменения транзакции 2 будут видны, потому что она завершилась до создания снимка;
- изменения транзакции 1 не будут видны, потому что она была активна на момент создания снимка;
- изменения транзакции 3 не будут видны, потому что она

началась позже создания снимка (не важно, закончилась она или нет).

Из чего состоит снимок

К сожалению, PostgreSQL видит картину совсем не так, как было показано на рисунке. Дело в том, что системе неизвестно, когда транзакции были зафиксированы. Известно только, когда они начинались (этот момент определяется номером транзакции), а вот факт завершения нигде не записывается.

Время фиксации отслеживается при включенном параметре track_commit_timestamp, off но оно никак не участвует в проверке видимости (хотя может быть полезным, например, для сторонних репликационных решений).

Кроме того, PostgreSQL всегда сохраняет время фиксации и время отмены транзакций в соответствующих журнальных записях, но использует эту информацию только при восстановлении до целевой точки.

Узнать можно лишь текущий статус транзакций. Эта информация есть в общей памяти сервера в структуре ProcArray, которая содержит список всех активных сеансов и их транзакций. Постфактум же невозможно выяснить, была ли какая-то транзакция активна в момент создания снимка.

Поэтому для получения снимка недостаточно сохранить момент его создания: требуется также запомнить, в каком статусе находились транзакции на этот момент. Без информации о статусах впоследствии невозможно будет понять, какие версии строк должны быть видны в снимке, а какие — нет.

Сравните, какая информация доступна системе в момент создания снимка и некоторое время спустя (белым кружком обозначена активная транзакция, а черными — завершенные):

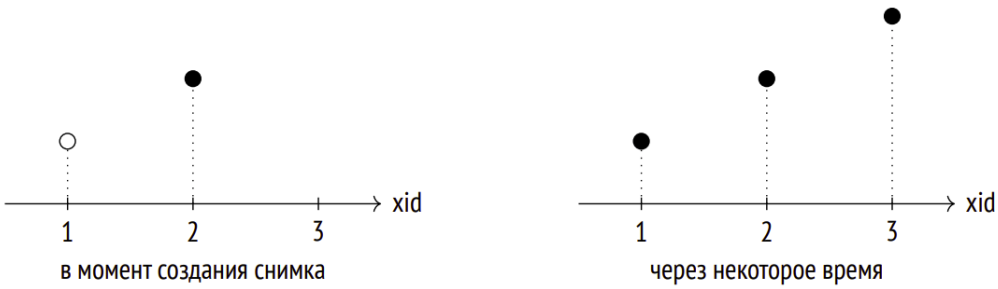

Рисунок 3 – информация в системе в момент создания снимка и некоторое время спустя

Если не запомнить, что во время создания снимка первая транзакция еще выполнялась, а третьей не существовало, их невозможно будет отличить от зафиксированной на тот момент второй транзакции и исключить из рассмотрения.

По этой причине в PostgreSQL нельзя создать снимок, показывающий согласованные данные по состоянию на некоторый момент в прошлом, даже если все необходимые для этого версии строк существуют в табличных страницах. Соответственно, невозможно реализовать и ретроспективные (темпоральные, flashback) запросы.

Интересно, что такая функциональность была заявлена как одна из целей Postgres и была реализована с самого начала, но ее убрали из СУБД, когда проект был взят на поддержку сообществом.

Итак, снимок данных состоит из нескольких значений, которые запоминаются в момент его создания:

- Нижняя граница снимка xmin, в качестве которой выступает номер самой старой активной транзакции. Все транзакции с меньшими номерами либо зафиксированы, и тогда их изменения видны в снимке, либо отменены, и тогда изменения игнорируются.
- Верхняя граница снимка xmax, в качестве которой берется значение, на единицу большее номера последней зафиксированной транзакции. Верхняя граница определяет момент времени, в который был сделан снимок. Все транзакции с номерами, большими или равными xmax, не завершены или не существуют, и поэтому изменения таких транзакций точно не видны.
- Список активных транзакций xip_list (xid-in-progress list), в который попадают номера всех активных транзакций, за исключением виртуальных, которые никак не влияют на видимость.

Также в снимке сохраняются еще несколько параметров, но они пока не так важны. Графически можно представить снимок как прямоугольник, охватывающий транзакции от xmin до xmax:

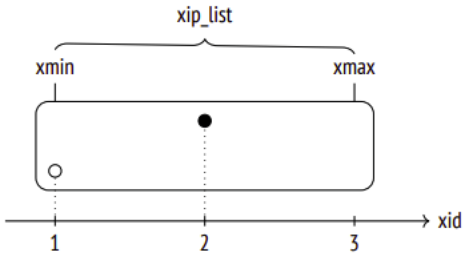

Рисунок 4 – снимок данных

Чтобы посмотреть, как видимость определяется снимком, воспроизведем ситуацию по показанному выше сценарию на таблице accounts.

```
=> TRUNCATE TABLE accounts;
```

Первая транзакция вставляет в таблицу первую строку и остается активной:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (1, 'alice', 1000.00);
=> SELECT pg_current_xact_id();
pg_current_xact_id
--------------------
790
(1 row)
```

Вторая транзакция вставляет вторую строку и сразу завершается:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (2, 'bob', 100.00);
=> SELECT pg_current_xact_id();
pg_current_xact_id
--------------------
791
(1 row)
=> COMMIT;
```

В этот момент (в другом сеансе) создаем новый снимок. Для этого достаточно выполнить любой запрос, но мы сразу посмотрим на снимок с

помощью специальной функции:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> -- txid_current_snapshot() до v.13
SELECT pg_current_snapshot();
pg_current_snapshot
--------------------
790:792:790
(1 row)
```

Функция выводит через двоеточие три составляющих снимка: поля xmin, xmax и список xip_list (состоящий в данном случае из одного номера). После того как снимок создан, завершаем первую транзакцию:

```
=> COMMIT;
```

Третья транзакция выполняется после создания снимка и меняет вторую строку. Появляется новая версия:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> SELECT pg_current_xact_id();
pg_current_xact_id
-------------------
792
(1 row)
=> COMMIT;
```

В созданном снимке видна только одна версия:

```
=> SELECT ctid, * FROM accounts;
ctid | id | client | amount
-------+----+--------+--------
(0,2) | 2 | bob | 100.00
(1 row)
```

Хотя в таблице их три:

```
=> SELECT * FROM heap_page('accounts',0);
ctid | state | xmin | xmax
-------+--------+-------+-------
(0,1) | normal | 790 c | 0 a
(0,2) | normal | 791 c | 792 c
```

```
(0,3) | n
```

Как PostgreSQL понимает, какие версии показывать? По сформулированным выше правилам в снимке видны изменения только следующих зафиксированных транзакций:

- с номерами xid < xmin — безусловно (например, транзакция, создавшая таблицу accounts);
- с номерами xmin ⩽ xid < xmax — за исключением попавших в список xip_list.

Первая строка (0,1) не видна, так как номер создавшей ее транзакции входит в список активных транзакций (хотя и попадает при этом в диапазон снимка).

Последняя версия второй строки (0,3) не видна, поскольку создана транзакцией с номером, выходящим за верхнюю границу снимка.

Зато видна первая версия второй строки (0,2): номер создавшей ее транзакции попадает в диапазон снимка, но не входит в список активных транзакций (вставка видна), и при этом номер удалившей транзакции выходит за верхнюю границу снимка (удаление не видно).

```
=> COMMIT;
```

Видимость собственных изменений

Картину несколько усложняет определение видимости собственных изменений транзакции, поскольку возможна ситуация, при которой должна быть видна только часть из них. Например, курсор, открытый в определенный момент, ни при каком уровне изоляции не может видеть последующие изменения.

Для этого в заголовке версии строки есть специальное поле (которое отображается в псевдостолбцах cmin и cmax), показывающее порядковый номер операции внутри транзакции. Столбец cmin представляет номер операции вставки, а cmax — операции удаления, но для экономии места в заголовке строки это на самом деле одно поле, а не два разных. Считается, что вставка и удаление той же строки в одной транзакции выполняются редко. (Если это все-таки происходит, то в то же самое поле вставляется специальный «комбо-номер», для которого обслуживающий процесс запоминает реальные

cmin и cmax.)

В качестве примера начнем транзакцию и вставим в таблицу строку:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (3, 'charlie', 100.00);
=> SELECT pg_current_xact_id();
pg_current_xact_id
--------------------
793
(1 row)
```

Теперь откроем курсор для запроса, возвращающего число строк в таблице.

```
=> DECLARE c CURSOR FOR SELECT count(*) FROM accounts;
```

И после этого вставим еще одну строку:

```
=> INSERT INTO accounts VALUES (4, 'charlie', 200.00);
```

Выведем содержимое таблицы, добавив столбец cmin для строк, вставленных нашей транзакцией (для других строк это значение не имеет смысла):

```
=> SELECT xmin, CASE WHEN xmin = 793 THEN cmin END cmin, *
FROM accounts;
xmin | cmin | id | client | amount
------+------+----+---------+---------
790 | | 1 | alice | 1000.00
792 | | 2 | bob | 200.00
793 | 0 | 3 | charlie | 100.00
793 | 1 | 4 | charlie | 200.00
(4 rows)
```

Запрос курсора обнаружит три строки, а не четыре — строка, добавленная после открытия курсора, не попадет в снимок данных, поскольку в нем учитываются только версии строк с cmin < 1:

```
=> FETCH c;
count
-------
3
(1 row)
```

Разумеется, этот номер cmin также запоминается в снимке, но вывести его средствами SQL не получится.

Горизонт транзакции

Для транзакции, работающей со снимком, нижняя граница xmin этого снимка (номер самой ранней транзакции, активной на момент его создания) имеет важный смысл — она определяет горизонт транзакции.

Горизонт транзакции без активного снимка (например, транзакции с уровнем изоляции Read Committed между выполнениями операторов) определяется ее собственным номером, если он ей присвоен.

Все транзакции, находящиеся за горизонтом (то есть транзакции с номерами xid < xmin), уже гарантированно зафиксированы. А это значит, что за своим горизонтом транзакция всегда видит только актуальные версии строк (название навеяно, конечно, понятием горизонта событий в физике). PostgreSQL знает текущий горизонт транзакций всех процессов; собственный горизонт транзакция может увидеть в таблице pg_stat_activity:

```
=> BEGIN;
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
backend_xmin
--------------
793
(1 row)
```

Виртуальные транзакции хоть и не имеют настоящего номера, но используют снимки точно так же, как и обычные транзакции, и поэтому обладают собственным горизонтом. Исключение составляют виртуальные транзакции без активного снимка: для них понятие горизонта не имеет смысла, и они полностью «прозрачны» для системы с точки зрения снимков данных и видимости (несмотря на то что в pg_stat_activity.backend_xmin может остаться номер от старого снимка).

Похожим образом можно определить и горизонт базы данных. Для этого надо взять горизонты всех транзакций, работающих с этой базой, и среди них

найти наиболее «дальний», имеющий самый старый xmin. Это и будет тот горизонт, за которым неактуальные версии строк в этой базе данных уже никогда не будут видны ни одной транзакции. Такие версии строк могут быть безопасно удалены очисткой— именно поэтому понятие горизонта так важно с практической точки зрения.

Сделаем выводы из сказанного:

• если транзакция с уровнем изоляции Repeatable Read или Serializable (не важно, настоящая или виртуальная) выполняется долго, она тем самым удерживает горизонт базы данных и препятствует очистке;

• настоящая транзакция с уровнем изоляции Read Committed точно так же удерживает горизонт базы данных, выполняет ли она оператор или просто бездействует (находится в состоянии idle in transaction);

• виртуальная транзакция с уровнем изоляции Read Committed удерживает горизонт только в процессе выполнения операторов.

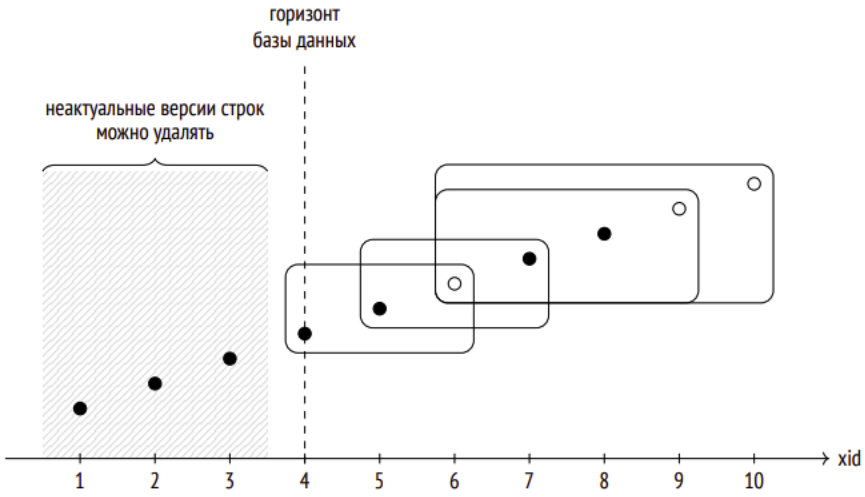

Рисунок 5 – горизонт транзакции

При этом на всю базу данных есть только один горизонт, и, если какая-то транзакция его удерживает — она не дает очищать данные внутри этого горизонта, даже те, к которым не обращалась.

Для общекластерных таблиц системного каталога используется отдельный горизонт, учитывающий транзакции во всех базах данных. А для временных таблиц, наоборот, не нужно учитывать никакие транзакции, кроме

выполняющихся в текущем процессе.

Продолжим пример. Транзакция в первом сеансе до сих пор выполняется и продолжает удерживать горизонт, что можно проверить, увеличив счетчик транзакций:

```
=> SELECT pg_current_xact_id();
pg_current_xact_id
--------------------
794
(1 row)

=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
backend_xmin
--------------
793
(1 row)
```

И только после завершения транзакции горизонт продвигается вперед, позволяя очистке удалить неактуальные версии строк:

```
=> COMMIT;
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
backend_xmin
--------------
795
(1 row)
```

В идеале не следует совмещать активные обновления (порождающие версии строк) с долгими транзакциями, поскольку это будет приводить к разрастанию таблиц и индексов.

Снимок данных для системного каталога

Хотя системный каталог и представлен обычными таблицами, при обращении к ним нельзя задействовать тот же снимок данных, что используется транзакцией или оператором. Снимок должен быть достаточно «свежим», чтобы включать все последние изменения, ведь иначе транзакция

могла бы увидеть уже неактуальное определение столбцов таблицы или пропустить созданное ограничение целостности.

Вот простой пример:

```
=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
=> SELECT 1; -- создан снимок для транзакции

=> ALTER TABLE accounts ALTER amount SET NOT NULL;

=> INSERT INTO accounts(client, amount) VALUES ('alice', NULL);
ERROR: null value in column "amount" of relation "accounts"
violates not-null constraint
DETAIL: Failing row contains (1, alice, null).
=> ROLLBACK;
```

Команда INSERT «увидела» ограничение целостности, которое появилось уже после того, как был создан снимок данных транзакции. Может показаться, что такое поведение нарушает изоляцию, но если бы транзакция успела обратиться к таблице accounts, то команда ALTER TABLE была бы заблокирована до окончания этой транзакции.

В целом система ведет себя так, как будто для каждого обращения к системному каталогу создается новый снимок. Но реализация, конечно, более сложна, поскольку постоянное создание новых снимков негативно сказалось бы на производительности; кроме того, многие объекты системного каталога кешируются, и это тоже надо учитывать.

Экспорт снимка данных

Бывают ситуации, когда несколько параллельных транзакций должны гарантированно видеть одну и ту же картину данных. В качестве примера можно привести утилиту pg_dump, умеющую работать в параллельном режиме: все рабочие процессы должны видеть базу данных в одном и том же состоянии, чтобы резервная копия получилась согласованной.

Разумеется, нельзя полагаться на то, что картины данных совпадут просто потому, что транзакции запущены «одновременно». Такие гарантии дает механизм экспорта и импорта снимка.

Функция pg_export_snapshot возвращает идентификатор снимка, который может быть передан в другую транзакцию (внешними по отношению к СУБД средствами):

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT count(*) FROM accounts;
count
-------
4
(1 row)
=> SELECT pg_export_snapshot();
pg_export_snapshot
--------------------
00000004-0000006E-1
(1 row)
```

Перед тем как выполнить первый оператор, другая транзакция может импортировать снимок с помощью команды SET TRANSACTION SNAPSHOT. Предварительно надо установить уровень изоляции Repeatable Read или Serializable, потому что на уровне Read Committed операторы будут использовать собственные снимки:

```
=> DELETE FROM accounts;
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SET TRANSACTION SNAPSHOT '00000004-0000006E-2';
```

Теперь вторая транзакция будет работать со снимком первой и, соответственно, видеть четыре строки (а не ноль):

```
=> SELECT count(*) FROM accounts;
count
-------
4
(1 row)
```

Разумеется, изменения, сделанные первой транзакцией после экспорта снимка, не будут видны второй транзакции (и наоборот): действуют обычные правила видимости.

Время жизни экспортированного снимка совпадает со временем жизни

экспортирующей транзакции.

```
=> COMMIT;
```

```
=> COMMIT;
```

Задание на практическую работу:

Реализуйте работу со снимками данных в параллельных транзакциях с уровнями изоляции Repeatable Read и Serializable.

Уровень Serializable:

1. В первом сеансе начните новую транзакцию с уровнем изоляции Serializable. Вычислите количество заказов с суммой 20 000 рублей. Добавьте новый заказ на 30 000 рублей. Получите снимок транзакции.

2. Во втором сеансе начните новую транзакцию с уровнем изоляции Serializable. Импортируйте снимок из первой транзакции. Вычислите количество заказов с суммой 30 000 рублей.

3. В первом сеансе вычислите количество заказов с суммой 20 000 рублей. Вычислите количество заказов с суммой 30 000 рублей.

4. Во втором сеансе добавьте новый заказ на 20 000 рублей и снова вычислите количество заказов с суммой 30 000 рублей. Вычислите количество заказов с суммой 20 000 рублей.

5. Зафиксируйте транзакции в обоих сеансах.

Соответствует ли результат ожиданиями? Что вам дал импорт снимка из первой транзакции?

Уровень Repeatable Read:

1. В первом сеансе начните новую транзакцию с уровнем изоляции Repeatable Read. Вычислите количество заказов с суммой 20 000 рублей. Добавьте новый заказ на 30 000 рублей. Получите снимок транзакции.

2. Во втором сеансе начните новую транзакцию с уровнем изоляции Repeatable Read. Импортируйте снимок из первой транзакции. Вычислите

количество заказов с суммой 30 000 рублей.

3. В первом сеансе вычислите количество заказов с суммой 20 000 рублей. Вычислите количество заказов с суммой 30 000 рублей.

4. Во втором сеансе добавьте новый заказ на 20 000 рублей и снова вычислите количество заказов с суммой 30 000 рублей. Вычислите количество заказов с суммой 20 000 рублей.

5. Зафиксируйте транзакции в обоих сеансах.

Соответствует ли результат ожиданиями? Что вам дал импорт снимка из первой транзакции?