Partitioning Shines in PostgreSQL 11

Amit Langote, NTT OSS Center

PGConf.ASIA, Tokyo
Dec 11, 2018

About me

- Amit Langote
- Work at NTT OSS Center developing PostgreSQL
- Contributed mainly to table partitioning
- Gave a presentation on related topic at the last year's PGConf.ASIA

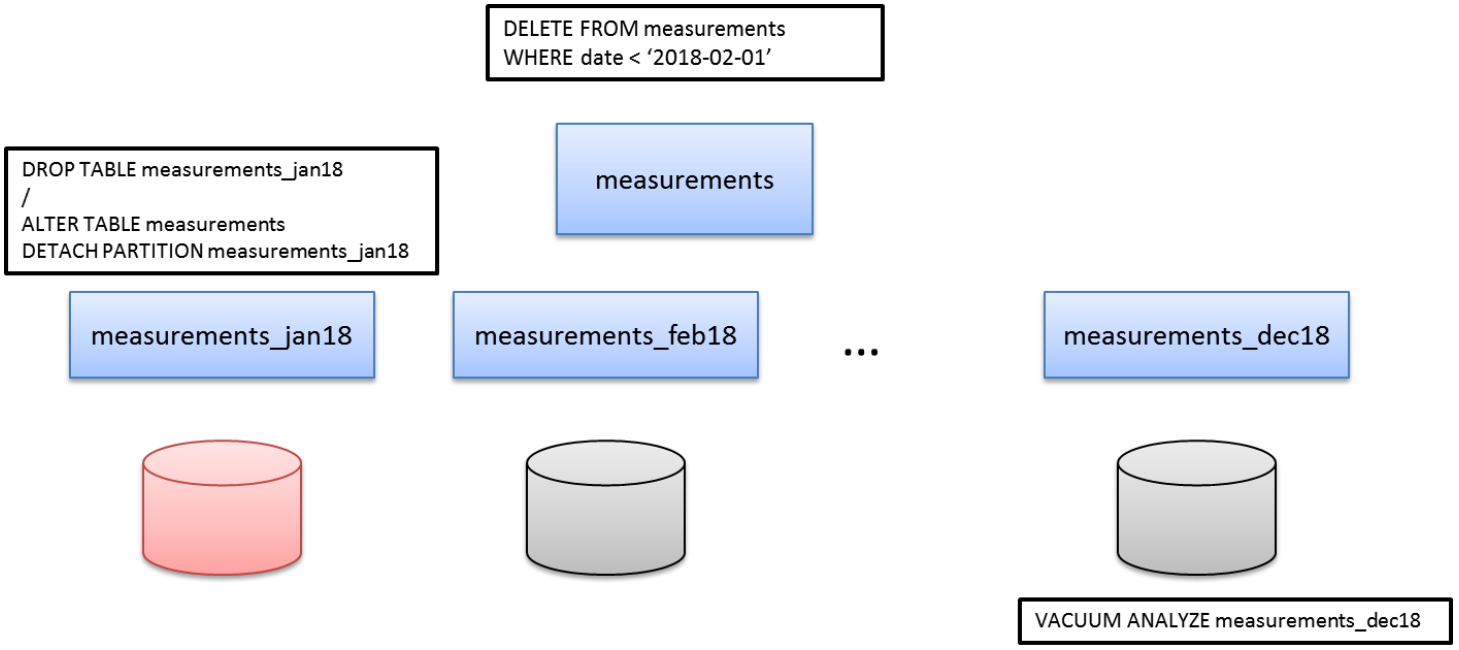
Agenda

- What is partitioning?
- What is declarative partitioning?
- Limitations of partitioning in PostgreSQL 10
- New partitioning features in PostgreSQL 11
- Explanation of new partitioning features
- Performance problems
- Ongoing work for PostgreSQL 12 for performance
- Future work on performance

NTT

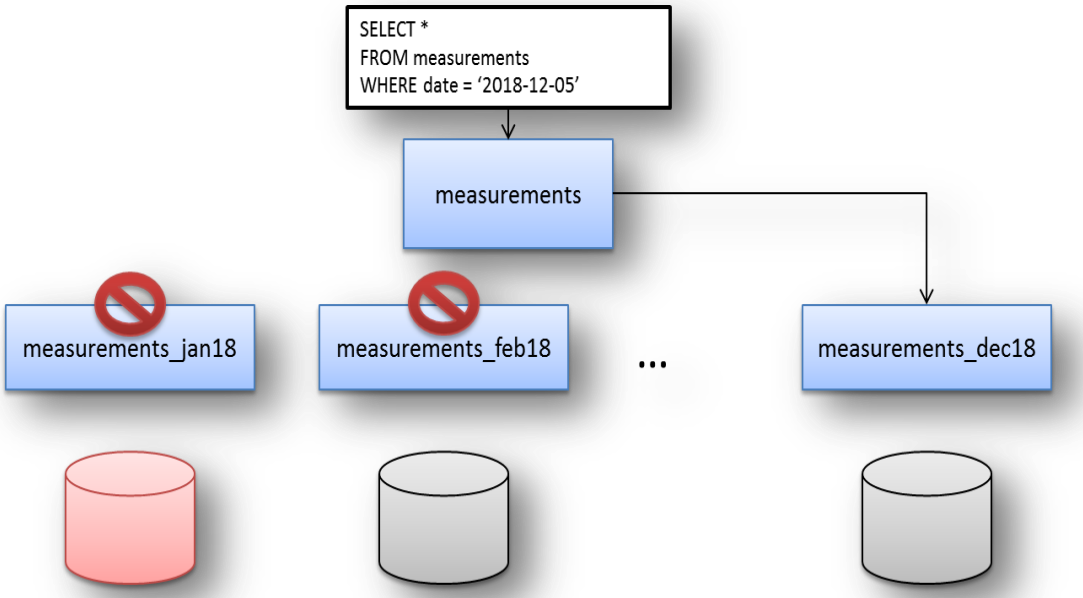
What is partitioning?

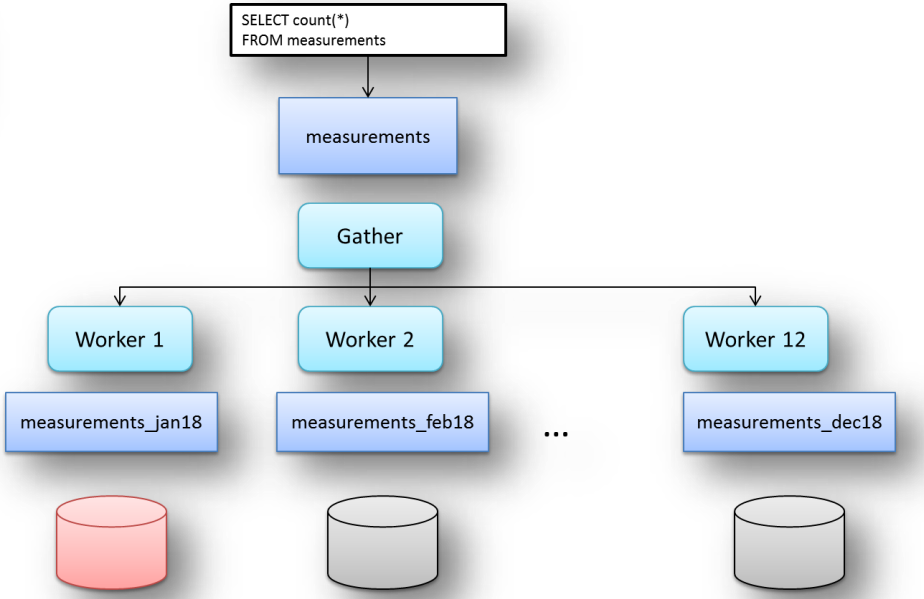
- Dividing a large table into smaller ones for manageability and performance
- **Manageability**: quick archival and maintenance of smaller chunks of independent data



What is partitioning?

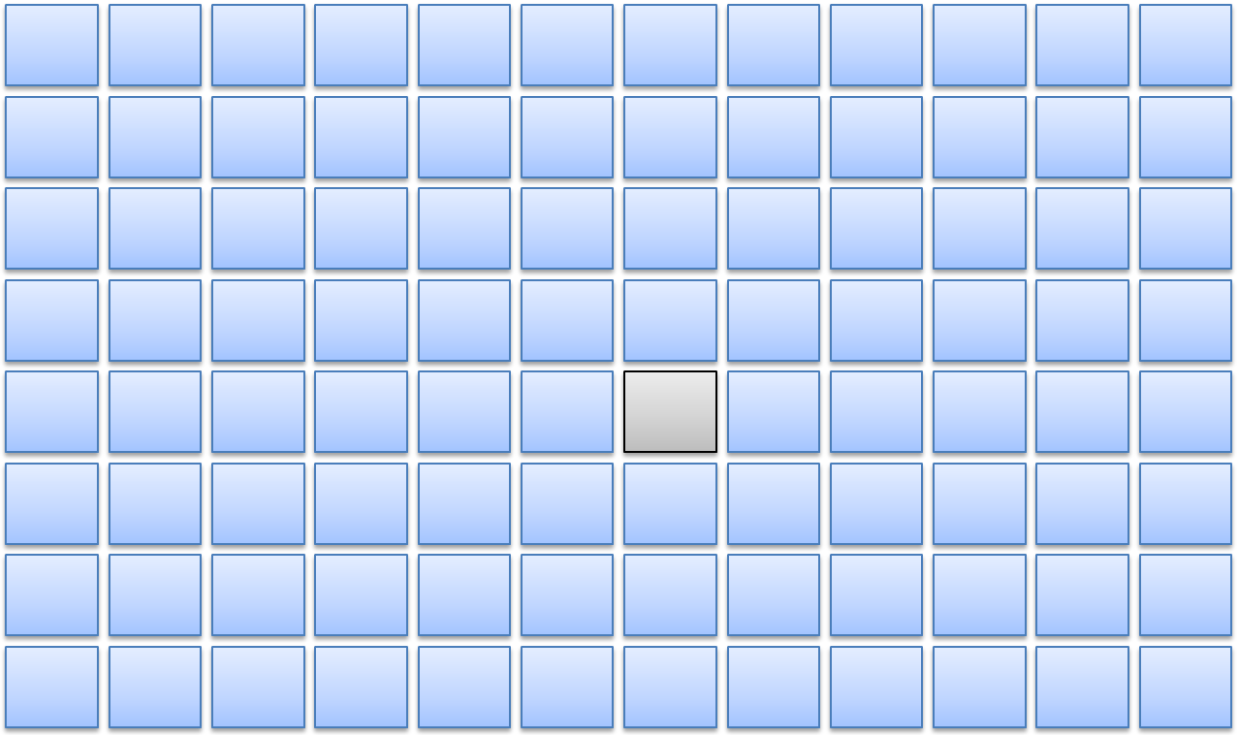
- **Performance**: quick query execution by pruning unnecessary partitions and parallel processing





What is partitioning?

- **Scalability:** partitioning related optimizations ensure that performance doesn't degrade as the data size increases



What is declarative partitioning?

- A feature added in PostgreSQL 10 to make partitioning easier and faster
- Up to PostgreSQL 9.6
  - Use table inheritance to link parent table to its partitions
  - Add `CHECK` constraint to child tables as partition constraint (*constraint exclusion* can prune unnecessary partitions based on it)
  - Create `BEFORE INSERT FOR EACH ROW` trigger on the parent table to redirect inserted data to correct partition (known as *tuple routing*)
- In PostgreSQL 10
  - Create parent tables with `PARTITION BY` clause to specify how to divide the incoming data based on the values of partition key columns
  - Create partitions with PARTITION OF ... FOR VALUES clause to specify the set of partition key values that partition allows
  - Partition constraints and tuple routing handled internally
  - Many limitations!

PostgreSQL 10 limitations

- No hash partitioning
- No default partition to "catch" unattended data
- No indexes and row-level triggers on parent tables
  - Can be created on individual partitions manually
- No foreign key constraint on parent tables
  - Can be created on individual partitions manually
- No `UPDATE` tuple routing
- No `INSERT ON CONFLICT` on parent tables
- Partition pruning still based on constraint exclusion

New partitioning features in PostgreSQL 11

- Hash partitioning
- Default partition to "catch" unattended data
- Create indexes on parent tables
  - To specify UNIQUE, include partition key columns in the index key
- AFTER ... FOR EACH ROW triggers on parent tables
  - BEFORE ... FOR EACH ROW not supported!
- Create foreign key on parent tables
  - Foreign keys referencing parent tables not supported!
- UPDATE tuple routing
- INSERT ON CONFLICT on parent tables
  - Handling some cases requires UNIQUE index to be present

New partitioning features in PostgreSQL 11

- New partitioning-based optimization features
  - Improved partition pruning (no longer uses constraint exclusion)
  - Run-time partition pruning
  - Partitionwise join and aggregation

NTT

Hash partitioning

- Sometimes it's not clear which values to assign to individual partitions
  - ➢ Something you need to decide when using list and range partitioning
- Hash partitioning is useful in that case
- Just decide how many partitions to create
- Example with 4 hash partitions:

```
create table measurement (sensor_id text, data json) partition by hash (sensor_id);
create table measurement0 partition of measurement for values with (modulus 4, remainder 0);
create table measurement1 partition of measurement for values with (modulus 4, remainder 1);
create table measurement2 partition of measurement for values with (modulus 4, remainder 2);
create table measurement3 partition of measurement for values with (modulus 4, remainder 3);
```

Hash partitioning

- Server will assign the partition by computing the remainder as `hashfunc(sensor_id) % 4`, where `hashfunc` is a hashing function for `text` data type
- Hashing as described above ensures uniform distribution of rows among partitions
- Documentation contains tips on how to increase the number of hash partitions without needing to redistribute the data contained in existing partitions

https://www.postgresql.org/docs/devel/sql-createtable.html#SQL-CREATETABLE-PARTITION

Default partition

- In some cases, you may not know in advance all the values for which to create partitions
  - No, PostgreSQL won't create a partition *automatically* if a new value shows up
  - Instead you get an error that a partition for given value doesn't exist
- Default partition helps with that
  - It doesn't have a specific set of values assigned to it
  - Any value for which there is no partition defined will go into the default partition
- There can be only 1 default partition
- Hash partitioned tables cannot have default partition
- Syntax:

```
create table default_partition partition of parent_table default;
```

Indexes

- NOT some new type of index that covers all partitions in one storage-level object
- Since parent table doesn't store data by itself, any indexes created on it should be cascaded to its partitions which store data
  - PostgreSQL 10 doesn't implement that cascading
- PostgreSQL 11 does!
- Any indexes defined on the parent are duplicated in all partitions
  - Both existing partitions and those created or attached after-the-fact will have the index
- Parent indexes can be used for UNIQUE constraint too
  - Must include partition key columns in the index key
- Example of a UNIQUE index:

Indexes

```
create table accounts (id text, name text) partition by hash (id);
create table accounts0 partition of accounts for values with (modulus 4, remainder 0);
create table accounts1 partition of accounts for values with (modulus 4, remainder 1);
create table accounts2 partition of accounts for values with (modulus 4, remainder 2);
create table accounts3 partition of accounts for values with (modulus 4, remainder 3);
-- ok
create unique index on accounts (id);

-- can’t create unique index if its key doesn’t contain partition key
create unique index on accounts (name);
ERROR:  insufficient columns in UNIQUE constraint definition
DETAIL:  UNIQUE constraint on table "accounts" lacks column "id" which is part of the
partition key.

-- ok if it’s non-unique though
create index on accounts (name);
```

Indexes

```
-- can also specify the UNIQUE or PRIMARY KEY constraint inline


drop table accounts;
create table accounts (id text primary key, name text) partition by hash (id);
create table accounts0 partition of accounts for values with (modulus 4, remainder 0);
create table accounts1 partition of accounts for values with (modulus 4, remainder 1);
create table accounts2 partition of accounts for values with (modulus 4, remainder 2);
create table accounts3 partition of accounts for values with (modulus 4, remainder 3);
```

Indexes

```
-- see that the index is present in partitions


\d accounts0
               Table "public.accounts0"
  Column   |  Type   | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
 id        | text    |           | not null |
 branch_id | integer |           |          |
Partition of: accounts FOR VALUES WITH (modulus 4, remainder 0)
Indexes:
    "accounts0_pkey" PRIMARY KEY, btree (id)
```

Triggers

- Again, not some new kind of trigger specialized for partitioned tables
- Parent tables can already have statement-level triggers, but any row-level triggers must be defined on partitions due to certain implementation details
  - PostgreSQL 10 doesn't implement cascading of trigger definition to partitions
- PostgreSQL 11 does!
- Only AFTER ... FOR EACH ROW triggers are cascaded
  - Cascading BEFORE ... FOR EACH ROW triggers may be pursued in the future after taking care of some corner cases

Foreign keys

- With PostgreSQL 10, you can add foreign keys to individual partitions (in both directions), but not to parent table
- PostgreSQL 11 lets you add it to parent table and cascades the definition to partitions
  - ➢ But only the outgoing foreign keys
- Examples:

```
create table accounts (id text primary key, branch_id int) partition by hash (id);
create table accounts0 partition of accounts for values with (modulus 4, remainder 0);
create table accounts1 partition of accounts for values with (modulus 4, remainder 1);
create table accounts2 partition of accounts for values with (modulus 4, remainder 2);
create table accounts3 partition of accounts for values with (modulus 4, remainder 3);
create table branches (id int primary key, name text);

alter table accounts add foreign key (branch_id) references branches (id);
```

Foreign keys

```
-- or specify the foreign key inline
drop table accounts;
create table accounts (id text primary key, branch_id int references branches) partition
by hash (id);
create table accounts0 partition of accounts for values with (modulus 4, remainder 0);
create table accounts1 partition of accounts for values with (modulus 4, remainder 1);
create table accounts2 partition of accounts for values with (modulus 4, remainder 2);
create table accounts3 partition of accounts for values with (modulus 4, remainder 3);
```

Foreign keys

```
\d accounts0
               Table "public.accounts0"
  Column   |  Type   | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
 id        | text    |           | not null |
 branch_id | integer |           |          |
Partition of: accounts FOR VALUES WITH (modulus 4, remainder 0)
Indexes:
    "accounts0_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "accounts_branch_id_fkey" FOREIGN KEY (branch_id) REFERENCES branches(id)

-- this doesn't work yet
create table history (aid text references accounts, delta int);
ERROR:  cannot reference partitioned table "accounts"
```

UPDATE tuple routing

- One may occasionally be forced to update the partition key of a row such that the new row needs to be moved to another partition
  - ➢ PostgreSQL 10 would throw its hands up and show an error message
- PostgreSQL 11 handles the case gracefully by re-routing the new row
- Consider the following somewhat artificial example:

```
create table sensor_readings (name text, reading int) partition by list (left(name, 2));
create table "sensor_readings_AB" partition of sensor_readings for values in ('AB');
create table "sensor_readings_CD" partition of sensor_readings for values in ('CD');

insert into sensor_readings values ('AB0001', 0), ('CD0001', 0);

-- realize sensor name's really 'BA0001', not 'AB0001', so create a partition like that
create table "sensor_readings_BA" partition of sensor_readings for values in ('BA');
```

UPDATE tuple routing

```
-- try to update all 'AB0001' entries to reflect that; bad luck in PostgreSQL 10
update sensor_readings set name = 'BA0001' where name = 'AB0001';
ERROR:  new row for relation "sensor_readings_AB" violates partition constraint
DETAIL:  Failing row contains (BA0001, 0).

-- PostgreSQL 11 to the rescue

update sensor_readings set name = 'BA0001' where name = 'AB0001';
select tableoid::regclass, * from sensor_readings;
       tableoid       |  name   | reading
----------------------+---------+---------
 "sensor_readings_BA" | BA0001 |       0
 "sensor_readings_CD" | CD0001 |       0
(2 rows)
```

INSERT ON CONFLICT or upsert

- It's annoying when useful features like upsert don't work whereas you would expect them to,
  - Due to PostgreSQL 10's lack of support of indexes on parent tables, it would outright reject upsert command, because an index is needed
  - Indexes defined on individual partitions don't help, because planner doesn't consider them (INSERT mentions parent table)
- PostgreSQL 11 supports adding indexes on parent tables, which can optionally be UNIQUE
- In other words, PostgreSQL 11 supports upsert!
- Example:

INSERT ON CONFLICT or upsert


```
-- note that there is a unique constraint on id column

create table accounts (id text unique, balance double) partition by hash (id);
create table accounts0 partition of accounts for values with (modulus 4, remainder 0);
create table accounts1 partition of accounts for values with (modulus 4, remainder 1);
create table accounts2 partition of accounts for values with (modulus 4, remainder 2);
create table accounts3 partition of accounts for values with (modulus 4, remainder 3);

insert into accounts values (1, 100);
insert into accounts values (1, 150) on conflict (id) do update set balance =
excluded.balance;
select tableoid::regclass, * from accounts;
 tableoid  | id | balance
-----------+----+---------
 accounts1 | 1  |     150
(1 row)
```

NTT

INSERT ON CONFLICT or upsert


```
-- one more example, just to show that DO NOTHING action works too! :)

create table branches (id int unique, name text) partition by hash (id);
create table branches0 partition of branches for values with (modulus 3, remainder 0);
create table branches1 partition of branches for values with (modulus 3, remainder 1);
create table branches2 partition of branches for values with (modulus 3, remainder 2);
insert into branches values (1, 'SF') on conflict (id) do nothing;
insert into branches values (1, 'SF') on conflict (id) do nothing;
select tableoid::regclass, * from branches;
 tableoid   | id | name
------------+----+------
 branches2 |  1 | SF
```

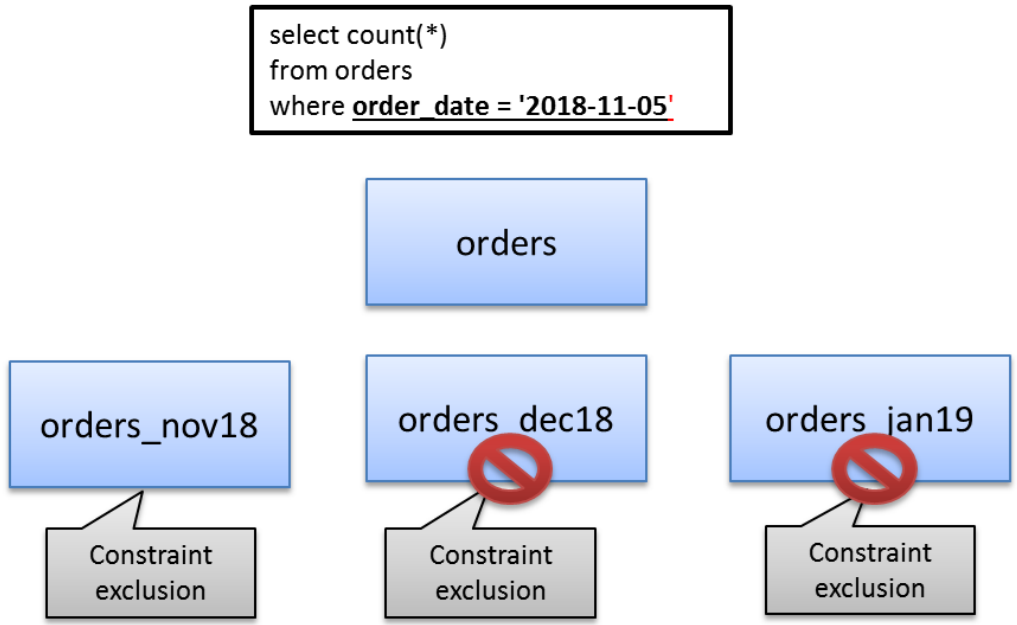
NTT

Optimization features

- Improved partition pruning mechanism
- Run-time partition pruning
- Partitionwise join and aggregation

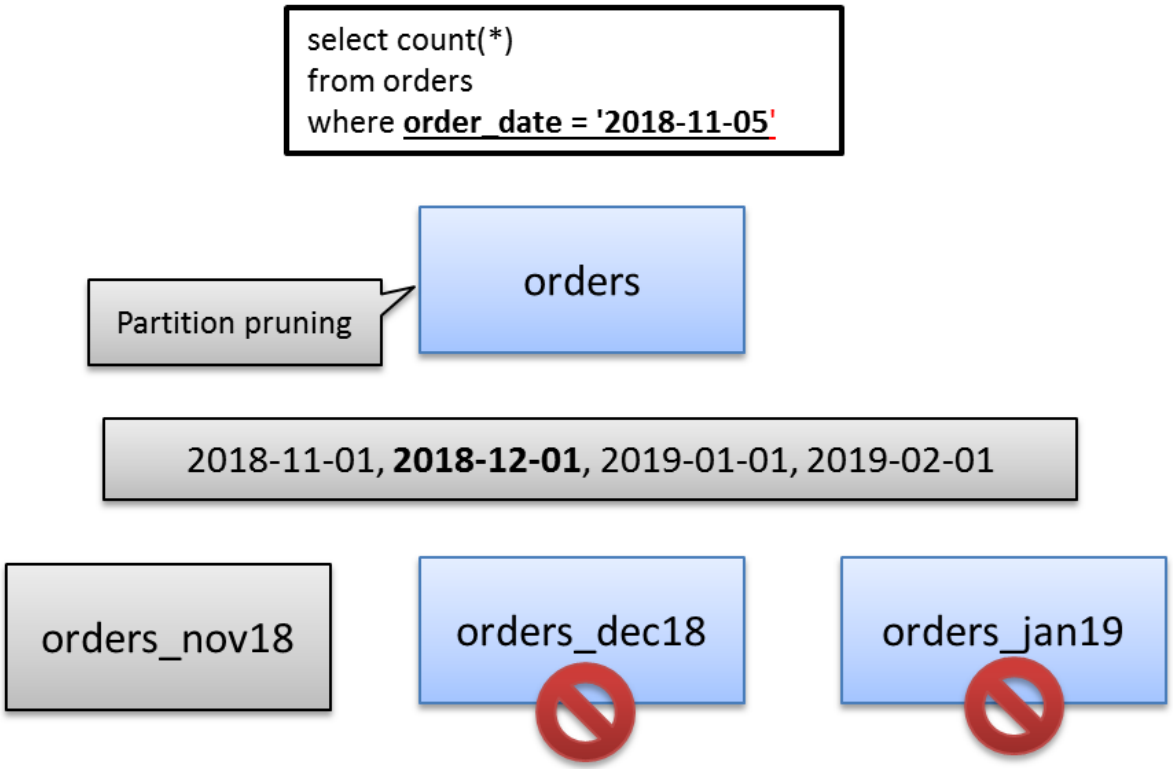
Improved partition pruning mechanism

- With PostgreSQL 10, partition pruning occurs by means of constraint exclusion
  - Need to look at every partition and prune those whose partition constraint contradicts query's WHERE condition
  - Having to look at each partition makes pruning a scalability bottleneck



Improved partition pruning mechanism

- With PostgreSQL 11, looking at parent table is enough to decide which partitions to prune, so pruning is no longer a bottleneck



Run-time pruning or dynamic pruning

- With PostgreSQL 10, pruning can only occur during planning, so the following cases can't use it:
  - Extended query protocol is used (such as with prepared statements), and a generic plan is deemed cheaper
  - When scanning a partitioned table lying on the inner side of nested loop join with join key same as the partition key
  - Partition key is compared to a subquery
- With PostgreSQL 11, planner can now create a plan such that pruning can kick in during execution, so the above cases don’t need to scan all partitions
- Example:

Run-time pruning or dynamic pruning

```
-- firstly, this is what happens on PostgreSQL 10 which lacks this feature

explain (analyze, costs off, timing off) select * from sensor_readings where left(name, 2)
= (select 'BA');
                              QUERY PLAN
-----------------------------------------------------------------------
 Append (actual rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Result (actual rows=1 loops=1)
   ->  Seq Scan on "sensor_readings_AB" (actual rows=0 loops=1)
         Filter: ("left"(name, 2) = $0)
   ->  Seq Scan on "sensor_readings_CD" (actual rows=1 loops=1)
         Filter: ("left"(name, 2) = $0)
   ->  Seq Scan on "sensor_readings_BA" (actual rows=0 loops=1)
         Filter: ("left"(name, 2) = $0)
         Rows Removed by Filter: 1
 Planning time: 0.691 ms
 Execution time: 0.152 ms
(13 rows)
```

Run-time pruning or dynamic pruning

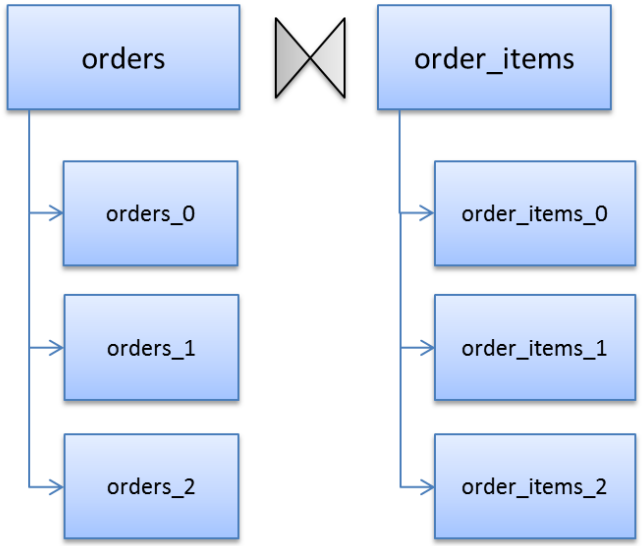
```
-- PostgreSQL 11

explain (analyze, costs off, timing off) select * from sensor_readings where left(name, 2)
= (select 'BA');
                            QUERY PLAN
------------------------------------------------------------------------------
 Append (actual rows=0 loops=1)
   InitPlan 1 (returns $0)
     ->  Result (actual rows=1 loops=1)
   ->  Seq Scan on "sensor_readings_AB" (never executed)
         Filter: ("left"(name, 2) = $0)
   ->  Seq Scan on "sensor_readings_BA" (actual rows=1 loops=1)
         Filter: ("left"(name, 2) = $0)
   ->  Seq Scan on "sensor_readings_CD" (never executed)
         Filter: ("left"(name, 2) = $0)
 Planning Time: 1.641 ms
 Execution Time: 0.315 ms
(11 rows)
```

- There is also a new parameter called enable_partition_pruning to turn pruning off if you want
  - Controls both plan-time and run-time pruning

Partitionwise join and aggregation

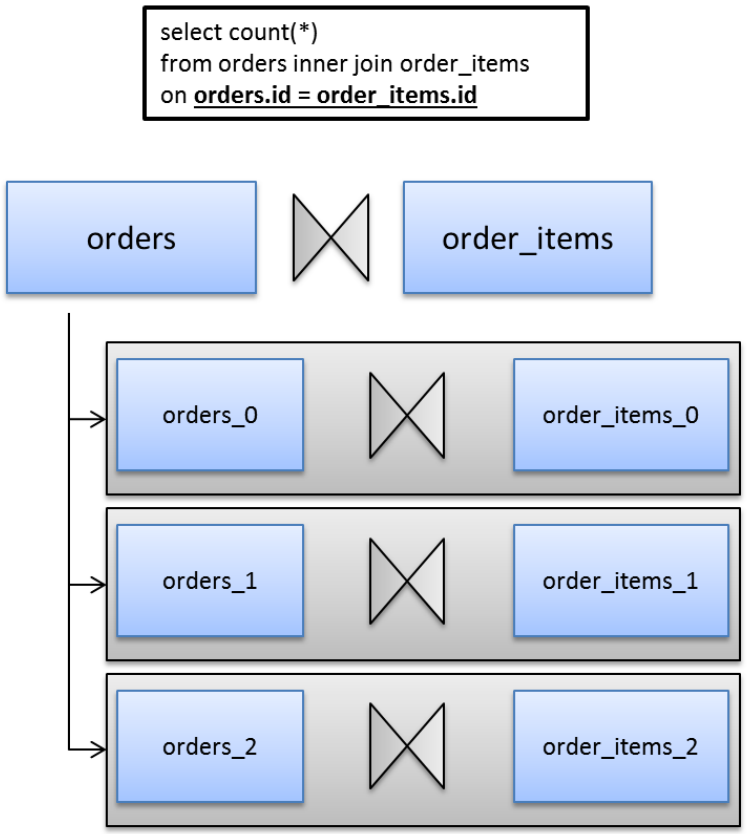
- It's a good strategy in general to push as much work as possible down to individual partitions, because they're smaller than the whole table
  - ➢ Being small means more reliable statistics and ability to use hash-based join or aggregation
- With PostgreSQL 10, join and aggregation have to wait until all partitions are scanned and their outputs combined

```
select count(*)
from orders inner join order_items
on orders.id = order_items.id
```



Partitionwise join and aggregation

- With PostgreSQL 11, both joins and aggregation can be performed at partition level, using techniques called partitionwise join and aggregation, respectively



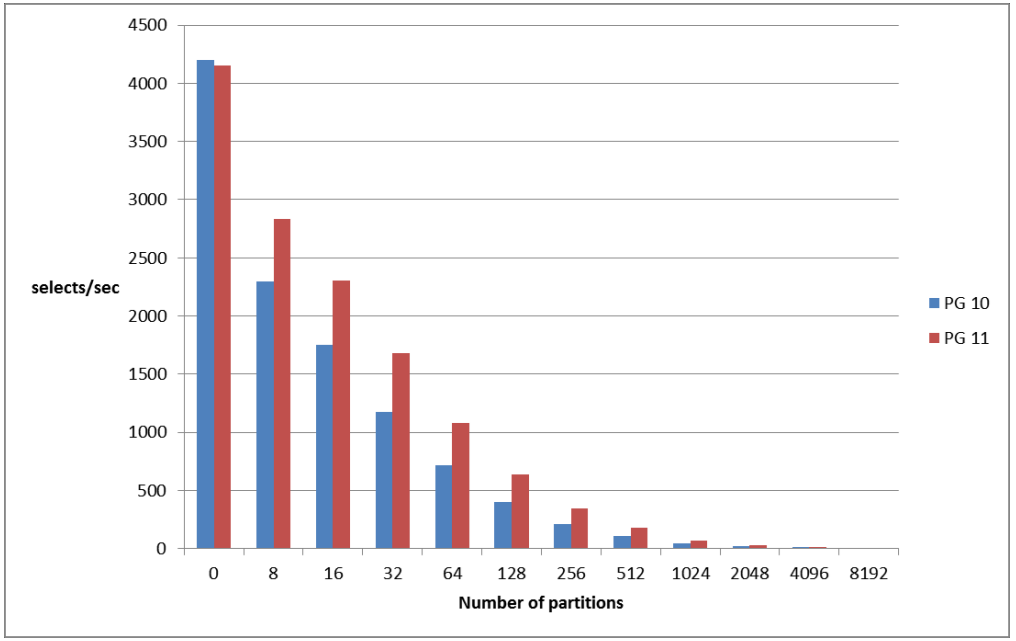
Partitionwise join and aggregation

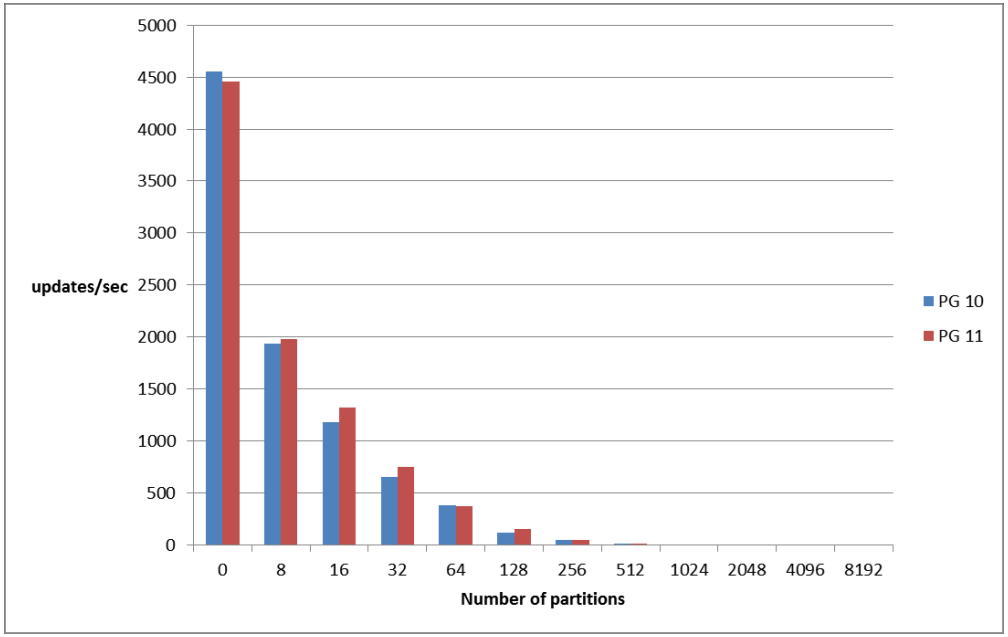
- Partitionwise aggregation is performed in one step per partition, if grouped on the partition key
  - If not grouped or if grouping key doesn't contain partition key, then it is carried out in two stages where a partial aggregate is computed per partition and all partial aggregates are later combined
- Partitionwise join is restricted to the case where both sides have exactly same set of partitions
- Both partitionwise join and aggregation optimizations are currently disable by default
  - To enable, use enable_partitionwise_join and enable_partitionwise_aggregate, respectively

Performance problems

- While it's great that PostgreSQL 11 considerably improved partitioning usability, users still can't go beyond hundreds of partitions without performance degrading significantly

Performance problems

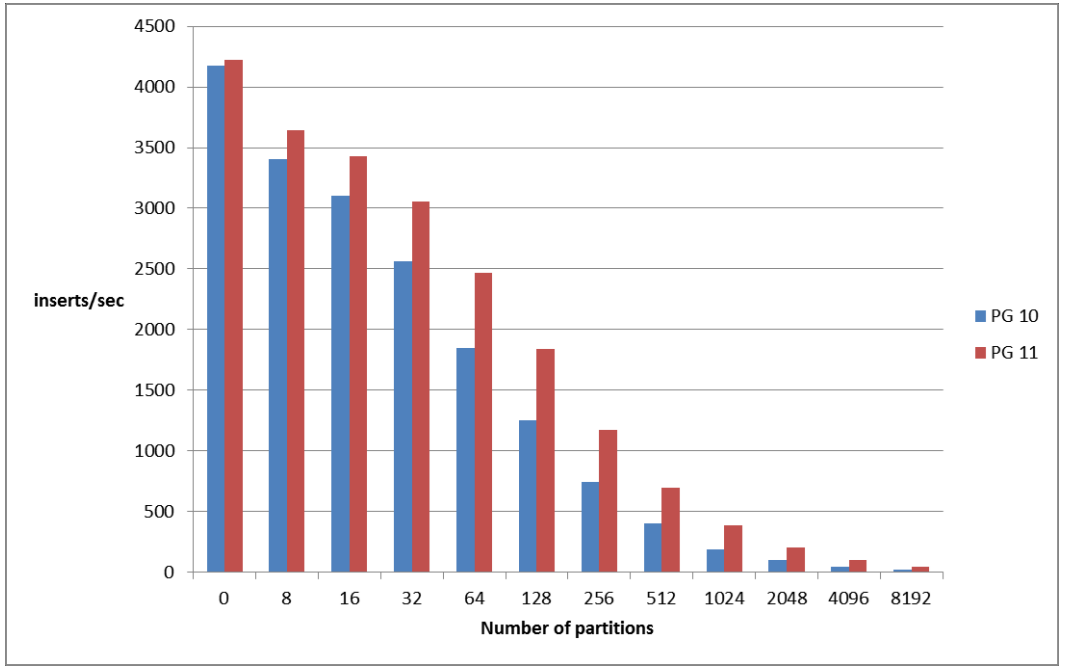




- PostgreSQL 11 slightly improves SELECT performance because it can now use the new pruning mechanism which scales well, but there are other bottlenecks in how planner handles partitions
- UPDATE (and DELETE) performance is same as PostgreSQL 10, because they cannot use the new pruning mechanism

Performance problems

- Here's single-row insert command which affects a single partition



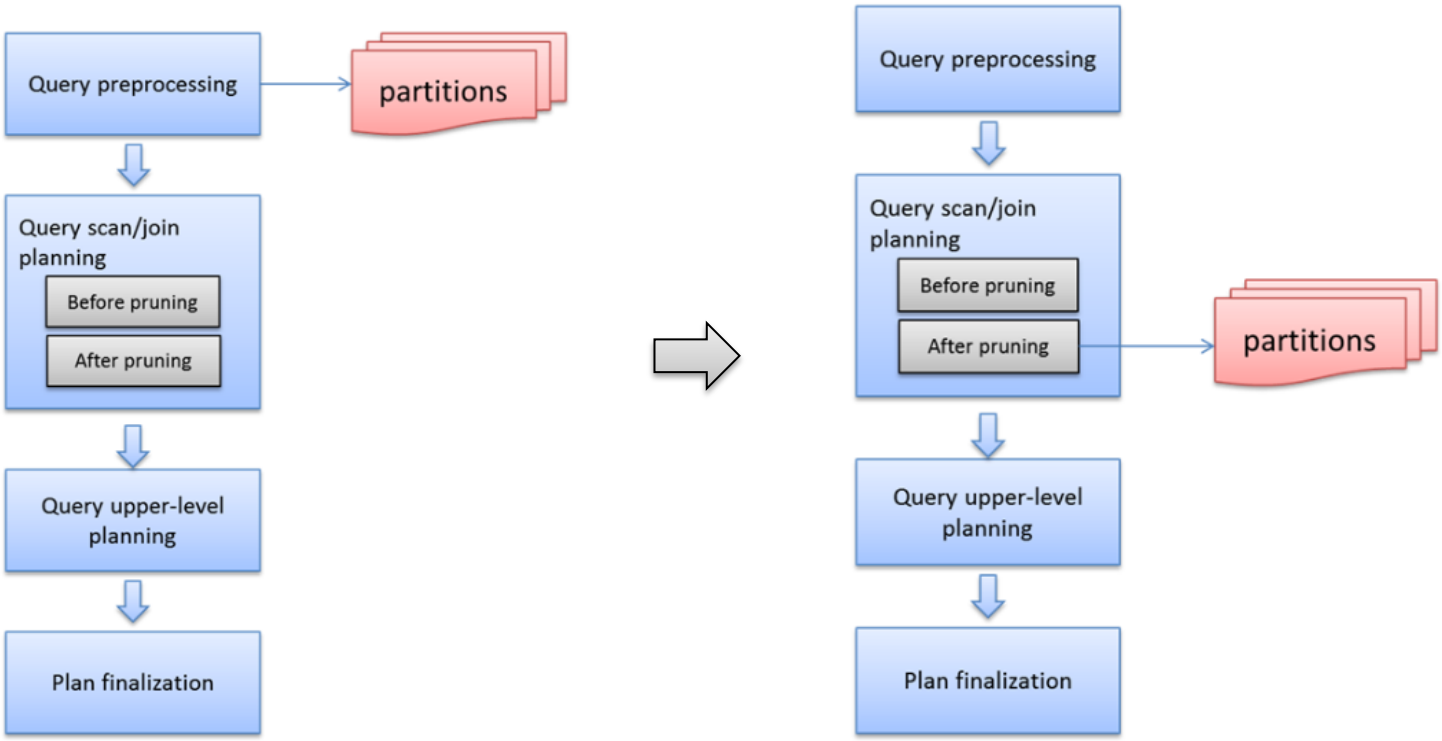
- Again, PostgreSQL 11 performs slightly better due to some improvements to tuple routing code

PostgreSQL 12 ongoing work (performance)

- Some of the patches under development will lead to better performance and scalability

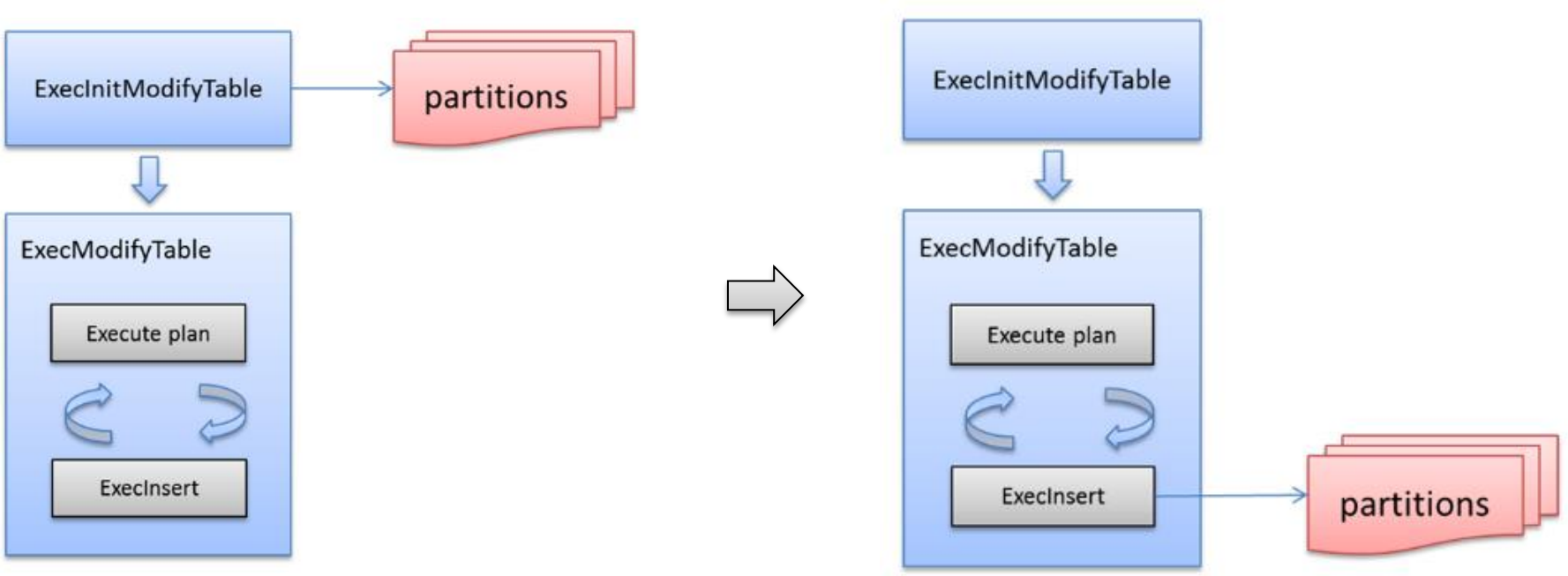
PostgreSQL 12 ongoing work (performance)

- Currently, for selects and updates, partitions are opened long before planner performs pruning
  - ➢ Patches refactor the planner such that partitions can be opened after pruning



PostgreSQL 12 ongoing work (performance)

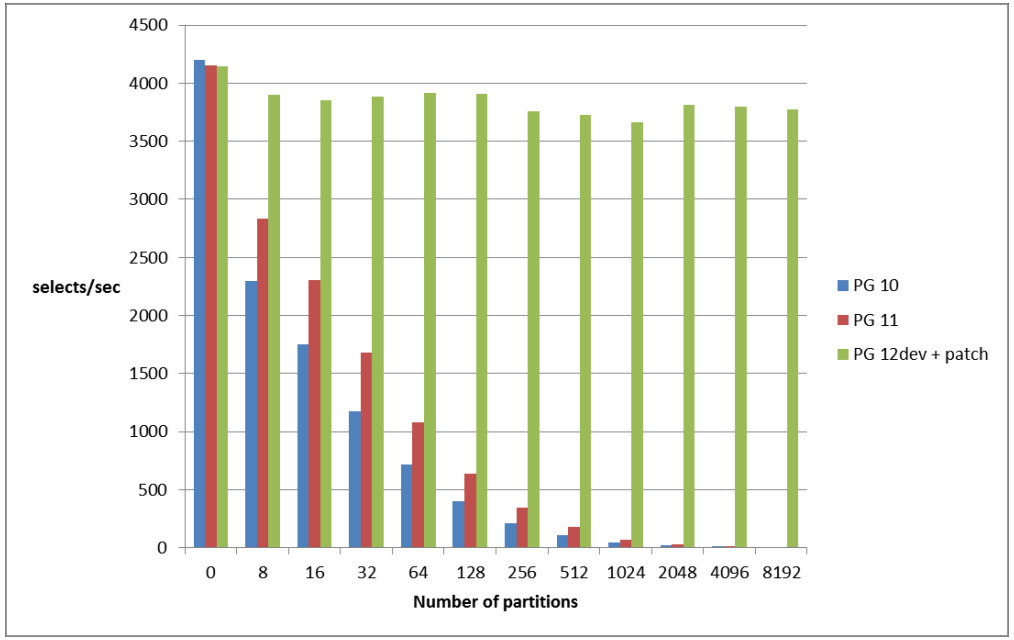
- For inserts, all partitions are locked at the beginning of the execution
  - ➢ Patches modify the execution to lock only the partitions that are affected

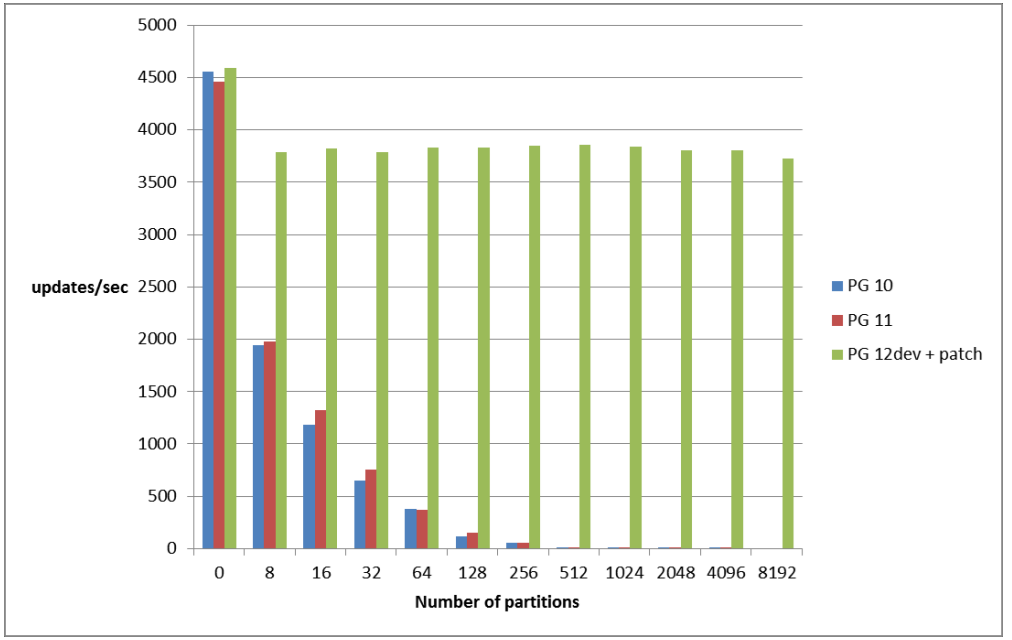


PostgreSQL 12 ongoing work (performance)

Innovative R&D by NTT

- Performance of simple select, update, and insert queries
  - ➢ Figures should not deteriorate too significantly even beyond 8192 partitions



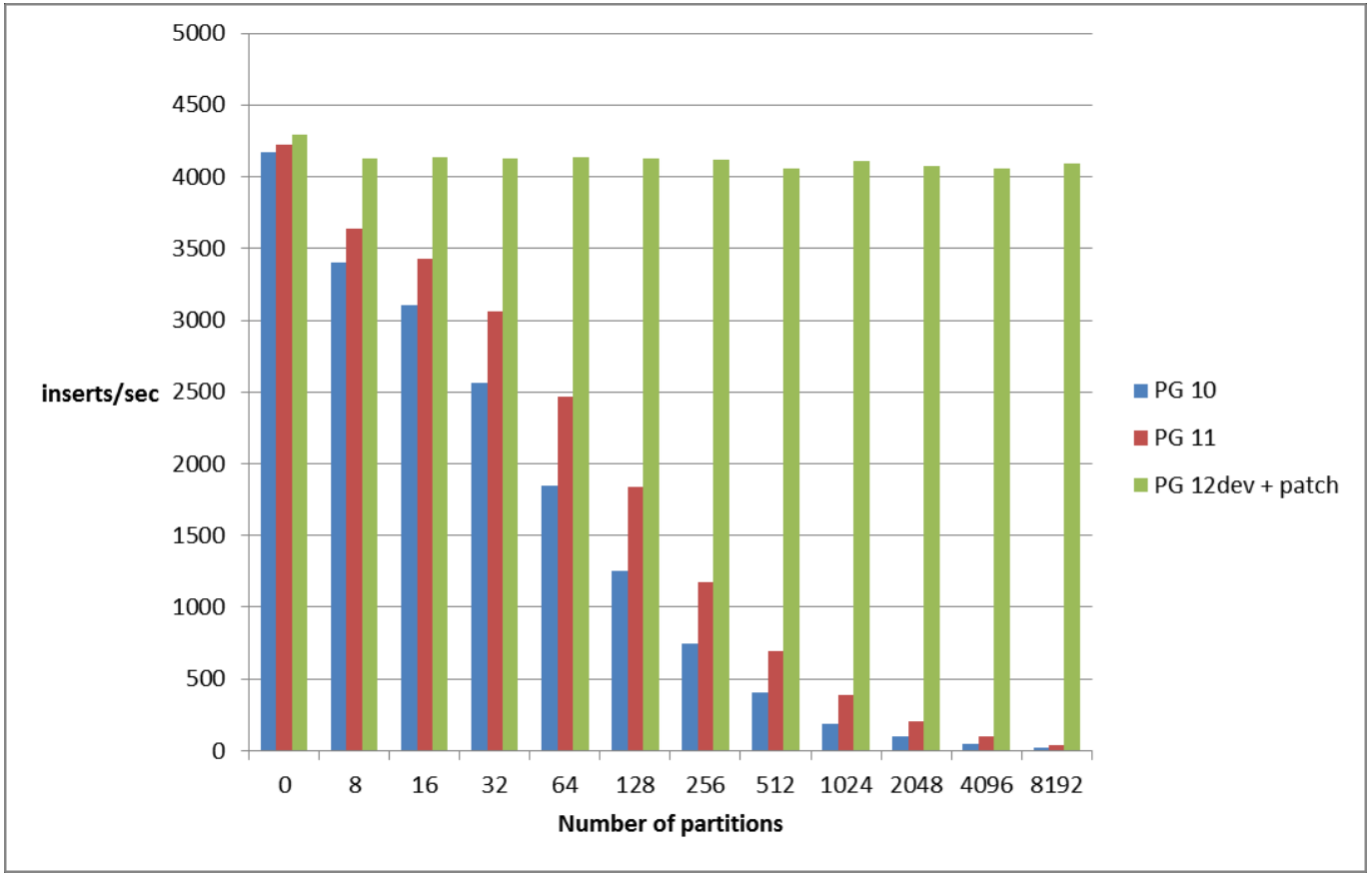


NTT

PostgreSQL 12 ongoing work (performance)

Innovative R&D by NTT

- Single-row Inserts



NTT

PostgreSQL 12 ongoing work (performance)

Innovative R&D by NTT

- Patch to improve planner to speed up and scale better for queries that affect 1 or few partitions
  - https://commitfest.postgresql.org/20/1778/
- Patch to delay locking of partitions
  - https://commitfest.postgresql.org/21/1887/ (insert/update tuple routing)
  - https://commitfest.postgresql.org/21/1897/ (for all queries using generic plans)
- Patch to avoid MergeAppend overhead in some cases when scanning partitions
  - https://commitfest.postgresql.org/21/1850/

NTT

Future (performance)

- Patches so far only optimize the cases where 1 or few partitions are accessed due to pruning
  - There can be situations where planner can't prune, so planner must consider all partitions which can get really slow as the number of partitions increases
  - Existing plan nodes to scan or modify (update/delete) require planner to look at each partition individually to select an optimal scan method for each
  - Maybe, create new plan node(s) that doesn't require planner to look at partitions

Summary

- Declarative partitioning basics
- Limitations of declarative partitioning in PostgreSQL 10
- New partitioning features in PostgreSQL 11
- Performance and scalability problems
- Ongoing and future work for improving performance and scalability

NTT

References

- Partitioning Improvements in PostgreSQL 11 (Robert Haas)
  - https://postgresconf.org/conferences/2018/program/proposals/partitioning-improvements-in-postgresql-11
- Partitioning Improvements in v11 (Alvaro Herrera)
  - https://wiki.postgresql.org/images/8/82/Alvherre-partitioning-2018.pdf
- Partitioning Improvements in PostgreSQL 11 (2ndQuadrant Blog, Alvaro Herrera)
  - https://blog.2ndquadrant.com/partitioning-improvements-pg11/

Thank you!

- Questions?