Концепции PostgreSQL

Alexander Korotkov

Postgres Professional

PGDay.RU 2015

Исходные концепции postgres

Исходные концепции, заложенные в postgres, как исследовательский проект Berkeley.

- Поддержка сложных типов данных
- Расширяемость
- Rules (правила)
- Heap вместо undo/redo логов
- Элементы объектности

Stonebraker M., Rowe L. A., Hirohama M. The implementation of POSTGRES //IEEE Transactions on Knowledge & Data Engineering. – 1990. – №. 1. – С. 125-142.

Поддержка сложных типов данных

Postgres поддерживает следующие типы данных by-design:

1. базовые типы данных,
2. композитные типы данных,
3. массивы.

2 + 3 = документоориентированное хранение (в 1990 году!).

Поддержка сложных типов данных получила своё дальнейшее развитие: hstore, xml, json, jsonb.

Расширяемость

It is imperative that a user be able to construct new access methods to provide efficient access to instances of nontraditional base types Michael Stonebraker, Jeff Anton, Michael Hirohama.

Extendability in POSTGRES , IEEE Data Eng. Bull. 10 (2) pp.16-23, 1987

- Типы данных
- Функции
- Процедурные языки
- Операторы
- Классы операторов (operator classes)
- Методы доступа (access methods)

Расширяемость методов доступа в настоящий момент не полная. Не хватает команды CREATE ACCESS METHOD и расширяемости WAL.

Rules

Было:

```
on new EMP.salary where EMP.name = ”Fred”
then do replace
E (salary = new.salary) from E in EMP
where E.name = ”Joe”
```

Стало:

```
CREATE RULE emp_insert AS ON insert TO emp
WHERE NEW.name = ’Fred’
DO ALSO UPDATE emp e SET salary = NEW.salary
WHERE e.name = ’Joe’;
```

- Низкий overhead: нет триггера, переписывается сам запрос.
- Не прозрачно для пользователя, можно словить глюки.

Как работают rules?

```
# EXPLAIN (COSTS off) INSERT INTO
  emp (name, salary) VALUES ('Fred', 1000);
                QUERY PLAN
----------------------------------------
 Insert on emp
   ->  Result

 Update on emp e
   Update on emp e
   Update on salesman e_1
   ->  Seq Scan on emp e
         Filter: (name = 'Joe'::bpchar)
   ->  Seq Scan on salesman e_1
         Filter: (name = 'Joe'::bpchar)
```

Непредсказуемые rules

```
# CREATE TABLE t1 (val float8);
# CREATE TABLE t2 (val float8);
# CREATE RULE t1_insert AS ON insert TO t1
  DO ALSO INSERT INTO t2 VALUES (NEW.val);
# INSERT INTO t1 VALUES (random());
# SELECT * FROM t1;
        val
--------------------
 0.104912102222443
(1 row)
# SELECT * FROM t2;
        val
--------------------
 0.835614582523704
(1 row)
```

Выпилить rules?

Simon Riggs предлагал начать процесс выпиливания rules, т.к. появились триггеры на view:
http://www.postgresql.org/message-id/CA+U5nMLzz7MPud6zRvw4gbq66FbSjqLYNB+UL=1sXTUS-a97fA@mail.gmail.com.

Но не получилось, потому что:

- Есть пользователи, которые используют rules.
- Rules используются внутри для реализации обычных view.
- Некоторые считают, что rules – это не так уж и плохо.

Элементы объектности: наследование

Было:

```
create EMP (name = c12, dept = DEPT, salary = float)
create SALESMAN (quota = float) inherits (EMP)
```

Стало:

```
CREATE TABLE emp (
    id SERIAL PRIMARY KEY, name CHAR(12),
    dept_id INTEGER REFERENCES dept(id), salary FLOAT);
CREATE TABLE salesman (quota FLOAT) INHERITS (emp);
```

По прямому назначению наследование в PostgreSQL применяют редко, зато через него сделан partitioning.

Элементы объектности: методы (было)

```
create DEPT (
    dname = c10,
    floor = integer,
    floor-space = polygon)
define function set-of-DEPT as retrieve (DEPT. all)
    where DEPT.floor = $.floor
create FLOORS (
    floor = integer,
    depts = set-of-DEPT)
```

Элементы объектности: методы (стало)

```
CREATE TABLE floor (floor INTEGER PRIMARY KEY);
CREATE TABLE dept (
    id SERIAL PRIMARY KEY, dname CHAR(10),
    floor INTEGER REFERENCES floor(floor),
    ”floor-space” POLYGON);
CREATE FUNCTION ”set-of-DEPT”(floor) RETURNS SETOF dept
AS $$
    SELECT * FROM dept WHERE floor = $1.floor;
$$ LANGUAGE sql;
# SELECT f.floor, f.”set-of-DEPT” FROM floor f;
 floor |                    set-of-DEPT
-------+-------------------------------------------------
     1 | (1,”sales     ”,1,”((0,0),(1,1),(2,2),(2,1))”)
     1 | (2,”marketing ”,1,”((4,4),(5,6),(3,4))”)
(2 rows)
```

Элементы объектности: методы (как это работает?)

- Запись tab.func равносильна записи func(tab). (Напоминает a[i] и i[a] в C).
- Можно не только работать с функциями, как с колонками, но и с колонками, как с функциями.

```
# SELECT id(dept), dname(dept) FROM dept;
 id |   dname
----+-----------
  1 | sales
  2 | marketing
(2 rows)
```

Heap: что было

Update не делается in-place, заводится новая версия строки в heap. Delete помечает строку как удалённую.

- Не используются undo/redo логи.
- Возможен time-travel.
- Необходимость VACUUM.

Без redo лога

Теоретически – возможно, если...

- Иметь полный контроль за тем, что и когда фактически пишется на диск.
- В конце каждой транзакции фиксировать все изменённые данные.
- Структуры данных, используемые в индексах – crash safe.

На практике – почти не возможно.

MVCC, WAL: Вадим Михеев

Date: Mon, 09 Aug 1999 16:08:19 +0800
From: Vadim Mikheev
To: Oleg Bartunov
Subject: Re: indices grow !

----skipped ------
А что его понимать-то! -:)
Основное что надо помнить:

Запрос (те Query - то что читает записи из базы используя Seq/Index scans и отбирает их в соответствии с условиями в WHERE используя joins, subselects etc) видит (те возвращает) только те записи, который были живы в момент старта запроса(READ COMMITTED)/транзакции(SERIALIZABLE).

Всё остальное лишь производное -:)
....

http://www.sai.msu.su/~megera/postgres/mvcc.html

MVCC

MVCC – механизм, позволяющий каждой транзакции видеть свой «слепок» (snapshot) базы данных на определенный момент времени, хотя данные на текущий момент уже могли измениться.

В PostgreSQL MVCC обеспечивается тем, что данные не удаляются, старые версии строк остаются с отметками об окончании их актуальности, параллельно заводятся новые версии строк. Специальный процесс VACUUM удаляет старые версии строк.

MVCC

- ctid – физическое расположение кортежа: (страница,смещение)
- xmin – id транзакции, создавшей данную строку
- xmax – id транзакции, удалившей данную строку
- cmin – id команды транзакции, создавшей данную строку
- cmax – id команды транзакции, удалившей данную строку

MVCC: пример (1)

```
DELETE FROM mvcc_demo;
DELETE 0
INSERT INTO mvcc_demo VALUES (1);
INSERT 0 1
SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5413 |    0 |   1
(1 row)
BEGIN WORK;
BEGIN
UPDATE mvcc_demo SET val = 2;
UPDATE 1
```

MVCC: пример (2)

```
SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5414 | 0    |   2
(1 row)
```

```
           SELECT xmin, xmax, * FROM mvcc_demo;
            xmin | xmax | val
           ------+------+-----
            5413 | 5414 |   1
           (1 row)
```

```
COMMIT WORK;
COMMIT
```

MVCC: пример (3)

```
DELETE FROM mvcc_demo;
DELETE 1
INSERT INTO mvcc_demo VALUES (1);
INSERT 0 1
BEGIN WORK;
BEGIN
DELETE FROM mvcc_demo;
DELETE 1
ROLLBACK WORK;
ROLLBACK
SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5415 | 5416 |   1
(1 row)
```

MVCC: условие видимости

Коментарий к исходному коду в src/backend/utils/time/tqual.c:

```
((Xmin == my-transaction &&       inserted by the current transaction
  Cmin < my-command &&            before this command, and
  (Xmax is null ||                the row has not been deleted, or
   (Xmax == my-transaction &&     it was deleted by the current transaction
    Cmax >= my-command)))         but not before this command,
||                                or
 (Xmin is committed &&            the row was inserted by a committed transaction, and
     (Xmax is null ||             the row has not been deleted, or
      (Xmax == my-transaction &&  the row is being deleted by this transaction
       Cmax >= my-command) ||     but it's not deleted "yet", or
      (Xmax != my-transaction &&  the row was deleted by another transaction
       Xmax is not committed))))  that has not been committed
```

mao [Mike Olson] says 17 march 1993: the tests in this routine are correct; if you think they're not, you're wrong, and you should think about it again. i know, it happened to me.

WAL (Write-Ahead Log)

WAL бывает:

- На логическом уровне (insert/update/delete и т.д.);
- На уровне блоков.

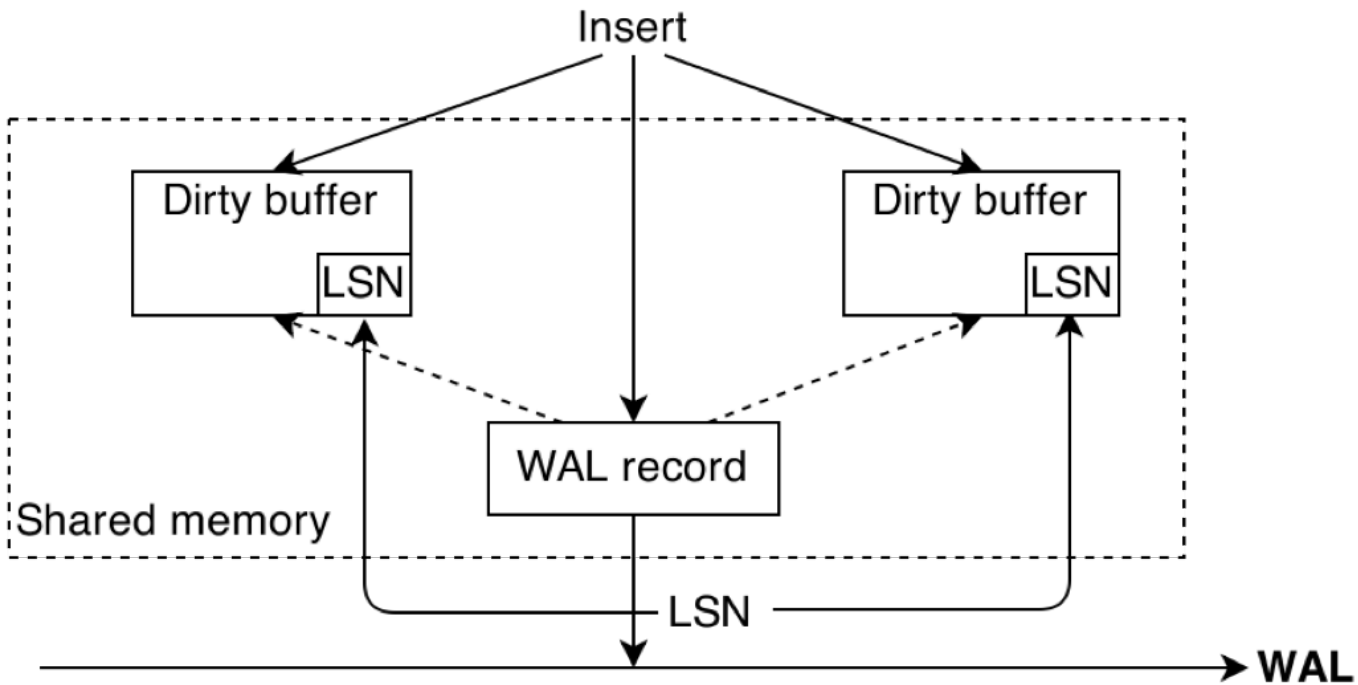
Для того, чтобы сделать WAL на физическом уровне нужно одно из двух:

- Всегда иметь на диске consistent snapshot данных;
  - Держать все данные в оперативной памяти или
  - держать на диске не меньше двух копий данных одновременно.
- Контролировать порядок фактической записи буферов ОС на диск (очень зависит от ОС).

Поэтому в PostgreSQL WAL на уровне блоков.

WAL в PostgreSQL

Любое изменение на дисковых страницах вначале записывает в WAL. Прежде, чем PostgreSQL сообщает об успешном commit’е транзакции для WAL делается fsync. Этим обеспечивается durability.



Heap: что стало

- Получился MVCC.
- WAL таки нужен.
- Undo лог действительно можно не использовать.
- Time-travel’ом пожертвовали ради меньшего bloat’а.

Что такое ACID?

- Atomicity – Атомарность
- Consistency – Согласованность
- Isolation – Изолированность
- Durability – Надёжность

Isolation (изолированность)

Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат. Включая неявное влияние, нарушение инвариантов, сохраняемых транзакцией. Строго говоря, нужен настоящий SERIALIZABLE.

- В PostgreSQL 9.1 появился SSI, реализующий настоящий SERIALIZABLE, достигаемые небольшим overhead'ом.
- В MySQL SERIALIZABLE достигается путём блокировки таблиц.
- В Oracle настоящего SERIALIZABLE нет...

Уровни изоляции транзакций (SQL-стандарт)

- READ UNCOMMITTED – могут быть увидены не закомиченые данные.
- READ COMMITTED – могут быть увидены только закомиченые данные.
- REPEATABLE READ – любая операция чтения должна повторяться.
- SERIALIZABLE – результат должен совпадать с последовательным выполнением транзакций.

Релазиция уровней изоляции в PostgreSQL

- SQL-стандарт разрешает использовать более сильный уровень изоляции транзакций.
- READ UNCOMMITTED не реализован, вместо него всегда используется READ COMMITTED. Не закомиченые изменения никто никогда не видит!
- До 9.1 REPEATABLE READ и SERIALIZABLE – одно и то же (snapshot isolation). Транзакция видит состояние базы на момент своего старта. Обновление одних и тех же строк вызывает ошибку.
- В рамках одной команды работает snapshot isolation.

Нарушение snapshot isolation в рамках команды

Такой запрос всегда увеличит balance на 100, даже если он был обновлён параллельной транзакцией.
UPDATE accounts SET balance = balance + 100 WHERE acctnum = 12345;
http://www.postgresql.org/docs/9.3/static/transaction-iso.html#XACT-READ-COMMITTED

Проблема SERIALIZABLE (1/3)

id	color
1	white
2	black
3	white
4	black
5	white
6	black

Проблема SERIALIZABLE (2/3)

```
BEGIN;
                                BEGIN;
UPDATE dots
SET color = 'black'
WHERE color = 'white';
                                UPDATE dots
                                SET color = 'white'
                                WHERE color = 'black';
COMMIT;
                                COMMIT;
```

Проблема SERIALIZABLE (3/3)

id	color
1	black
2	white
3	black
4	white
5	black
6	white

Такая ситуация не могла возникнуть при последовательном выполнении транзакций!

Serializable Snapshot Isolation (SSI)

- Введена в 9.1.
- Отслеживает зависимости между операциями чтения и записи различных транзакций (транзакция А прочитала то, что записала транзакция Б).
- Основывается на предикатных блокировках.
- При конфликте возникает ошибка (клиент должен быть готов повторить транзакцию).

Накладные расходы SSI

- Предикатные блокировки.
- Определние конфликтов.
- Повышенное число откатов транзакций.

Как писать приложения, использующий SSI

- Можно игнорировать парралелизм (не использовать SELECT FOR SHARE/UPDATE, LOCK).
- Быть готовым повторить транзакцию при ошибке.
- Объявлять читающие транзакции с помощью BEGIN READ ONLY;
- Избегать долгих транзакций.

Atomicity (атомарность)

Изменения, которые вносит в БД транзакция, могут быть либо полностью приняты, либо полностью отвергнуты. Транзакция не может быть зафиксирована частично (иначе что это будет за транзакция?).
Атомарность в PostgreSQL обеспечивается следующим.

- За счёт MVCC, пока транзакция работает, сохраняются и старые, и новые данные. В любой момент транзакцию можно принять или откатить.
- Commit/rollback транзакции защищается одной WAL-записью, которая атомарна.
- Таким образом, вся транзакция – атомарна.

Durability (надёжность)

Аппаратные сбои не должны приводить к потере данных.
Надёжность в PostgreSQL обеспечивается следующим.

- Все действия с данными делятся на атомарные операции, которые дублируются соответствующими WAL-записями.
- Если база упадёт, то при следующем запуске будет выполнен процесс recovery, в результате которого мы окажемся в том месте, на котором упали.
- Атомарные операции с данными устроены таким образом, чтобы никакие структуры данных не «поломались».

Consistency (согласованность)

Транзакции переводят БД из одного согласованного состояния в другое (внутри транзакции состояния могут быть несогласованными).
Согласованность в PostgreSQL обеспечивается следующим.

- Реализованы необходимые проверки для всех реализованных constraints, которые корректно работают при конкурентном доступе к БД.
- Constraints, реализованные на уровне приложения должны быть реализованы в соответствии с используемым уровнем изоляции транзакций.

Спасибо за внимание!

Александр Коротков
a.korotkov@postgrespro.ru
ООО «Постгрес Профессиональный»
http://postgrespro.ru/