БАЗЫ ДАННЫХ

Лекция 7

ПЛАН НА СЕГОДНЯ

- Изоляция в Postgres глазами пользователя

РАБОЧАЯ ТАБЛИЦА

- Для экспериментов создадим таблицу
- И слегка заполним

```
CREATE TABLE accounts(
id integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
client text,
amount numeric
);

INSERT INTO accounts VALUES
(1, 'alice', 1000.00), (2, 'bob', 100.00), (3, 'bob', 900.00);
```

READ COMMITTED

- SHOW default_transaction_isolation;
- В норме - read committed
- Можно начать транзакцию и посмотреть

```
BEGIN;
SHOW transaction_isolation;
```

ОБНОВИМ ТАБЛИЦУ

- Обновим таблицу
- И пока останемся в транзакции

```
UPDATE accounts SET amount = amount - 200 WHERE id = 1;
SELECT * FROM accounts WHERE client = 'alice';
```

ДРУГОЙ СЕАНС

- Заходим в транзакцию
- Читаем данные

```
BEGIN;
SELECT * FROM accounts WHERE client = 'alice';
```

NON-REPEATABLE

- COMMIT в первом сеансе
- Читаем данные во втором

```
SELECT * FROM accounts WHERE client = 'alice';
```

ПРАКТИЧЕСКИЕ СООБРАЖЕНИЯ

- Классический анти-паттерн
- (Псевдокод)

```
IF (SELECT amount FROM accounts WHERE id = 1) >= 1000 THEN
UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
END IF;
```

ВАРИАНТЫ РЕШЕНИЯ

- Перейти на Repeatable Read - заплатим производительностью
- Ситуативный вариант -

```
ALTER TABLE accounts ADD CHECK amount >= 0;
```

- Еще ситуативный - уложиться в один запрос/CTE
- Поставить локи на строки или на таблицу - заплатим производительностью

ЕЩЕ АНОМАЛИЯ

- Read Skew - не описана в стандарте
- Пример построен на "неправильном" паттерне чтения
- Что не отменяем факта аномалии
- И того, что могут быть причины читать "неправильным" способом

СЦЕНАРИЙ

- Одна транзакция переводит средства со счета на счет

```
BEGIN;
    UPDATE accounts SET amount = amount - 100 WHERE id = 2;
    UPDATE accounts SET amount = amount + 100 WHERE id = 3;
COMMIT;
```

СЦЕНАРИЙ

- Другая считает баланс у каждого и суммирует

```
BEGIN;
    SELECT amount FROM accounts WHERE id = 2;
    SELECT amount FROM accounts WHERE id = 3;
COMMIT;
```

РАЗБЕРЕМ

- Первый SELECT успевает до COMMIT-а первой транзакции
- Второй работает после, 100 посчитаем дважды
- Схема схожа с Non-Repeatable Read, но это отдельная проблема
- Конкретно здесь лечится агрегацией в одном запросе

ЗАТЯНЕМ ЗАПРОС

- Используем pg_sleep() в SELECT, чтобы затянуть обработку строк
- Приготовим транзакцию во втором сеансе
- Запустим SELECT, быстро сделаем COMMIT

```
SELECT amount, pg_sleep(10) FROM accounts WHERE client = 'bob';
```

ИСПОЛЬЗУЕМ ФУНКЦМЮ

- VOLATILE-функцию с подзапросом

```
CREATE FUNCTION get_amount(id integer) RETURNS numeric
AS $$
SELECT amount FROM accounts a WHERE a.id = get_amount.id;
$$ VOLATILE LANGUAGE sql;
SELECT get_amount(id), pg_sleep(10) FROM accounts WHERE client =
```

КОГДА ТАКОЕ БЫВАЕТ

- Если уровень изоляции - Read Committed
- И функция VOLATILE
- То есть в настройках по умолчанию

ЕЩЕ СЦЕНАРИЙ

- Снова у Боба два счета с общей суммой 1000
- Боб в одной транзакции переводит средства, не себе
- В другой транзакция проходит акция - бонус 1 процент тем, у кого на счетах не меньше, чем 1000

ВТОРОЙ ЗАПРОС

```
UPDATE accounts SET amount = amount * 1.01
WHERE client IN (
SELECT client
FROM accounts
GROUP BY client
HAVING sum(amount) >= 1000
);
```

ЧТО ПРОИСХОДИТ

- Выбираются строки для обновления
- Коммита нет - поэтому опираемся на "старое" состояние
- Боб подпадает под критерий
- Начинаем обновлять строки

ЧТО ПРОИСХОДИТ

- Возникает заминка с локом на id = 3
- INSERT не может закончиться
- Тем временем случается COMMIT первой транзакции
- Вторая читает новое значение
- Случается то, что не должно было случиться

КАК ПОТЕРЯТЬ ОБНОВЛЕНИЯ

- Две транзакции начинают с того, что читают баланс Алисы

```
SELECT amount FROM accounts WHERE id = 1;
```

- Сохраняют его где-то у себя
- Обе увеличивают значение на 100 и обновляют:

```
 UPDATE accounts SET amount = 800.00 + 100 WHERE id = 1
RETURNING amount;
```

РЕЗУЛЬТАТ

- Обе видят 900 как результат
- Тут неправ тот, кто пишет такие запросы
- Но в Repeatable Read такого нет

REPEATABLE READ

- Снова две транзакции
- В первой все по-старому
- Во второй меняем уровень изоляции

ПЕРВЫЙ СЕАНС

```
BEGIN;
    SELECT * FROM accounts ORDER BY id;

    UPDATE accounts SET amount = 200.00 WHERE id = 2;
    UPDATE accounts SET amount = 800.00 WHERE id = 3;
    INSERT INTO accounts VALUES

    SELECT * FROM accounts ORDER BY id;
```

ВТОРОЙ СЕАНС

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
    SELECT * FROM accounts ORDER BY id;

    -- commit в первой

    SELECT * FROM accounts ORDER BY id;

COMMIT

SELECT * FROM accounts ORDER BY id;
```

ПОВТОРИМ ЭКСПЕРИМЕНТЫ С ОБНОВЛЕНИЕМ

ПОВТОРИМ ЭКСПЕРИМЕНТЫ С ОБНОВЛЕНИЕМ

```
-- повисим, пока не закомитимся в первой

-- получим ошибку

ROLLBACK

SELECT * FROM accounts WHERE client = 'bob';
```

ДВОЙНАЯ ЗАПИСЬ

- Если обе RepeatableRead - одна запишется
- В другой будет ошибка сериализации
- Если одна RepeatableRead, другая ReadCommited, зависит от того, кто первый
- Если на локе повиснет RC - потеряем обновление

TRADE-OFF

- RR дает бОльшую надежность
- В обмен получаем ненулевую вероятность неудавшейся записи
- Но только записи
- Запросам на чтение это не грозит

WRITE SKEW

- Зададим правило: если общий баланс положительный, то на отдельных счетах можно иметь негативный баланс
- Первая транзакция получает общий баланс счетов Боба

```
BEGIN ISOLATION LEVEL REPEATABLE READ;

SELECT sum(amount) FROM accounts WHERE client = 'bob';
```

WRITE SKEW

- Вторая делает то же
- Первая списывает деньги с одного из счетов

```
UPDATE accounts SET amount = amount - 600.00 WHERE id = 2;
```

- А вторая - с другого
- COMMIT - и видим нарушение

ЕШЕ АНОМАЛИЯ

- Три транзакции: две обновляющие, одна только читает
- Первая считает проценты по всем счетам Боба
- И зачисляет их на один из счетов
- Вторая списывает средства с другого счета

ЕШЕ АНОМАЛИЯ

```
BEGIN ISOLATION LEVEL REPEATABLE READ;

UPDATE accounts SET amount = amount + (
SELECT sum(amount) FROM accounts WHERE client = 'bob'
) * 0.01
WHERE id = 2;
```

ЕШЕ АНОМАЛИЯ

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
 UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
COMMIT;
```

ЧТО ДАЛЬШЕ

- Вот закомититься бы первой транзакции - и все ОК
- Но тут третья транзакция - и начинаются неприятности
- Обратим внимание на взаимоотношение первых двух
- Как эта пара сериализуется

СЕРИАЛИЗАЦИЯ

- Вторая закончилась, первая - нет
- Но вторую нельзя сериализовать перед первой
- Потому что она меняет данные, на которых первая базировалась
- А первую перед второй - можно

СЕРИАЛИЗАЦИЯ

- И тут приходит третья - читающая
- И она должна видеть итоги второй - как закомиченной
- В итоги первой - нет

ЕШЕ АНОМАЛИЯ

```
BEGIN ISOLATION LEVEL REPEATABLE READ;

SELECT * FROM accounts WHERE client = 'alice';

-- COMMIT в первой

SELECT * FROM accounts WHERE client = 'bob';

COMMIT;
```

SERIALIZABLE

- На этом уровне не происходит аномалий
- Но можем получать ошибки
- Попробуем write-skew

WRITE SKEW

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
SELECT sum(amount) FROM accounts WHERE client = 'bob';

--  Вторая

BEGIN ISOLATION LEVEL SERIALIZABLE;
SELECT sum(amount) FROM accounts WHERE client = 'bob';
```

WRITE SKEW

```
UPDATE accounts SET amount = amount - 600.00 WHERE id = 2;

--  Вторая

UPDATE accounts SET amount = amount - 600.00 WHERE id = 3;
COMMIT;
```

ОТКЛАДЫВАЕМАЯ ТРАНЗАКЦИЯ

- Ключевое слово DEFERABLE
- Исполнение откладывается до безопасного момента

```
UPDATE accounts SET amount = 900.00 WHERE id = 2;
UPDATE accounts SET amount = 100.00 WHERE id = 3;
SELECT * FROM accounts WHERE client = 'bob' ORDER BY id;
```

ПРИМЕР

```
BEGIN ISOLATION LEVEL SERIALIZABLE; -- 1

UPDATE accounts SET amount = amount + (
SELECT sum(amount) FROM accounts WHERE client = 'bob'
) * 0.01
WHERE id = 2;

-- вторая

BEGIN ISOLATION LEVEL SERIALIZABLE; -- 2
UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
COMMIT;
```

ПРИМЕР

```
BEGIN ISOLATION LEVEL SERIALIZABLE READ ONLY DEFERRABLE; -- 3
SELECT * FROM accounts WHERE client = 'alice';

-- подвисаем

--  в первой
COMMIT;
```

SERIALIZABLE

- SERIALIZABLE имеет смысл, только если все транзакции такие
- Если нет - она превращается в RR
- default_trabsaction_isolation - задает уровень по умолчанию

TRADE-OFF

- В конкретном приложении выбираем уровень изоляции
- Баланс гарантий и издержек
- По умолчанию RC

SERIALIZABLE

- Поля xmin, xmax у каждой записи
- infomask - маски записи
- `src/include/access/htup_details.h`
- `src/backend/access/transam/README`

INSERT

```
CREATE TABLE t(
id integer GENERATED ALWAYS AS IDENTITY,
s text
);
CREATE INDEX ON t(s);

BEGIN;
INSERT INTO t(s) VALUES ('hello');
SELECT pg_current_xact_id();

SELECT *
FROM heap_page_items(get_raw_page('t',0)) \gx
```

INSERT

```
SELECT '(0,'||lp||')' AS ctid,
CASE lp_flags
WHEN 0 THEN 'unused'
WHEN 1 THEN 'normal'
WHEN 2 THEN 'redirect to '||lp_off
WHEN 3 THEN 'dead'
END AS state,
t_xmin as xmin,
t_xmax as xmax,
(t_infomask & 256) > 0 AS xmin_committed,
(t_infomask & 512) > 0 AS xmin_aborted,
(t_infomask & 1024) > 0 AS xmax_committed,
(t_infomask & 2048) > 0 AS xmax_aborted
FROM heap_page_items(get_raw_page('t',0)) \gx
```

```
CREATE FUNCTION heap_page(relname text, pageno integer)
RETURNS TABLE(ctid tid, state text, xmin text, xmax text)
AS $$
SELECT (pageno,lp)::text::tid AS ctid,
CASE lp_flags
WHEN 0 THEN 'unused'
WHEN 1 THEN 'normal'
WHEN 2 THEN 'redirect to '||lp_off
WHEN 3 THEN 'dead'
END AS state,
t_xmin || CASE
WHEN (t_infomask & 256) > 0 THEN ' c'
WHEN (t_infomask & 512) > 0 THEN ' a'
ELSE ''
END AS xmin,
t_xmax || CASE
WHEN (t_infomask & 1024) > 0 THEN ' c'
WHEN (t_infomask & 2048) > 0 THEN ' a'
ELSE ''
END AS xmax
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

INSERT

```
SELECT * FROM heap_page('t',0);
SELECT xmin, xmax, * FROM t;
```

СОСТОЯНИЕ ТРАНЗАКЦИИ

- Транзакция читает записи страницы
- Видим xmin
- Надо понять, завершена она или нет
- Если нет - пропускаем

СОСТОЯНИЕ ТРАНЗАКЦИИ

- Первым делом смотрим в структуру ProcArray
- Хранит данные о текущих клиентских процессах
- И их активных транзакциях
- Если там нет, она завершена

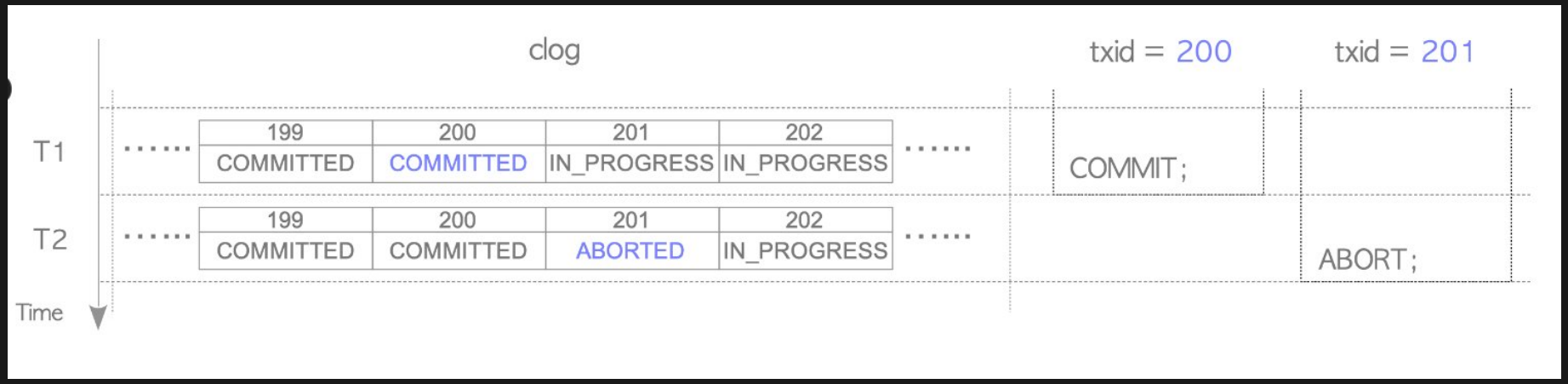
СОСТОЯНИЕ ТРАНЗАКЦИИ

- Если завершена, надо понять, в каком состоянии
- Это можно узнать в commit-логе (CLOG)
- CLOG - это что-то вроде массива
- Индекс - номер транзакции

СОСТОЯНИЕ ТРАНЗАКЦИИ

- Хвост массива кешируется
- В целом он хранится на диске
- Там можно узнать про каждую завершенную транзакцию
- Как она завершилась

POSTGRES


clog
txid = 200
txid = 201
T1
199
200
201
202
COMMITTED
COMMITTED
IN_PROGRESS
IN_PROGRESS
COMMIT;
T2
199
200
201
202
COMMITTED
COMMITTED
ABORTED
IN_PROGRESS
ABORT;
Time

СОСТОЯНИЕ ТРАНЗАКЦИИ

- Для свежих быстро поймем статус
- Для менее свежих может быть дороговато
- И как-то хотелось бы прорежать CLOG
- Не растить его вечно

СОСТОЯНИЕ ТРАНЗАКЦИИ

- Для этого пишем в запись флаги состояния создавшей ее транзакции
- Можем так делать в силу монотонности
- Но не всегда делаем
- Поэтому два отдельных флага
- Чтобы отличать состояние "данных пока нет"

СОСТОЯНИЕ ТРАНЗАКЦИИ

- При самом INSERT-е пока оставляем флаги нетронутыми
- Потому что внутри самой INSERT-транзакции мы не предскаазываем будущее
- И не знаем, чем транзакция завершится
- А обновлять флаги отдельным шагом может быть неудобно

СОСТОЯНИЕ ТРАНЗАКЦИИ

- В Postgres последующий SELECT может обновить флаги
- Т.е. SELECT может оставить грязную страницу
- А обновлять флаги отдельным шагом может быть неудобно

ПРОДОЛЖИМ

```
COMMIT;
SELECT * FROM heap_page('t',0);

SELECT * FROM t;
SELECT * FROM heap_page('t',0);
```

УДАЛЕНИЕ

- В поле xmax прописываем id удаляющей транзакции
- И есть парные флаги для "кеширования" данных о состоянии
- И тоже в самой транзакции их еще не ставим
- Это сделает следующий запрос

ПРОДОЛЖИМ

```
BEGIN;
  DELETE FROM t;
  SELECT pg_current_xact_id();
  SELECT * FROM heap_page('t',0);
  ROLLBACK;
  SELECT * FROM heap_page('t',0);
  SELECT * FROM t;
  SELECT * FROM heap_page('t',0);
END;
```

UPDATE

- Композиция INSERT + DELETE
- По стоимости - зависит от того, что обновляется
- Худший вариант - много небольших полей
- Но не настолько много, чтобы что-то серьезное уехало в TOAST
- И изменяются 1-2 поля

UPDATE

- Если немного маленьких полей
- И большие, лежащие в TOAST
- И они не меняются
  - Будет OK

ПРОДОЛЖИМ

```
BEGIN;
 UPDATE t SET s = 'BAR';
 SELECT pg_current_xact_id();
 SELECT * FROM t;
 SELECT * FROM heap_page('t',0);
COMMIT;
```

ИНДЕКСЫ

- Индексы - это тоже таблицы
- Но своих полей xmin/xmax у них нет
- Есть ссылки на страницы, xmin/xmax берем там
- Это снижает нагрузку на изменение индекса
- Но она есть

ПРОДОЛЖИМ

```
CREATE FUNCTION index_page(relname text, pageno integer)
RETURNS TABLE(itemoffset smallint, htid tid)
AS $$
SELECT itemoffset,
htid -- ctid before v.13
FROM bt_page_items(relname,pageno);
$$ LANGUAGE sql;

SELECT * FROM index_page('t_s_idx',1);
```

ВИРТУАЛЬНЫЕ ТРАНЗАКЦИИ