Partitioning in Postgres
How Far We've Come

Amit Langote
NTT OSS Center
PGConf.EU 2019, Milan, Italy

Slides URL: https://amitlan.github.io/files/pgconf-eu-2019.pdf

About the speaker

- Live and work in Tokyo
- Contribute to community Postgres
- Contributed mainly to the development of declarative partitioning, command progress reporting, among others. In recent releases, worked primarily on improving the performance and the scalability of partitioning

Outline

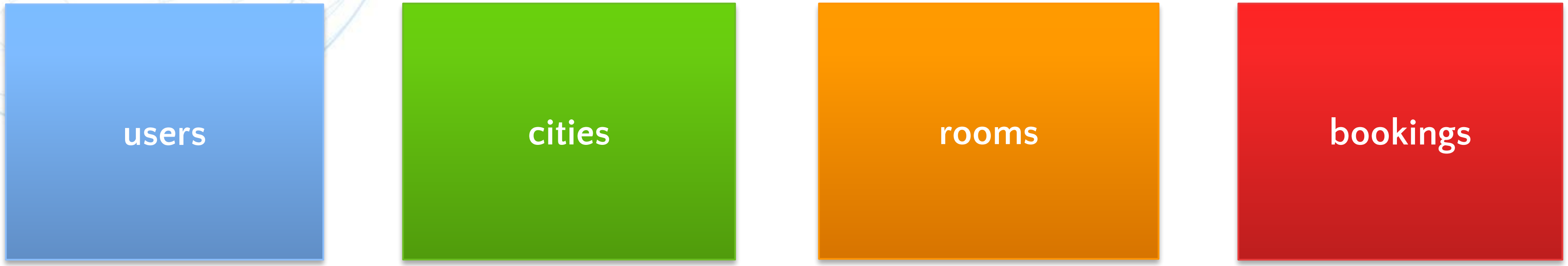
- Partitioning concepts
- Partitioning in Postgres: the “old” way
- The coming of declarative partitioning
- Postgres 10 and 11: foundations
- Postgres 12: performance

Outline

- Partitioning concepts
- Partitioning in Postgres: the “old” way
- The coming of declarative partitioning
- Postgres 10 and 11: foundations
- Postgres 12: performance

Data growth

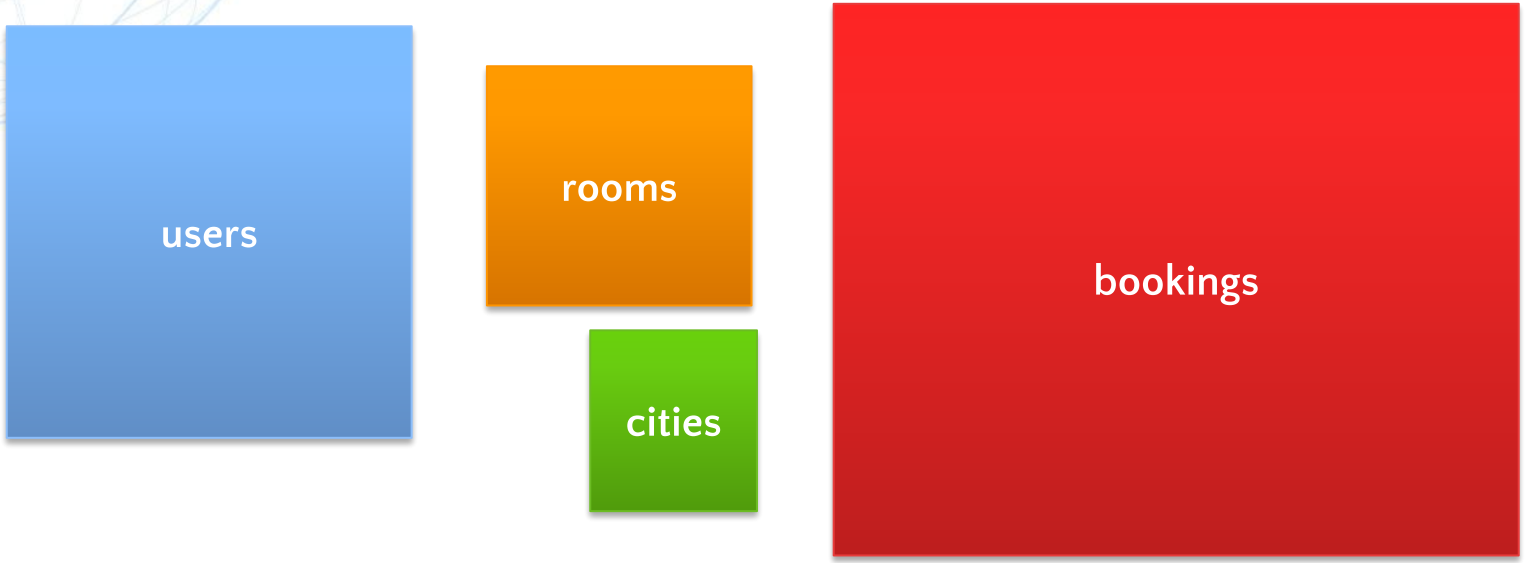
- Non-trivial applications accumulate and access data, usually with the help of a database system. As an example, consider an application that allows users to browse and rent rooms in different cities.



- Irrespective of the data model, data corresponding to certain modeled entities may grow pretty quickly, which makes the operations on those data slower

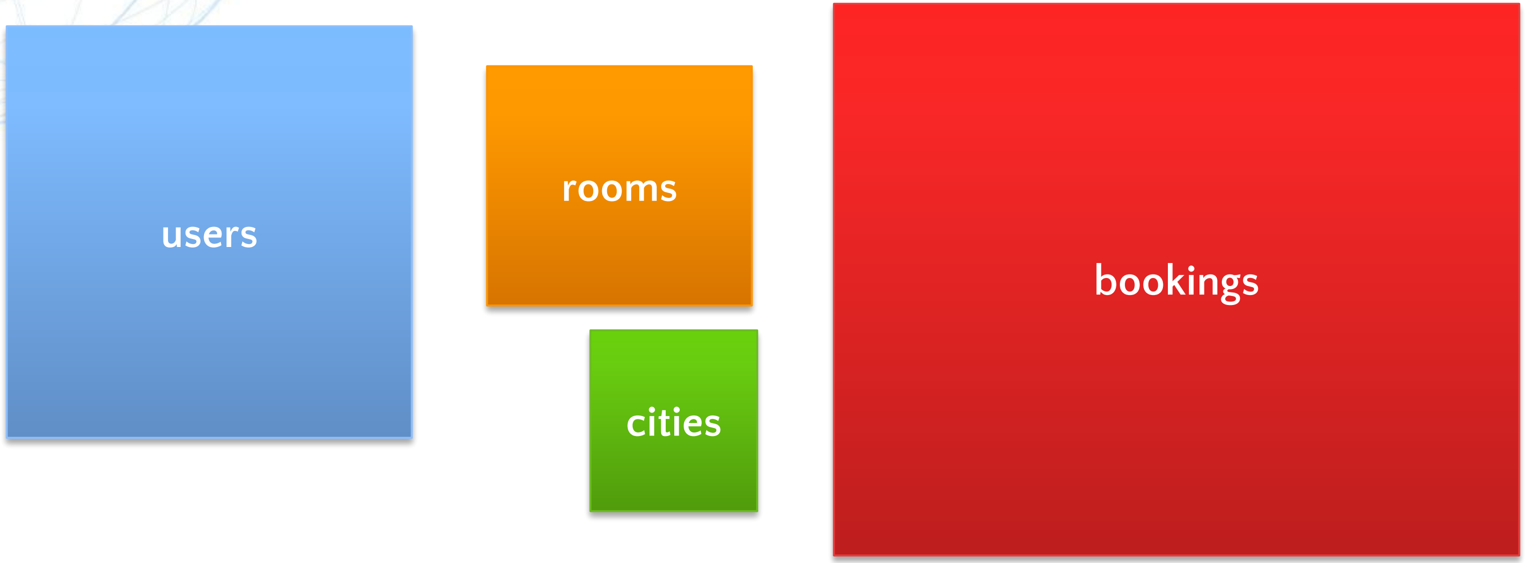
Data growth

- Assuming relational model, this means tables for certain entities growing too big for the current configuration of the database. So for example, actions which require looking up a user or updating a booking will become slower, because the database has to process ever growing amount of data unrelated to a given request



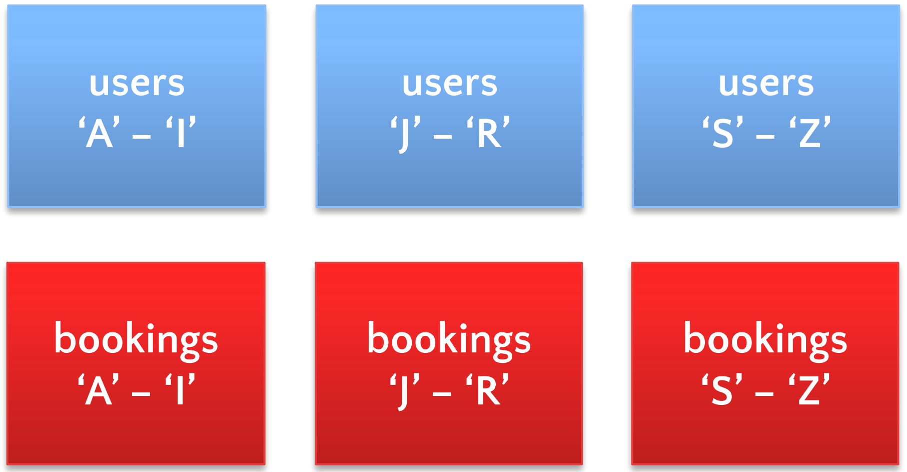
Dealing with data growth

- Up to a point, administrators can get away with various tricks like upgrading the hardware, configuring more resources for database operation, upgrading database software to get better performance and so on



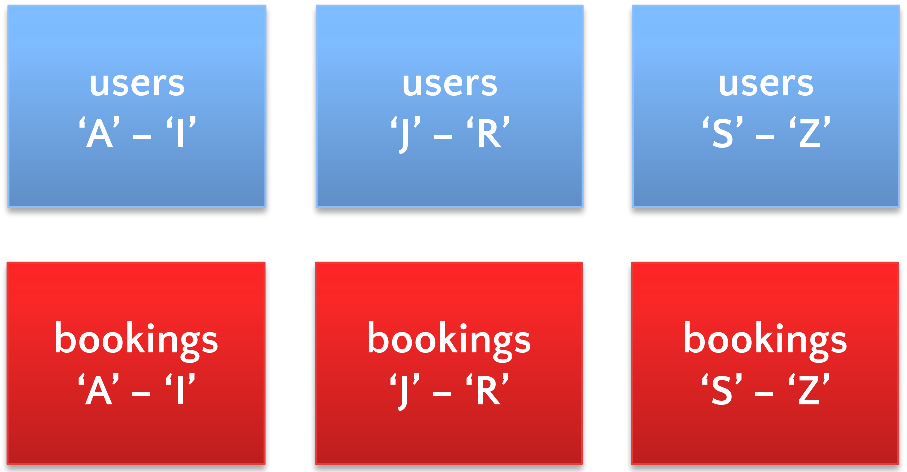
Dealing with data growth: Partitioning

- Another time-tested option is to use partitioning
- What is logically one entity in the application is managed as multiple, individually-addressable objects in the database system. In our example, the big users table can be broken down into smaller tables, each containing an application-defined subset of the data



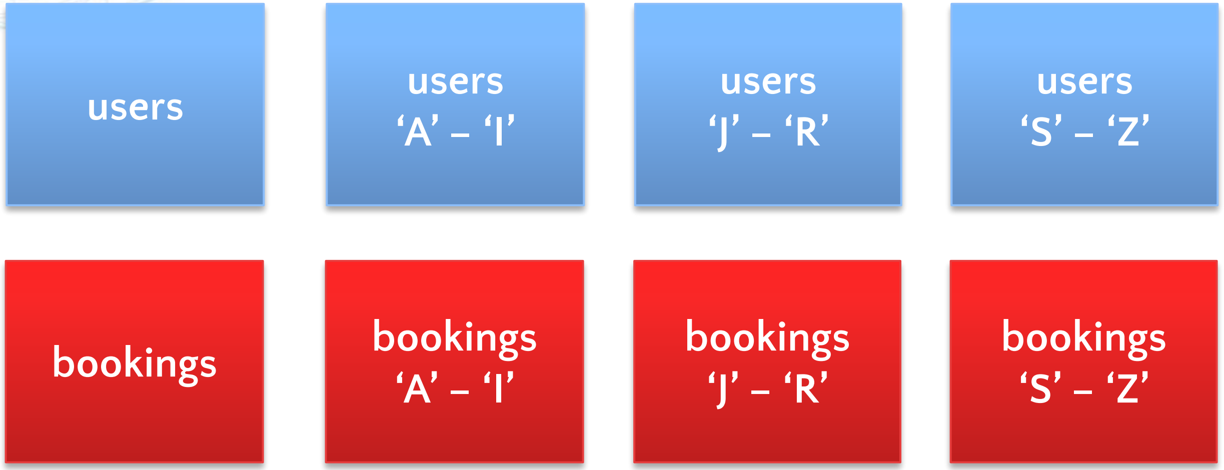
Partitioning

- Great thing about partitioning is that it allows the application to manipulate only the partitions of interest. For example, to look up a user whose name starts with 'A', the application may issue the operation to only the relevant partition.
- Partitioning can be defined and implemented entirely in the application code and the database system only has to deal with multiple smaller tables.



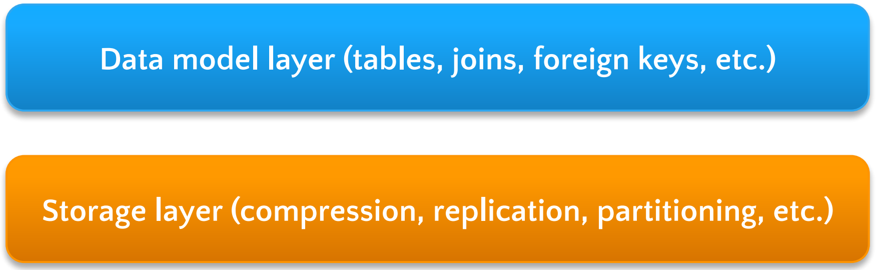
Partitioning

- Or partitioning could be local to the database, that is, the application continues to refer to the original table name, which is mapped by the database system to its partitions
- In this approach, the partitioning is still defined by the application code and most of the functionality implemented in the database.



Partitioning

- Some natively distributed database systems also contain a concept of partitioning that is wholly database-controlled.
- Such database systems are typically implemented in layers. For example, the layer that implements data model is separate from the layer that implements storage and the application typically only interacts with the data model layer

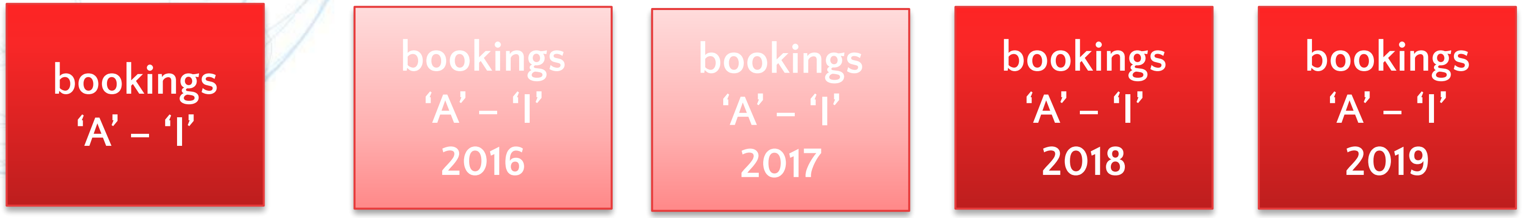

Data model layer (tables, joins, foreign keys, etc.)
Storage layer (compression, replication, partitioning, etc.)

- Partitioning in this case is implemented at the storage layer and may not always be tunable by the application

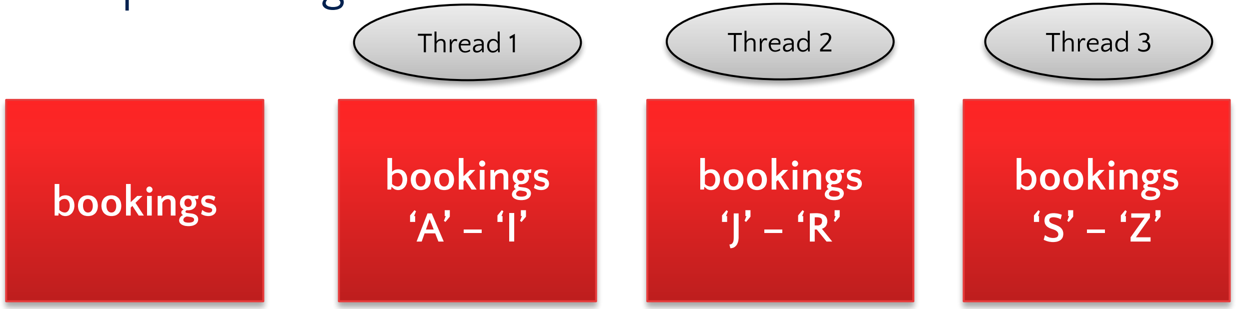
Partitioning

There are other benefits to doing partitioning:

- Efficient archiving.



- Parallel processing

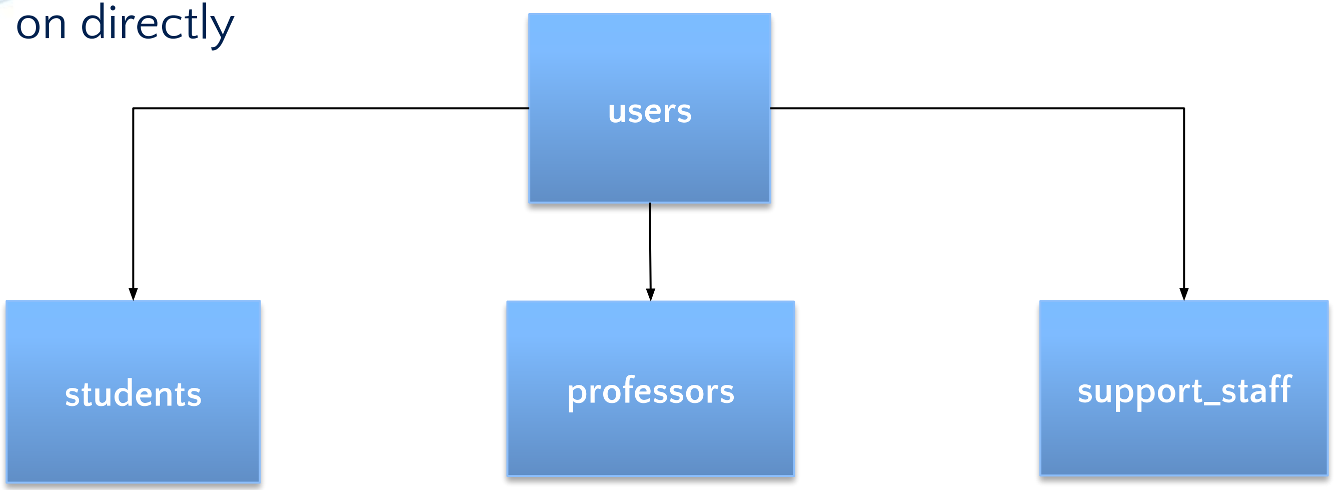


Outline

- Partitioning concepts
- Partitioning in Postgres: the “old” way
- The coming of declarative partitioning
- Postgres 10 and 11: foundations
- Postgres 12: performance

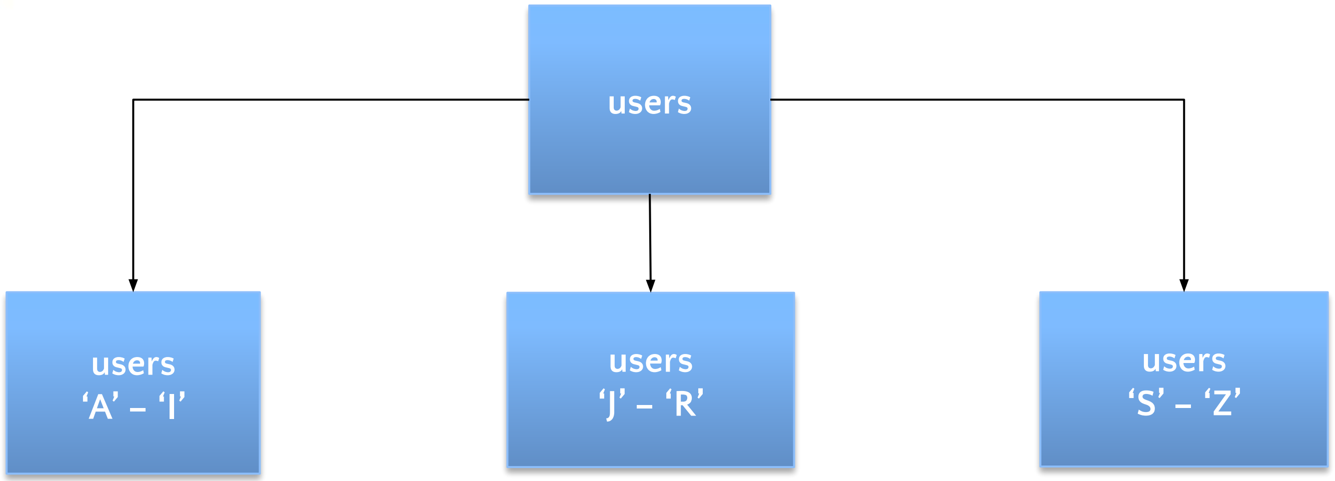
Partitioning in Postgres: the “old” way

- Postgres has long supported in-database partitioning, even though the main optimization for partitioning came around much later (14 years ago) when such workloads started appearing in the Postgres wild
- It’s based on table inheritance, a feature to group related tables by making them all inherit from the same parent table
- Operations on the parent table implicitly affect children, although children can be operated on directly



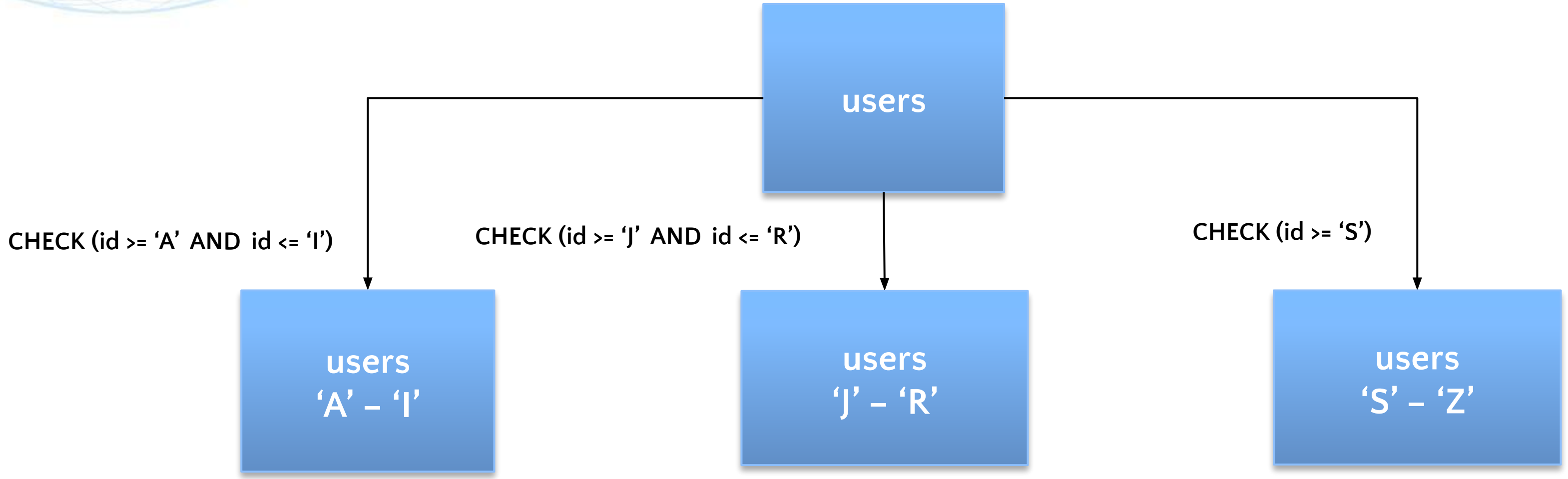
Partitioning in Postgres: the “old” way

- Partitions form a group of related tables (same schema, different subsets of data)
- Parent table acts like an abstract class, whereas partitions contain the actual data
- The application issues operations on the parent table which Postgres internally applies to the child tables



Partitioning in Postgres: the “old” way

- Postgres can avoid processing irrelevant child tables with some additional setup
- To do so, the application needs to describe, using a CHECK constraint defined on each child table, the subset of the total data that the table contains
- If the query’s restrictions contradict the table’s CHECK constraint, it won’t be scanned
- This feature is called *constraint exclusion* and is present in Postgres since v8.1

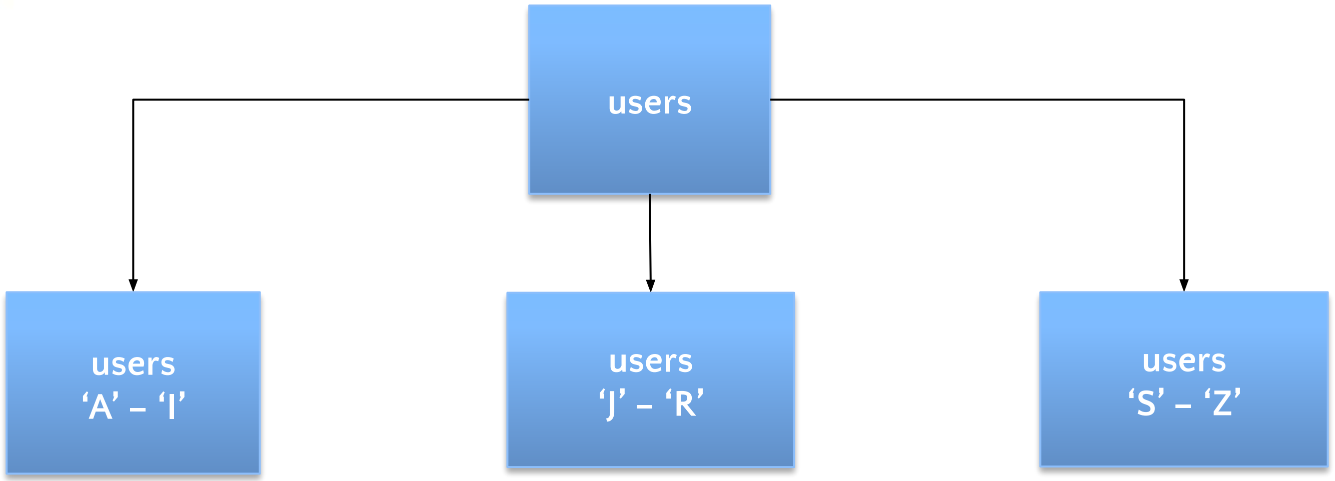


Partitioning in Postgres: the “old” way

- While selecting data from partitions is transparent to the application, inserting isn’t, that is, data needs to be distributed among partitions using application logic
- It’s typical to use a database trigger – define a trigger on the parent table that catches any INSERTs done on it and the executed code performs the INSERT on the correct child table instead of the parent table

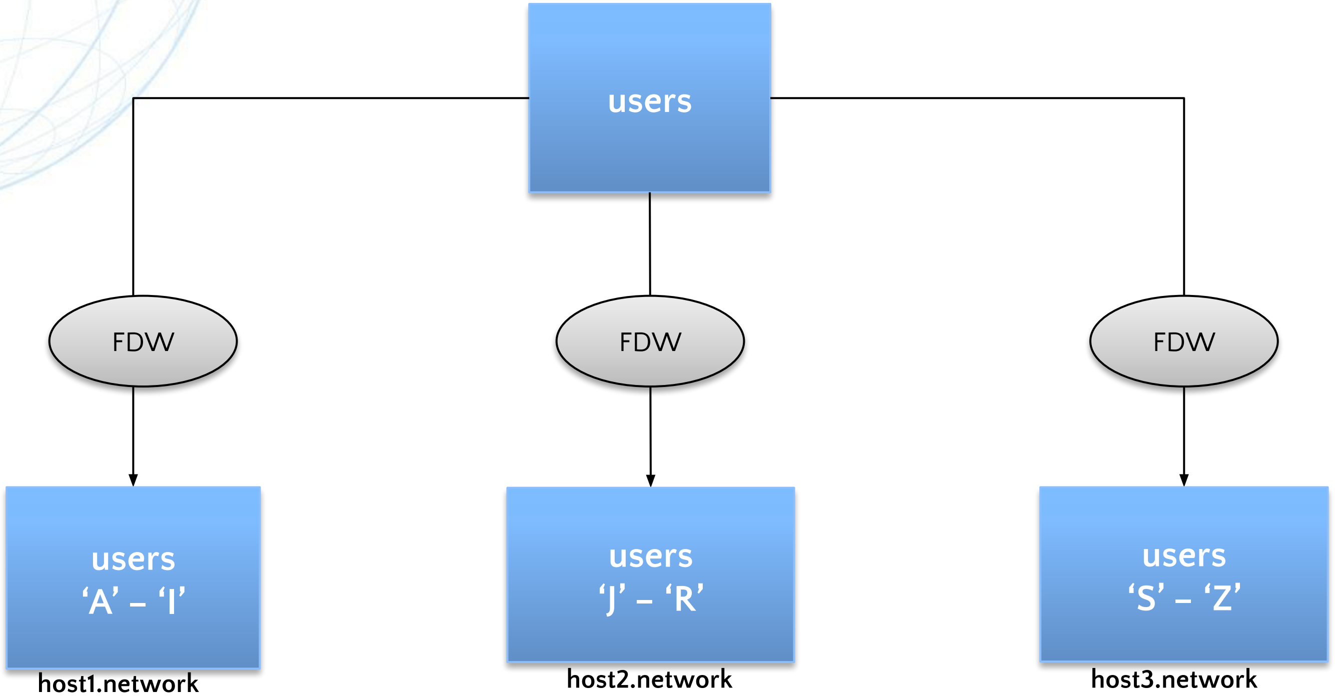
Partitioning in Postgres: the “old” way

```
CREATE TRIGGER users_insert_redirect
BEFORE INSERT ON users
FOR EACH ROW
EXECUTE PROCEDURE users_insert_redirect()
```



Partitioning in Postgres: the “old” way

- The child tables can also be foreign tables (starting in v9.5 released in 2016), so it allows the partitions to span multiple machines, allowing for a primitive form of scale-out



Partitioning in Postgres: the “old” way

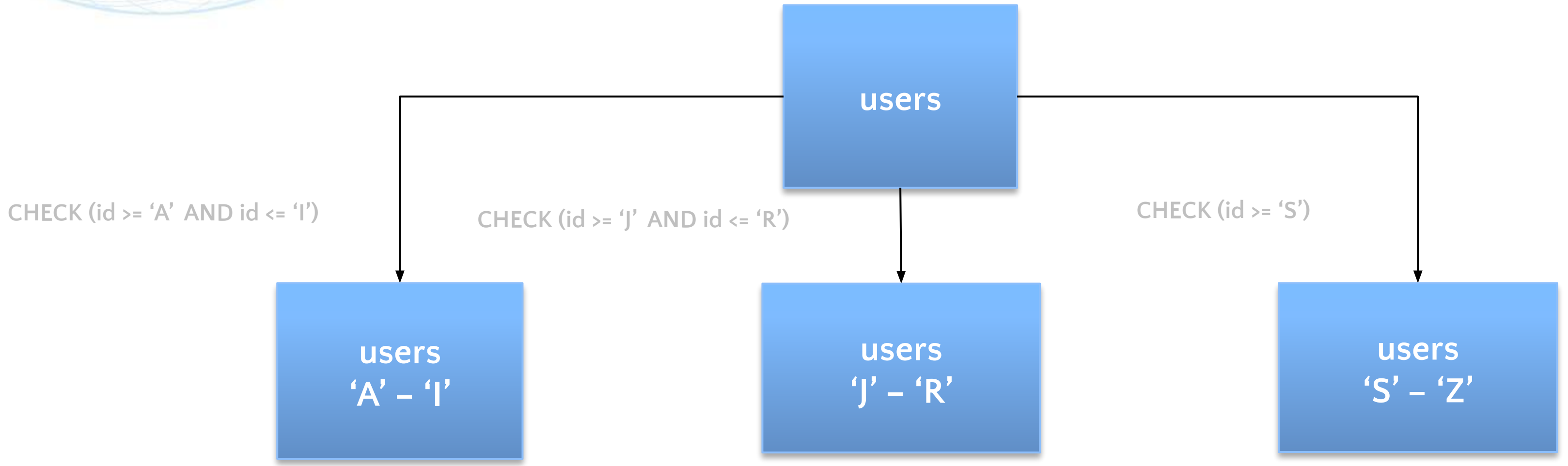
The “old” way gets the job done, but is inefficient in various ways:

- Poor usability, because the application still has to bear most of the responsibility for making sure that partitioning is set up correctly
- Poor performance, especially as the number of partitions increases, because the architecture of query processing in Postgres is not really fine-tuned for manipulating many tables in the handling of a given query

Outline

Declarative partitioning

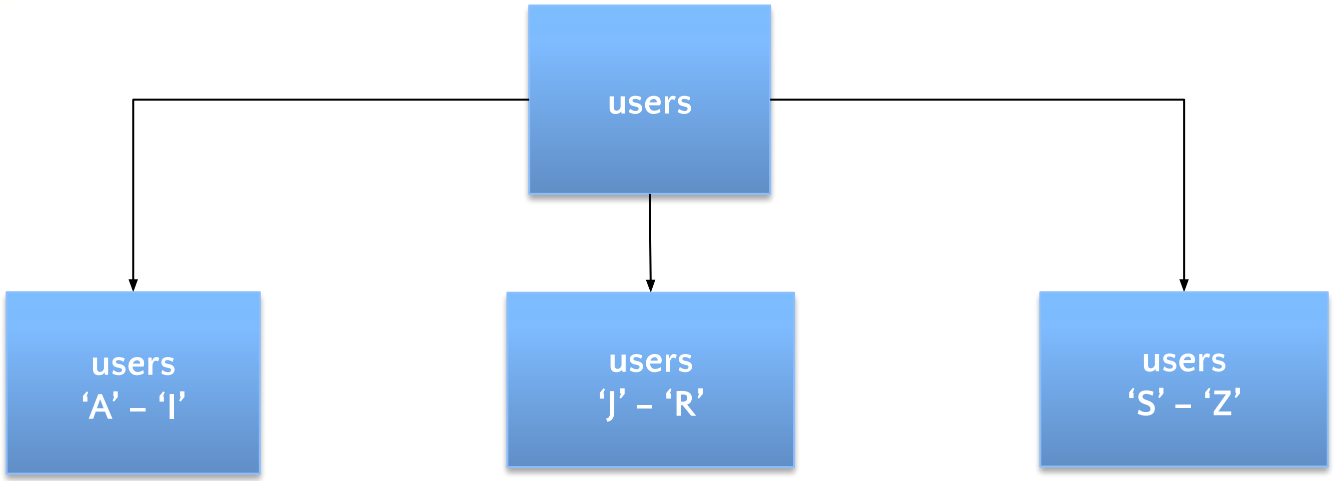
- It's the same old architecture, but with a new syntax to define partitions
- You, the application developer, are still responsible for deciding what partitions to create and also for creating them using the new commands (CREATE TABLE + partition clause)
- Postgres takes care of the rest!
- No need to create CHECK constraints



Declarative partitioning

- There's no need for the trigger too

```
CREATE TRIGGER users_insert_redirect
BEFORE INSERT ON users
FOR EACH ROW
EXECUTE PROCEDURE
users_insert_redirect()
```



Declarative partitioning

- That’s really all there is to the “declarative” qualifier – the new syntax and the out-of-the-box enforcement of partitioning
- Most of the buzz around declarative partitioning has really to do with getting the stuff that used to work on normal tables to also work with partitioning, rather than, say, improving the architecture of partitioning
- The bright side is that Postgres can use partitioning metadata to better optimize queries over declarative partitions compared to “old”-style partitions, which are optimized with generic tricks like constraint exclusion
- The number of declarative partitions that can be reasonably handled is also bigger, because the newly enabled optimizations allow to paper over some architectural limitations with processing many tables
- Maybe we need to address those fundamental architectural limitations head on, now more than ever

Outline

Postgres 10

- Postgres 10 introduced declarative partitioning, with the basics:
  - The syntax for RANGE and LIST partitioning
  - Commands to “attach”, “detach” partitions
  - Multi-level partitioning
  - Automatic enforcement of partition constraint
  - INSERT and COPY (except for foreign table partitions)

Postgres 10

```
-- partitioned table
CREATE TABLE users (id text, country text, ...) PARTITION BY RANGE (id);

-- partition (empty)
CREATE TABLE users_a_to_i PARTITION OF users FOR VALUES FROM ('a') TO ('j’);

-- or “attach” existing table as partition
CREATE TABLE users_a_to_i (LIKE users);
COPY users_a_to_i FROM ‘users_a_to_i.csv’ CSV;
ALTER TABLE users ATTACH PARTITION users_a_to_i FOR VALUES IN ('japan');

-- multi-level partitioning
CREATE TABLE users_j_to_r PARTITION OF users FOR VALUES FROM ('j') TO ('s') PARTITION BY LIST
(country);
CREATE TABLE users_j_to_r_japan PARTITION OF users_j_to_r FOR VALUES IN ('japan');
```

Postgres 10

```
-- dropping a partition
DROP TABLE users_a_to_i;

-- or “detach” to keep the data in partition around
ALTER TABLE users DETACH PARTITION users_a_to_i;

-- partition constraint
INSERT INTO users_j_to_r_japan VALUES ('jun', 'india');
ERROR:  new row for relation "users_j_to_r_japan" violates partition constraint
DETAIL:  Failing row contains (jun, india).
```

Postgres 10

```
-- insert and copy on partitioned table
INSERT INTO users VALUES ('jun', 'Japan');
COPY users FROM stdin CSV;
>> rin,japan
>> Ctrl-D

SELECT * FROM users_j_to_r_japan;
 name | country
------+---------
 jun  | Japan
 rin  | japan
(2 rows)

-- partitions must be defined for all anticipated data
INSERT INTO users VALUES ('jun', 'india');
ERROR:  no partition of relation "users_j_to_r" found for row
DETAIL:  Partition key of the failing row contains (lower(country)) = (india).
```

Postgres 10

- Setting up partitioning consists of much less chores, when compared with “old”-style partitioning
- Bulk inserts are at least an order of magnitude faster than with a plpgsql trigger
- Some pretty unintuitive limitations though
  - Can’t create indexes, UNIQUE constraints, foreign key constraints, row-level triggers on partitioned tables, even though they can be defined on the individual partitions
- No performance improvements over “old”-style for select queries
- Other limitations stem from the expectations that people have developed after working for many years with the “old”-style partitioning and having to solve a few issues by themselves, like:
  - Transparent handling of cross-partition UPDATE operations
  - Automatic creation of partitions for new keys or a “default” partition that would capture keys for which there is no partition defined

Postgres 11

- A pretty significant release for partitioning
- Most of the restrictions on partitioned table DDL are lifted:
  - Indexes can be defined (*locally-partitioned indexes*)
  - UNIQUE constraints can be defined, provided the UNIQUE key includes the partition key
  - Foreign keys can be defined, although foreign keys cannot point to partitioned tables
  - Row-level triggers can be defined

Postgres 11

```
-- partitioned index
CREATE INDEX ON users (country);
```

This creates what's known as locally-partitioned index, meaning the index covers only the values of a given partition and each partition gets one

```
\d users_j_to_r_japan
        Table "public.users_j_to_r_japan"
 Column  | Type | Collation | Nullable | Default
---------+------+-----------+----------+---------
 name    | text |           |          |
 country | text |           |          |
Partition of: users_j_to_r FOR VALUES IN ('japan')
Indexes:
    "users_j_to_r_japan_country_idx" btree (country)
```

Postgres 11

```
-- needless to say, any future partitions would automatically inherit all indexes
CREATE TABLE users_j_to_r_india PARTITION OF users_j_to_r FOR VALUES IN ('india');
\d users_j_to_r_india
           Table "public.users_j_to_r_india"
   Column    |  Type   | Collation | Nullable | Default
────────────┼─────────┼───────────┼──────────┼─────────
 id         | text    |           |          |
 country    | text    |           |          |
 account_id | integer |           |          |
Partition of: users_j_to_r FOR VALUES IN ('india')
Indexes:
    "users_j_to_r_india_id_idx" btree (id)

-- inherited indexes can't be dropped, because of course
DROP INDEX users_j_to_r_india_id_idx;
ERROR:  cannot drop index users_j_to_r_india_id_idx because index users_j_to_r_id_idx requires it
HINT:  You can drop index users_j_to_r_id_idx instead.
```

Postgres 11

```
-- UNIQUE constraint
CREATE UNIQUE INDEX ON users (id, country);
Or
ALTER TABLE users ADD CONSTRAINT unique_user_id_country UNIQUE (id, country);
```

Because the index that underlies the unique constraint is locally-partitioned, it can only ensure uniqueness at the individual partition level. However, by using the partition key as the UNIQUE key, global uniqueness is ensured because the partitioning distributes data into non-overlapping sets, each of which is guarded by a UNIQUE index

```
-- UNIQUE constraint must include partition keys at all levels
CREATE UNIQUE INDEX ON users(id);
ERROR:  insufficient columns in UNIQUE constraint definition
DETAIL:  UNIQUE constraint on table "users_j_to_r" lacks column "country" which is part of the
partition key.
```

Postgres 11

```
-- FOREIGN KEY constraint
CREATE TABLE accounts (id int, ..., PRIMARY KEY (id));
ALTER TABLE users ADD CONSTRAINT user_account_foreign_key FOREIGN KEY (account_id) REFERENCES accounts;
```

Although, foreign keys can't reference partitioned tables.

```
CREATE TABLE accounts (id int, ..., PRIMARY KEY (id)); PARTITION BY RANGE (id);
ALTER TABLE users ADD CONSTRAINT user_account_foreign_key FOREIGN KEY (account_id) REFERENCES accounts;
ERROR:  cannot reference partitioned table "accounts"
```

Postgres 11

- There are new partitioning features too, such as:
  - HASH partitions
  - DEFAULT partition (boundless partition)
  - Transparent handling of cross-partition UPDATE
  - INSERT/COPY to foreign table partitions

Postgres 11

```
-- hash partitioned table
CREATE TABLE users (id text, country text, ...) PARTITION BY HASH (id);

CREATE TABLE users1 PARTITION OF users FOR VALUES WITH (MODULUS 4, REMAINDER 0) PARTITION BY LIST
(country);
CREATE TABLE users1_japan PARTITION OF users1 FOR VALUES IN ('japan');
-- default partition
CREATE TABLE users1_default PARTITION OF users1 DEFAULT;

CREATE TABLE users2 PARTITION OF users FOR VALUES WITH (MODULUS 4, REMAINDER 1) PARTITION BY LIST
(country);
CREATE TABLE users2_japan PARTITION OF users2 FOR VALUES IN ('japan');
CREATE TABLE users2_default PARTITION OF users2 DEFAULT;

so on...
```

Postgres 11

```
-- transparent cross-partition update

insert into users values ('rin', 'japan');
insert into users values ('jun', 'japan');
select tableoid::regclass, * from users;
   tableoid    | id  | country
───────────────┼─────┼─────────────
 users2_japan  | rin | japan
 users3_japan  | jun | japan
(2 rows)

update users set country = 'india' where id = 'rin';
select tableoid::regclass, * from users;
    tableoid     | id  | country
─────────────────┼─────┼─────────────
 users2_default  | rin | india
 users3_japan    | jun | japan
(2 rows)
```

Postgres 11

- The following new techniques are now applied when processing queries involving partitions.
  - Partition pruning, supersedes constraint exclusion
  - Apply partition pruning during execution, in addition to during planning
  - Partition-wise join
  - Partition-wise aggregate

Postgres 11

- The usability of the “new” partitioning has far surpassed that of the “old”-style partitioning with the release of Postgres 11, with new features that would be hard or outright impossible to emulate with the latter
- However, as noted earlier, the basic architecture that’s being used hasn’t changed much, which limits the number of partitions that can be used to somewhere around low 100s, because any given query would need to touch them all
- This results in an unacceptable performance, especially for point queries on partitioned tables, which are ideally handled by touching only the relevant partitions

Outline

- Partitioning concepts
- Partitioning in Postgres: the “old” way
- The coming of declarative partitioning
- Postgres 10 and 11: foundations
- Postgres 12: performance

Postgres 12

- Although a lot of effort was focused on refactoring the partitioning code base to reduce partitioning overheads for common operations, a bunch of new features have landed:
  - Foreign keys can now reference partitioned tables
  - Partition bound syntax now allows specifying arbitrary expressions, in addition to just literal values that the earlier syntax allowed
  - More intuitive handling of tablespace assigned to partitioned tables
  - psql commands for better listing of partitions
  - Collection of functions to introspect partition hierarchy

Postgres 12

```
-- FOREIGN KEY referencing a partitioned table
CREATE TABLE accounts (id int, ..., PRIMARY KEY (id)); PARTITION BY RANGE (id);
ALTER TABLE users ADD CONSTRAINT user_account_foreign_key FOREIGN KEY (account_id) REFERENCES accounts;

-- partition bound can by any expression
CREATE TABLE sensors (id text) PARTITION BY LIST (id);
CREATE TABLE sensor1 PARTITION OF sensors FOR VALUES IN ('sense_' || nextval('sensor_next_id'));
CREATE TABLE sensor2 PARTITION OF sensors FOR VALUES IN ('sense_' || nextval('sensor_next_id'));
\d+ sensors
Partition key: LIST (id)
Partitions: sensor1 FOR VALUES IN ('sense_1'),
            sensor2 FOR VALUES IN ('sense_2')

-- In v11 and earlier
CREATE TABLE sensor1 PARTITION OF sensors FOR VALUES IN ('sense' || nextval('sensor_next_id'));
ERROR:  syntax error at or near "||"
LINE 1: ...nsor1 PARTITION OF sensors FOR VALUES IN ('sense' || nextval...
                                                             ^
```

Postgres 12

```
-- tablespace and partitioning
CREATE TABLESPACE tmpspace LOCATION '/tmp/pg-sensor-parts';
ALTER TABLE sensors SET TABLESPACE tmpspace;
CREATE TABLE sensor3 PARTITION OF sensors FOR VALUES IN ('sense_' || nextval('sensor_next_id'));
\d sensor3
...
Partition of: sensors FOR VALUES IN ('sense_3')
Tablespace: "tmpspace"

-- In v11 and earlier
ALTER TABLE sensors SET TABLESPACE tmpspace;
CREATE TABLE sensor3 PARTITION OF sensors FOR VALUES IN ('sense_3');
\d sensor3
...
Partition of: sensors FOR VALUES IN ('sense_3')
```

Postgres 12

```
\dP
                 List of partitioned relations
 Schema |         Name          | Owner |       Type        | Table
--------+-----------------------+-------+-------------------+-------
 public | sensors               | amit  | partitioned table |
 public | users                 | amit  | partitioned table |
 public | users_id_country_idx  | amit  | partitioned index | users
(3 rows)
```

```
\dPt
List of partitioned tables
 Schema |  Name   | Owner
--------+---------+-------
 public | sensors | amit
 public | users   | amit
(2 rows)
```

```
\dPi
          List of partitioned indexes
 Schema |         Name          | Owner | Table
--------+-----------------------+-------+-------
 public | users_id_country_idx  | amit  | users
(1 row)
```

```
\dt
                  List of relations
 Schema |        Name        |       Type        | Owner
--------+--------------------+-------------------+-------
 public | sensor1            | table             | amit
 public | sensor2            | table             | amit
 public | sensor_next_id     | sequence          | amit
 public | sensors            | partitioned table | amit
 public | users              | partitioned table | amit
 public | users_a_to_i       | table             | amit
 public | users_j_to_r       | partitioned table | amit
 public | users_j_to_r_india | table             | amit
 public | users_j_to_r_japan | table             | amit
(9 rows)
```

Postgres 12

```
-- partitioning introspection functions: pg_partition_root, pg_partition_ancestors, pg_partition_tree

SELECT  relid::regclass AS table,
        parentrelid::regclass AS parent,
        CASE isleaf WHEN true THEN 'yes' ELSE 'no' END AS leaf,
        level,
        pg_partition_root(relid) AS root,
        ARRAY(SELECT pg_partition_ancestors(relid)) AS ancestors
FROM    pg_partition_tree('users')
ORDER BY      leaf;
       table        |    parent    | leaf | level | root  |                ancestors
--------------------+--------------+------+-------+-------+-------------------------------------------
 users              |              | no   |     0 | users | {users}
 users_j_to_r       | users        | no   |     1 | users | {users_j_to_r,users}
 users_a_to_i       | users        | yes  |     1 | users | {users_a_to_i,users}
 users_j_to_r_japan | users_j_to_r | yes  |     2 | users | {users_j_to_r_japan,users_j_to_r,users}
 users_j_to_r_india | users_j_to_r | yes  |     2 | users | {users_j_to_r_india,users_j_to_r,users}
(5 rows)
```

Postgres 12

- ATTACH PARTITION command no longer blocks queries, which makes adding new partitions a less disruptive operation than before, a huge operational plus
- Performance has been improved significantly by rewriting various pieces of code to process only the partitions that are needed by a query. So where previously, single-record queries would run in the amount of time that is proportional to the number of partitions, that is no longer the case.
- COPY on partitioned tables couldn’t use certain low-level optimizations like per-partition row-buffering, which has been fixed. That boosts COPY’s performance significantly when loading ordered data into range-partitioned tables, for example.
- Planner can now avoid doing explicit sorts for queries that need ordered data from certain partitioned tables, mainly range partitioned tables.

Last but not least

- Partitioning is not panacea
- There's a chance that you might "what have I done?!"
- Read up on best practices:

5.11.6. Declarative Partitioning Best Practices
https://www.postgresql.org/docs/current/ddl-partitioning.html#DDL-PARTITIONING-DECLARATIVE-BEST-PRACTICES

"The choice of how to partition a table should be made carefully as the performance of query planning and execution can be negatively affected by poor design."

Future enhancements

- It wouldn't be totally inappropriate to say that say that Postgres has decent partitioning at this point, but there's always more to be done 😊
- Many people have recently wished to see partition creation itself be automated
- Global indexes on partitioned tables
- Teach planner to consider partitioned indexes
- Optimizing for non-single-record queries
- Improve partitioning in scale-out clusters, from both usability and performance standpoints

Summary

In this talk, the following topics were covered:

- Partitioning concepts
- The “old” Postgres partitioning
- Declarative partitioning
- Timeline of declarative partitioning features
- Future enhancements

Thank you!

- For listening to this talk 😊
- To Postgres developers for writing code, reviewing code, reporting feedback/bugs related to partitioning 😊
- Questions?