

Getting more out of PostgreSQL

Advanced stuff!

Gavin Sherry

`gavin@alcove.com.au`

Alcove Systems Engineering

January 16, 2007

Outline

- 1 **Some cool SQL**
- 2 Increasing performance of SQL
- 3 Point in time recovery

How much can be achieved in SQL?

- In PostgreSQL, the SQL grammar is Turing complete so... you can do anything
- But, we're not going to write a web server
- awww : (

How much can be achieved in SQL?

- In PostgreSQL, the SQL grammar is Turing complete so... you can do anything
- But, we're not going to write a web server
- awww : (

How much can be achieved in SQL?

- In PostgreSQL, the SQL grammar is Turing complete so... you can do anything
- But, we're not going to write a web server
- awww : (

How is it Turing complete?

- Well, we have `generate_series()` – this gives us a loop

```
# select generate_series(1, 2);
 generate_series
```

```
-----
```

```
1
```

```
2
```

```
(2 rows)
```

What can generate_series() do?

- It means we can do cool stuff like

```
# select (current_date - i * '1 day'::interval)::date as day
       from generate_series(1, 8) i;
       day
```

```
-----
```

```
2007-01-14
```

```
2007-01-13
```

```
2007-01-12
```

```
2007-01-11
```

```
2007-01-10
```

```
2007-01-09
```

```
2007-01-08
```

```
2007-01-07
```

```
(8 rows)
```

We can also do nice tricks with arrays

- Say we want to aggregate rows into a string

```
# CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{}')
);

# select array_accum(i) from
      (select i from generate_series(1, 8) i) as foo;
      array_accum
-----
 {1,2,3,4,5,6,7,8}
(1 row)
```

We can also do nice tricks with arrays

- Say we want to aggregate rows into a string

```
# CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{} '
);

# select array_accum(i) from
      (select i from generate_series(1, 8) i) as foo;
      array_accum
-----
 {1,2,3,4,5,6,7,8}
(1 row)
```

Arrays and grouping

- So, lets do something useful
- Say we have a table `hits` with a date field and we want to see how many hits occurred for each day
- And, we want a pretty graph

```
# select d, count(*),
      array_to_string(array_accum('+'::text),'') as graph
from hits group by 1 order by 1 asc;
```

d	count	graph
2007-01-13	8	+++++++
2007-01-14	7	+++++++
2007-01-15	4	++++

(3 rows)

Speaking of grouping

- Something management types want is reports with running totals
- Who said we cannot do them?
- Lets get a running tota for the last week

```
# select h.d, h.count, sum(h2.count) as run from
  (select d, count(*) as count from hits
   group by 1) as h,
  (select d, count(*) as count from hits
   group by 1) as h2
 where h.d >= h2.d group by 1,2 order by h.d asc;
```

d	count	run
2007-01-13	8	8
2007-01-14	7	15
2007-01-15	4	19

(3 rows)

Speaking of grouping

- Something management types want is reports with running totals
- Who said we cannot do them?
- Lets get a running tota for the last week

```
# select h.d, h.count, sum(h2.count) as run from
  (select d, count(*) as count from hits
   group by 1) as h,
  (select d, count(*) as count from hits
   group by 1) as h2
 where h.d >= h2.d group by 1,2 order by h.d asc;
```

d	count	run
2007-01-13	8	8
2007-01-14	7	15
2007-01-15	4	19

(3 rows)

Hang on...

- Was it meant to be *for the last week*?

```
select h0.d, coalesce(sum(h2.count), 0) as run from
  (select g.d, count(h.d) as count from hits h right outer join
    (select (current_date - i * '1 day'::interval)::date as d from
      generate_series(0, 6) i) g on(h.d=g.d and
      h.d >= current_date - '1 week'::interval)
    group by 1) as h0 left outer join
  (select d, count(*) as count from hits
    where d >= current_date - '1 week'::interval
    group by 1) as h2 on(h0.d >= h2.d) group by 1 order by 1;
```

d	run
2007-01-09	0
2007-01-10	0
2007-01-11	0
2007-01-12	0
2007-01-13	8
2007-01-14	15
2007-01-15	19

How about totals (ROLLUP)

- These can be done in SQL as well

```
SELECT d, count(*) FROM hits GROUP BY 1
UNION ALL
SELECT NULL, count(*) FROM hits ORDER BY 1;
```

d	count
2007-01-13	8
2007-01-14	7
2007-01-15	4
	19

(4 rows)

Exercise

Extend this to other dimensions (userid). Then, extend it to a “data cube” or “cross tab” representation.

How about totals (ROLLUP)

- These can be done in SQL as well

```
SELECT d, count(*) FROM hits GROUP BY 1
UNION ALL
SELECT NULL, count(*) FROM hits ORDER BY 1;
```

d	count
2007-01-13	8
2007-01-14	7
2007-01-15	4
	19

(4 rows)

Exercise

Extend this to other dimensions (userid). Then, extend it to a “data cube” or “cross tab” representation.

How about totals (ROLLUP)

- These can be done in SQL as well

```
SELECT d, count(*) FROM hits GROUP BY 1
UNION ALL
SELECT NULL, count(*) FROM hits ORDER BY 1;
```

d	count
2007-01-13	8
2007-01-14	7
2007-01-15	4
	19

(4 rows)

Exercise

Extend this to other dimensions (userid). Then, extend it to a “data cube” or “cross tab” representation.

Outline

- 1 Some cool SQL
- 2 Increasing performance of SQL**
- 3 Point in time recovery

Procedure for addressing SQL performance issues

- Turn on `log_min_duration_statement` and see what's taking a long time
- Fix slow common queries first

Debugging query performance

- **EXPLAIN [ANALYZE]** gives you information about planning and plan execution

```
# explain analyze select * from hits where d = current_date;  
                                QUERY PLAN
```

```
-----  
Seq Scan on hits (cost=0.00..39.10 rows=10 width=8) (actual time=0.043..0.211 rows=4 loops=1)  
  Filter: (d = ('now'::text)::date)  
Total runtime: 0.317 ms  
(3 rows)
```

Stats

- One of the major factors in poor queries is poor statistics
- We store the stats in `pg_statistic`
- We use statistics to determine how many rows will match a query
- If we get this wrong, we may use operations which do not scale - like nested loop join

```
Nested Loop (cost=0.00..15.57 rows=30 width=4)
  (actual time=119.807..142.850 rows=1684 loops=1)
```

- Make stats collection more aggressive
- `ALTER TABLE .. ALTER COLUMN .. SET STATISTICS <100-1000>;`

Giving the planner better options

- Some queries will never be as fast as possible without modification
- Must add indexes or rewrite them
- Following are some classic patterns of poorly performing queries

The missing index

- Often, it's as simple as adding an index

Example

```
Seq Scan on accounts (cost=0.00..2029.00 rows=2000 width=4)
(actual time=113.016..212.117 rows=1 loops=1)
  Filter: (i = 2)
Total runtime: 212.203 ms
```

After adding an index

Example

```
Index Scan using accounts_i_idx on accounts (cost=0.00..10.01 rows=1 width=4)
(actual time=0.228..0.237 rows=1 loops=1)
  Index Cond: (i = 2)
Total runtime: 0.347 ms
```

The missing index

- Often, it's as simple as adding an index

Example

```
Seq Scan on accounts (cost=0.00..2029.00 rows=2000 width=4)
(actual time=113.016..212.117 rows=1 loops=1)
  Filter: (i = 2)
Total runtime: 212.203 ms
```

After adding an index

Example

```
Index Scan using accounts_i_idx on accounts (cost=0.00..10.01 rows=1 width=4)
(actual time=0.228..0.237 rows=1 loops=1)
  Index Cond: (i = 2)
Total runtime: 0.347 ms
```

The missing index

- Often, it's as simple as adding an index

Example

```
Seq Scan on accounts (cost=0.00..2029.00 rows=2000 width=4)
(actual time=113.016..212.117 rows=1 loops=1)
  Filter: (i = 2)
Total runtime: 212.203 ms
```

After adding an index

Example

```
Index Scan using accounts_i_idx on accounts (cost=0.00..10.01 rows=1 width=4)
(actual time=0.228..0.237 rows=1 loops=1)
  Index Cond: (i = 2)
Total runtime: 0.347 ms
```

Partial and functional indexes

- Don't forget, you can index `LOWER(lastname)` and `startdate IS NULL`
- This can solve significant performance issues!

```
CREATE INDEX people_fn_idx on people(lower(firstname));  
CREATE INDEX proj_un_processed on projects(startdate)  
    WHERE startdate ISNULL;
```

Unanchored tables

- Some developers do the funniest things
- You'll sometimes see a query like this

```
SELECT DISTINCT a.id FROM products a, parts b, prices c
WHERE a.id=b.id AND a.creation_date > current_timestamp - '1 year'
AND delivery_status = 't' and b.manufacturer = 'FOO';
```

- Now, for every row resulting from the join of 'a' to 'b', all rows from 'c' are returned because 'c' isn't qualified
- The developer probably added the `DISTINCT` because s/he didn't know why so many results we returned!
- Qualifying this correctly will result in an order of magnitude speed up

Unanchored tables

- Some developers do the funniest things
- You'll sometimes see a query like this

```
SELECT DISTINCT a.id FROM products a, parts b, prices c
WHERE a.id=b.id AND a.creation_date > current_timestamp - '1 year'
AND delivery_status = 't' and b.manufacturer = 'FOO';
```

- Now, for every row resulting from the join of 'a' to 'b', all rows from 'c' are returned because 'c' isn't qualified
- The developer probably added the `DISTINCT` because s/he didn't know why so many results we returned!
- Qualifying this correctly will result in an order of magnitude speed up

Missing indexes from foreign keys

- Developers regularly do the following

```
CREATE TABLE parts (  
    partid int primary key,  
    ...  
);
```

```
CREATE TABLE products (  
    prodid int primary key,  
    partid int references parts(partid)  
        on update cascade on delete cascade,  
    ...  
);
```

- There is no index on 'products.partid' so any delete or update to 'products' which is cascaded to products will have to do a sequential scan

Bloated index

- Some times, in tables which are regularly updated and deleted from, indexes suffer 'bloat'
- `REINDEX` rebuilds these indexes to remove the bloat
- Only do this during maintenance time
- To see if you're index is bloated
 - Multiply the key size by the number of rows
 - Determine the index size (`pg_relation_size()`)
 - If it is greater than the size of the underlying table, it's probably bloated

Tablespaces can work wonders

- Tablespaces allow you to put different database objects on different physical devices
- Problem is, not everyone can afford new storage!

Example

```
CREATE TABLESPACE indexspace LOCATION '/mnt/fast1/';  
ALTER INDEX hits_d_idx SET TABLESPACE indexspace;
```

Data partitioning

- Say your `hit` table gets *very big*
- You can partition the data!
- You'll have to turn on `constraint_exclusion`

Example

```
CREATE TABLE hits (d date, usrid int);
CREATE TABLE hits200701 (CHECK(d >= '2007-01-01' AND d < '2007-02-01'))
    INHERITS(hits);
...
CREATE RULE hits_ins_200701_rule AS ON INSERT TO hits
    WHERE (d >= '2007-01-01' AND d < '2007-02-01')
    DO INSERT INTO hits200701 VALUES(NEW.d, NEW.usrid);
```

An exercise

This doesn't really scale beyond 100 partitions. Using a trigger, build a partitioning system which scales.

Data partitioning

- Say your `hit` table gets *very big*
- You can partition the data!
- You'll have to turn on `constraint_exclusion`

Example

```
CREATE TABLE hits (d date, usrid int);
CREATE TABLE hits200701 (CHECK(d >= '2007-01-01' AND d < '2007-02-01'))
    INHERITS(hits);
...
CREATE RULE hits_ins_200701_rule AS ON INSERT TO hits
    WHERE (d >= '2007-01-01' AND d < '2007-02-01')
    DO INSERT INTO hits200701 VALUES(NEW.d, NEW.usrid);
```

An exercise

This doesn't really scale beyond 100 partitions. Using a trigger, build a partitioning system which scales.

Materialised views

- Application writers love doing this:

```
SELECT COUNT(*) FROM accounts;
```

- They like putting it on *every damn* page even better
- If `accounts` gets large, you're in trouble

What the l33t hax0rs do

```
CREATE TABLE accounts_count (count bigint);  
INSERT INTO accounts SELECT count(*) FROM accounts;  
CREATE RULE accounts_ins_rule AS ON INSERT TO  
  accounts  
  DO UPDATE accounts SET count = count + 1;
```

Materialised views

- Application writers love doing this:
`SELECT COUNT(*) FROM accounts;`
- They like putting it on *every damn* page even better
- If `accounts` gets large, you're in trouble

What the l33t hax0rs do

```
CREATE TABLE accounts_count (count bigint);
INSERT INTO accounts SELECT count(*) FROM accounts;
CREATE RULE accounts_ins_rule AS ON INSERT TO
  accounts
  DO UPDATE accounts SET count = count + 1;
```

Materialised views

- Application writers love doing this:
`SELECT COUNT(*) FROM accounts;`
- They like putting it on *every damn* page even better
- If `accounts` gets large, you're in trouble

What the l33t hax0rs do

```
CREATE TABLE accounts_count (count bigint);
INSERT INTO accounts SELECT count(*) FROM accounts;
CREATE RULE accounts_ins_rule AS ON INSERT TO
  accounts
  DO UPDATE accounts SET count = count + 1;
```

Materialised views

Exercise

This doesn't work for `DELETE` and `UPDATE`. Why not? Fix it to work (hint: triggers – again)

Sometimes, the old ways work

- With stateless environments like the web, we've forgotten old way – even when we're doing traditional client server
- Cursors provide a convenient way of running a query once and reusing the output
- Consider CRM style applications
 - Number of accounts is often high
 - Users page through accounts often – should we execute that query every time?
 - Often, we just want the top N accounts
 - Updates are rare, and invalidation is easy – See NOTIFY

Outline

- 1 Some cool SQL
- 2 Increasing performance of SQL
- 3 Point in time recovery**

Continuous backup

- Point-in-time recovery gives us a way to ship changes to other machines
- We say that backup is continuous in this scenario
- To get started, define an archive command

```
archive_command = 'scp -B %p postgres@backup:/archive/%f'
```

- `%p` - full path to the journal file to be archived
- `%f` - the name of the journal file
- `archive_command` can be any command which copies the file and returns zero on success, non-zero on failure
- If the command returns non-zero, the file will not be deleted but archiving will halt

PITR: performing a base backup

Example

```
$ psql template1 -c \  
    "select pg_start_backup('backup_'date +%Y-%M-%d_%H.%M.%S'');"\  
$ tar -zcf basedump.tar $PGDATA\  
$ psql template1 -c "select pg_stop_backup();"
```

- This takes a snapshot of the running system
- By itself, this data is corrupted
- The archived journals will recover it to a usable state

PITR: recovery process

- 1 Stop PostgreSQL
- 2 Copy `$PGDATA` and all tablespaces to a temporary location
- 3 Remove all files from `$PGDATA/base/` and all tablespaces
- 4 Restore your base backup, not forgetting tablespaces
- 5 Remove any files in `pg_xlog`
- 6 Copy any unarchived journal files from the data directory you moved in step 2.
- 7 Create a recovery configuration file, which does the reverse of the archive command
- 8 Start PostgreSQL

PITR: recovery.conf

- `recovery.conf` is the recovery configuration file
- Put this in `$PGDATA`.

```
restore_command = 'scp -B postgres@backup:/archivev/%f %p'  
recovery_target_time = '2006-05-25 11:00:00 EST'
```

PITR: recovery tricks

- PITR can recover to the most recent point in time or to a **specific point in time**
- Say a junior DBA drops a table at 11am and you do not discover this until 1pm
- Restore a base backup on a second system, roll forward to 11am
- Then, `pg_dump` the table out to disk and restore in production

PITR Caveats

- **Test this, before you come to rely on it**
- Test it regularly
- Journal files and base backups will not restore across system architectures
 - You cannot restore a 32-bit backup and archive to a 64-bit SPARC system
- It will take 10 seconds to 1 minute to process a journal file

Exercise

As of PostgreSQL 8.2, you can put the backup machine into 'continuous replay'. Setup such a system.

PITR Caveats

- **Test this, before you come to rely on it**
- Test it regularly
- Journal files and base backups will not restore across system architectures
 - You cannot restore a 32-bit backup and archive to a 64-bit SPARC system
- It will take 10 seconds to 1 minute to process a journal file

Exercise

As of PostgreSQL 8.2, you can put the backup machine into 'continuous replay'. Setup such a system.

PITR Caveats

- **Test this, before you come to rely on it**
- Test it regularly
- Journal files and base backups will not restore across system architectures
 - You cannot restore a 32-bit backup and archive to a 64-bit SPARC system
- It will take 10 seconds to 1 minute to process a journal file

Exercise

As of PostgreSQL 8.2, you can put the backup machine into 'continuous replay'. Setup such a system.