What to Expect in PostgreSQL 12

- Robert Haas | Postgres Vision 2019

Overview

- Table Partitioning
- Indexing
- The Query Planner
- SQL Features
- Odds and Ends
- Postscript: Advanced Server

Table Partitioning

Partition Pruning

```
rhaas=# \d foo
                Table "public.foo"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 a      | integer |           |          |
 b      | text    |           |          |
Partition key: RANGE (a)
Indexes:
    "foo_a_idx" UNIQUE, btree (a)
Number of partitions: 1000 (Use \d+ to list them.)

rhaas=# select * from foo where a = 20190625;
    a     |   b
----------+--------
 20190625 | filler
(1 row)
```

Partition Pruning

```
rhaas=# explain update foo set b = 'modification' where
a = 20190625;
                        QUERY PLAN
-----------------------------------------------------------
 Update on foo  (cost=0.42..8.44 rows=1 width=42)
   Update on foo20
   ->  Index Scan using foo20_a_idx on foo20
(cost=0.42..8.44 rows=1 width=42)
         Index Cond: (a = 20190625)
(4 rows)
```

Partition Pruning: Results

Version	SELECT	EXPLAIN UPDATE
v11	33.114 ms	223.592 ms
v12beta	0.432 ms	0.535 ms

Faster COPY into Partitioned Tables

```
rhaas=# \d bar
                Table "public.bar"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 a      | integer |           | not null |
 b      | text    |           |          |
Partition key: HASH (a)
Indexes:
    "bar_pkey" PRIMARY KEY, btree (a)
Number of partitions: 8 (Use \d+ to list them.)

rhaas=# copy bar from
'/Users/rhaas/testcase/testbar.csv' csv; -- 100k rows
Time: 330.384 ms (on v11)
Time: 276.175 ms (on v12beta, ~16% faster)
```

Concurrent ATTACH PARTITION

```
rhaas=# select count(*) from foo;
```

```
rhaas=# create table foo1000 (a int, b text);
CREATE TABLE
rhaas=# alter table foo attach partition foo1000 for
values from (1000000000) to (1001000000);
ALTER TABLE
```

- On v11, the ALTER TABLE will block until the query completes.
- On v12, it will not block.

Foreign Keys To Partitioned Tables

```
rhaas=# \d bar
                Table "public.bar"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 a      | integer |           | not null |
 b      | text    |           |          |
Partition key: HASH (a)
Indexes:
    "bar_pkey" PRIMARY KEY, btree (a)
Number of partitions: 8 (Use \d+ to list them.)

rhaas=# create table bar_details (id serial primary
key, a integer references bar (a), s text);
ERROR:  cannot reference partitioned table "bar"
```

- On v12, the error is gone!

Merge Append Avoidance

```
rhaas=# explain select * from foo order by a;
                        QUERY PLAN
-----------------------------------------------------------
 Merge Append  (cost=475.04..86230275.02
rows=1000001270 width=11)
   Sort Key: foo0.a
   ->  Index Scan using foo0_a_idx on foo0
(cost=0.42..31389.42 rows=1000000 width=11)
   ...
```

```
rhaas=# explain select * from foo order by a;

                        QUERY PLAN
-----------------------------------------------------------
 Append  (cost=425.15..36394030.55 rows=1000001270
width=11)

   ->  Index Scan using foo0_a_idx on foo0
(cost=0.42..31389.42 rows=1000000 width=11)
  ...
```

Merge Append Avoidance: Results

```
rhaas=# select * from foo order by a offset 1000000000;
 a | b
---+---
(0 rows)

Time: 209820.957 ms (03:29.821) (on v11)
Time: 169733.233 ms (02:49.733) (on v12beta,
      about 19% faster)
```

Indexing Improvements

btree Index Improvements: Test Setup

```
pgbench -i -s 100

create index on pgbench_accounts (filler);

select oid::regclass, pg_relation_size(oid) from
pg_class where relname like 'pgbench%' and relkind =
'i';

pgbench -T 300 -c 8 -j 8 -N

select oid::regclass, pg_relation_size(oid) from
pg_class where relname like 'pgbench%' and relkind =
'i';
```

btree Index Improvements: Results

Version	Index on “filler”	Index on “aid”
v11 (initial)	1089 MB	214 MB
v11 (final)	1204 MB	240 MB
v12beta (initial)	1091 MB (+0.1%)	214 MB (+0.0%)
v12beta (final)	1118 MB (-7.2%)	214 MB (-10.9%)

- Beware: The maximum size of an item that can be indexed using btree has decreased by 8 bytes!

REINDEX CONCURRENTLY

- REINDEX takes ShareLock on table and AccessExclusiveLock on index, blocking basically all access to the table – everything except prepared queries that don't use the index in question.
- REINDEX CONCURRENTLY takes ShareUpdateExclusiveLock on both table and index, permitting concurrent reads and writes.
- Similar to DROP INDEX CONCURRENTLY + CREATE INDEX CONCURRENTLY.
- Waits for concurrent transactions to end, twice.
- Watch out for invalid indexes if fails or is interrupted.

GiST and SP-GiST Indexes

- GiST indexes now support INCLUDE columns.
- SP-GiST indexes now support K-nearest-neighbor searches.
- GiST, GIN, and SP-GiST indexes now generate less WAL during index creation.
- VACUUM of GiST indexes is now more efficient and can recycle empty leaf pages.

CREATE INDEX Progress Reporting (1/2)

```
-[ RECORD 1 ]------+------------------------------
pid                | 66541
datid              | 16384
datname            | rhaas
relid              | 23914
index_relid        | 0
command            | CREATE INDEX
phase              | building index: scanning table
lockers_total      | 0
lockers_done       | 0
current_locker_pid | 0
blocks_total       | 1639345
blocks_done        | 523774
tuples_total       | 0
tuples_done        | 0
partitions_total   | 0
partitions_done    | 0
```

CREATE INDEX Progress Reporting (2/2)

```
-[ RECORD 1 ]------+------------------------------
pid                | 66541
datid              | 16384
datname            | rhaas
relid              | 23914
index_relid        | 0
command            | CREATE INDEX
phase              | building index: loading tuples in tree
lockers_total      | 0
lockers_done       | 0
current_locker_pid | 0
blocks_total       | 0
blocks_done        | 0
tuples_total       | 100000000
tuples_done        | 10409041
partitions_total   | 0
partitions_done    | 0
```

The Query Planner

Common Table Expressions (WITH) v11

```
rhaas=# explain (costs off) with x as (select * from
bar), y as (select * from bar) select * from x, y where
x.a = y.a;
            QUERY PLAN
----------------------------------
 Merge Join
   Merge Cond: (x.a = y.a)
   CTE x
     ->  Seq Scan on bar
   CTE y
     ->  Seq Scan on bar bar_1
   ->  Sort
         Sort Key: x.a
         ->  CTE Scan on x
   ->  Materialize
         ->  Sort
               Sort Key: y.a
               ->  CTE Scan on y
```

Common Table Expressions (WITH) v12

```
rhaas=# explain (costs off) with x as (select * from
bar), y as (select * from bar) select * from x, y where
x.a = y.a;
                QUERY PLAN
-----------------------------------
 Hash Join
   Hash Cond: (bar.a = bar_1.a)
   ->  Seq Scan on bar
   ->  Hash
         ->  Seq Scan on bar bar_1
```

- Substantially simpler query plan.
- In this particular case, about one-third faster.
- Actual gains will vary widely.
- Can use WITH … AS [NOT] MATERIALIZED (…)

Plan Cache Mode

- Prepared queries can be handled in two ways.
  - Strategy #1: Replan the query each time it's executed for the particular parameter values in use. ("custom plan")
  - Strategy #2: Create a plan that will work with any parameter values and reuse it. ("generic plan")
- By default, PostgreSQL will try to adaptively pick the best strategy.
- If you know better, you can set plan_cache_mode.
  - Typical use: Force custom plans, because the generic plans are worse than the planner thinks.
  - Possible use: Don't waste any planning time trying to create worthless custom plans.

Multivariate MCV Lists: Setup

```
rhaas=# create table t2 (a int, b int);
rhaas=# insert into t2 select mod(i,100), mod(i,100)
   from generate_series(1,1000000) s(i);
rhaas=# analyze t2;
rhaas=# explain analyze select * from t2
   where (a = 1) and (b = 1);
rhaas=# explain analyze select * from t2
   where (a = 1) and (b = 2);
rhaas=# create statistics s2 (mcv) on a, b from t2;
rhaas=# analyze t2;
rhaas=# explain analyze select * from t2
   where (a = 1) and (b = 1);
rhaas=# explain analyze select * from t2
   where (a = 1) and (b = 2);
```

Multivariate MCV List: Results

	… a = 1 AND b = 1	... a = 1 AND b = 2
Estimated Row Count	109	106
Estimated Row Count with Extended Statistics	10267	1
Actual Row Count	10000	0

Support Functions for SQL Functions

```
rhaas=# explain select * from
        generate_series(1, 437218) g;
                      QUERY PLAN
------------------------------------------------------
 Function Scan on generate_series g
(cost=0.00..10.00 rows=1000 width=4)
```

```
rhaas=# explain select * from
        generate_series(1, 437218) g;
                      QUERY PLAN
------------------------------------------------------

 Function Scan on generate_series g
(cost=0.00..4372.18 rows=437218 width=4)
```

SQL Features

Generated Columns

```
rhaas=# create table gce (a int, b int, c int
generated always as (a + b) stored);
```

- Column c can’t be manually updated.
- It will be recomputed after every INSERT/UPDATE.
- Easier (but not necessarily faster) than a TRIGGER.

SQL/JSON: jsonpath

```
rhaas=# select jsonb_path_query('{ "track" :
  {
    "segments" : [
      { "location":   [ 47.763, 13.4034 ],
        "start time": "2018-10-14 10:05:14",
        "HR": 73
      },
      { "location":   [ 47.706, 13.2635 ],
        "start time": "2018-10-14 10:39:21",
        "HR": 130
      } ]
  }
}', '$.track.segments[*].location');
 jsonb_path_query
--------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
(2 rows)
```

Nondeterministic Collations

- Normal collations do not allow ties.
- If you say something like “ORDER BY a, b,” the fact that we are also ordering by b only matters if there are completely-identical values in a.
- If you have something like “robert” and “Robert” in column a, those can’t be considered equal – PostgreSQL will insert a tiebreak rule.
- Nondeterministic collations let you do define collations with no tiebreak rule – values that are not identical can still be considered “equal.”

Odds & Ends

recovery.conf is no more

- Settings previously stored in recovery.conf are now in postgresql.conf
- Use recovery.signal or standby.signal to trigger recovery or standby mode
- Your backup management tool (or scripts) will likely need an update.
- A few recovery-related parameters can now be changed without restarting the server: archive_cleanup_command, promote_trigger_file, recovery_end_command, and recovery_min_apply_delay.

Enable or Disable Checksums Offline

```
[rhaas ~]$ pg_ctl stop
waiting for server to shut down.... done
server stopped
[rhaas ~]$ time pg_checksums -e
Checksum operation completed
Files scanned:  6289
Blocks scanned: 10080538
pg_checksums: syncing data directory
pg_checksums: updating control file
Checksums enabled in cluster

real    2m42.120s
user    0m20.674s
sys 1m30.388s
[rhaas ~]$ du -hs $PGDATA
 87G    /Users/rhaas/pgdata
```

Table Access Methods

- PostgreSQL can now support multiple table storage formats, just as we have for years been able to support multiple index formats (hash, btree, gist, etc.).
- Currently, the only in-core table storage method is ‘heap’.
- Expect more choices in a year or two.
- Support for hidden OID columns removed.

Miscellany

- GSSAPI encryption support
- Progress reporting for CLUSTER and VACUUM FULL
- SERIALIZABLE for parallel query
- pg_upgrade can use filesystem cloning.
- Improved psql tab completion for many DML commands.
- Unified logging framework for client tools, including colorization support.

Advanced Server 12

- Interval Partitioning
- Compound Triggers
- MEDIAN, LISTAGG
- CAST(MULTISET)
- System View Improvements

Thanks

- Any questions?