Partitioning in Postgres
How Far We've Come

Amit Langote
NTT OSS Center
PGConf.ASIA 2019, Bali

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

About the speaker

- Live and work in Tokyo.
- Contribute to community Postgres, have written feature patches, bug fixes, review other people's code.
- Contributed to features like declarative partitioning, command progress reporting, among others. In recent releases, worked primarily on improving the performance and the scalability of partitioning.

Outline

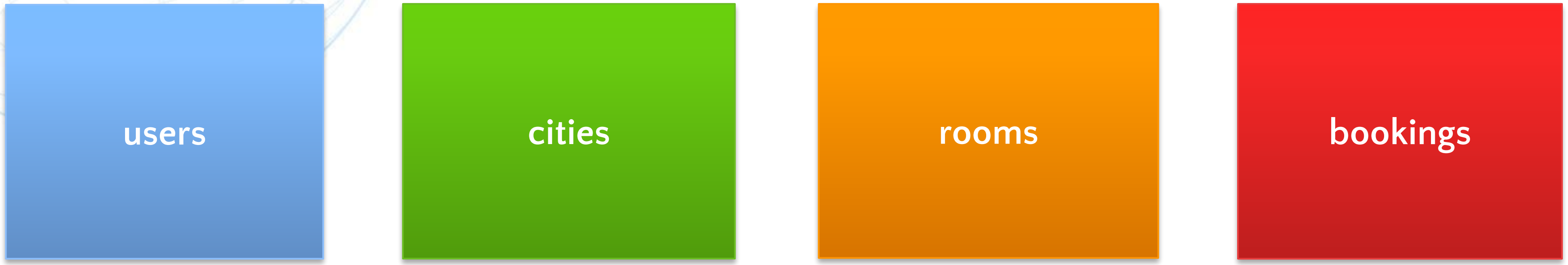
- Partitioning concepts
- Partitioning in Postgres: the “old” way
- The coming of declarative partitioning
- Postgres 10 and 11: foundations
- Postgres 12: performance

Outline

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

Data growth

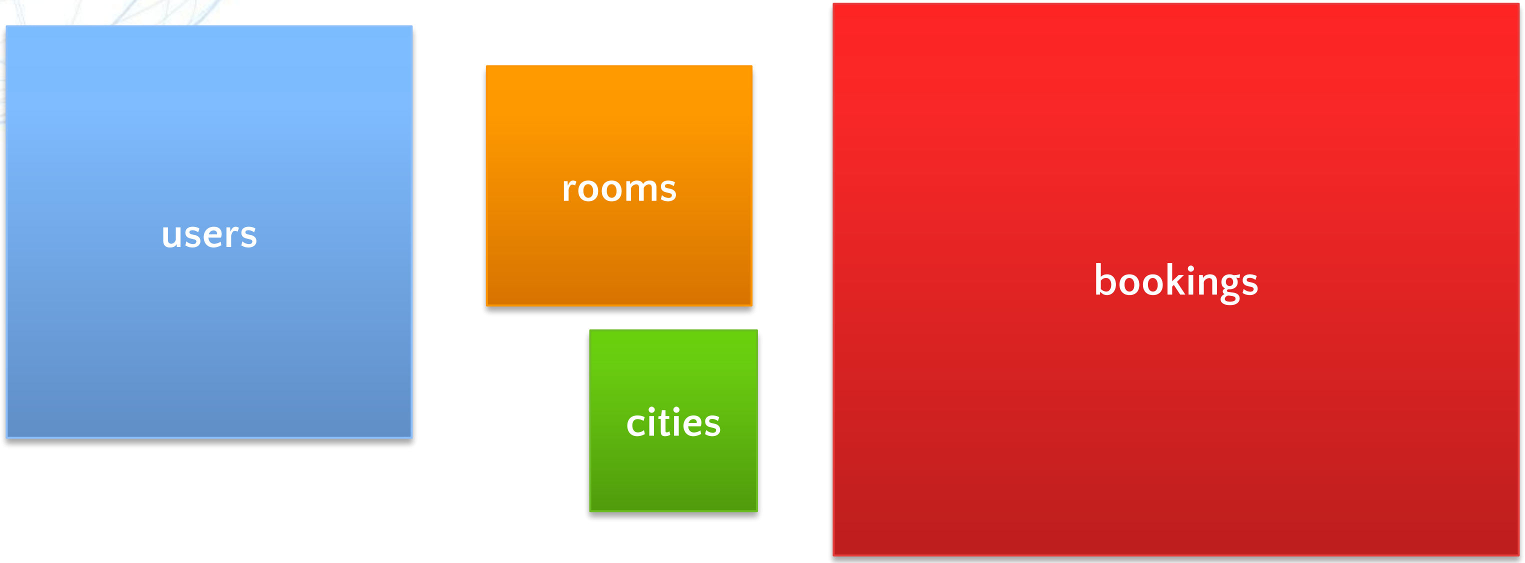
- Non-trivial applications accumulate and access data, usually with the help of a database system. As an example, consider an application that allows users to browse and rent rooms in different cities.



- Even with a rigorously vetted data model and normalization, data corresponding to certain modeled entities may grow pretty quickly, likely due to organic growth in application usage, which makes the operations on those data slower

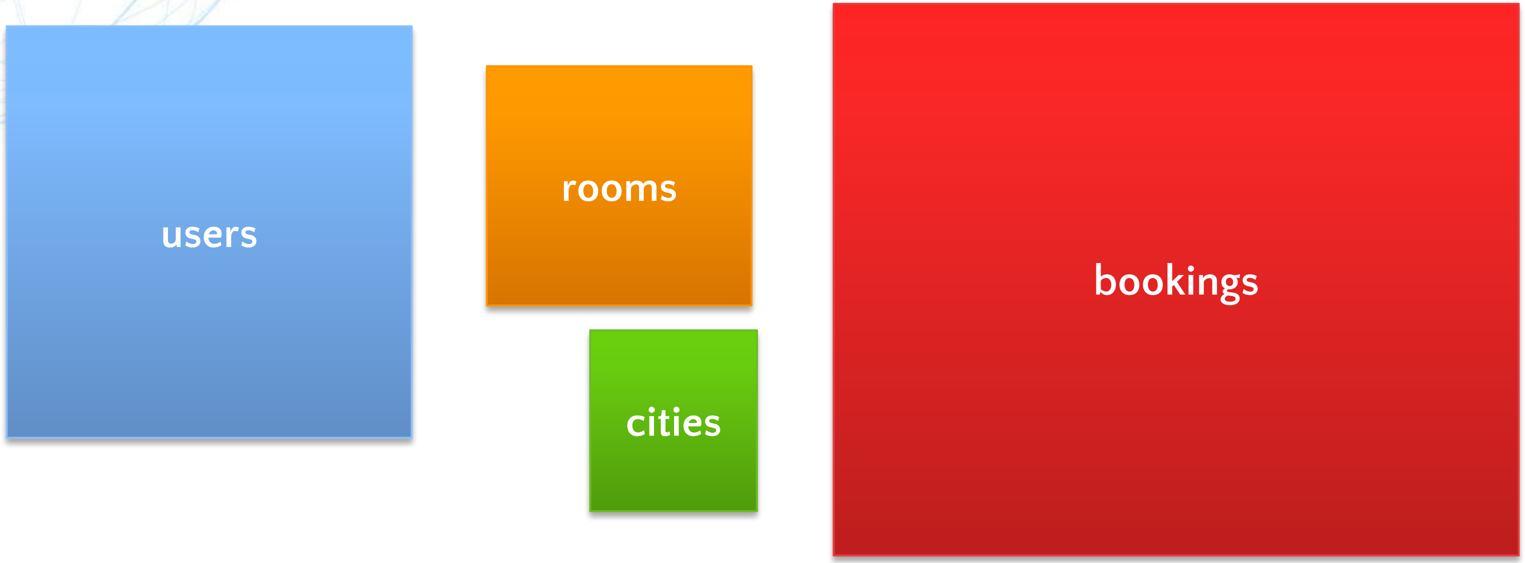
Data growth

- Assuming relational model, this means tables for certain pieces of data growing too big for the current configuration of the database. So for example, looking up a user or updating a booking will start becoming slower, because the database has to process ever growing amount of unrelated data.



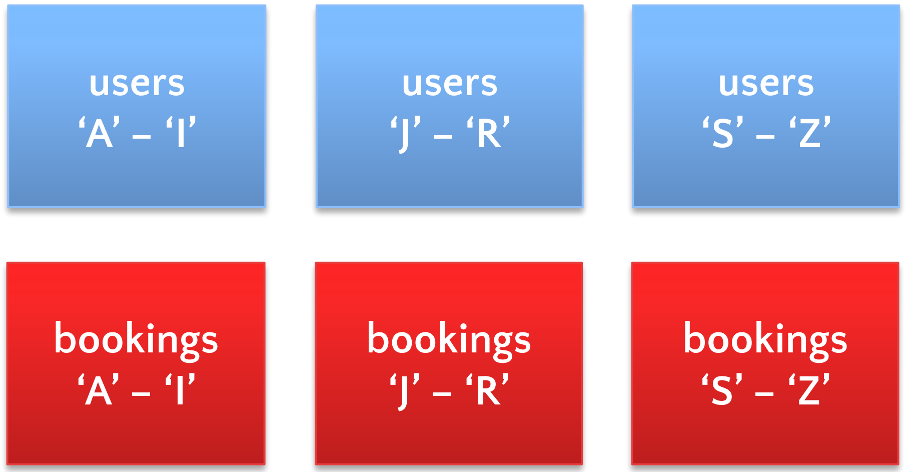
Handling data growth

- Up to a point, administrators can get away by increasing the configured resources and/or migrating to faster hardware or by tuning the database.
- Database software could itself be improved to handle bigger tables more efficiently with better algorithms and code-level optimizations.



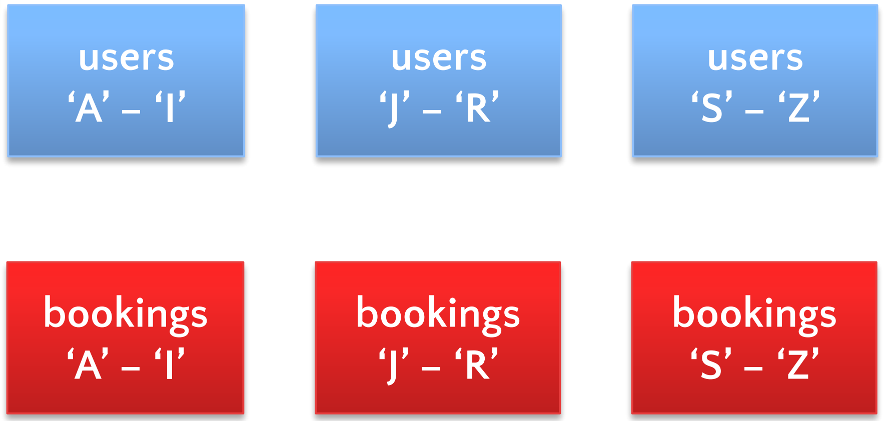
Handling data growth: Partitioning

- Another way of dealing with the resource-hungry data entities is to break them into smaller pieces
- What is logically one entity in the application is manipulated as multiple, individually-addressable objects in the database system. In our example, entities users and bookings could each live as multiple tables in the database, with each table storing an application-defined subset of the data.



Partitioning

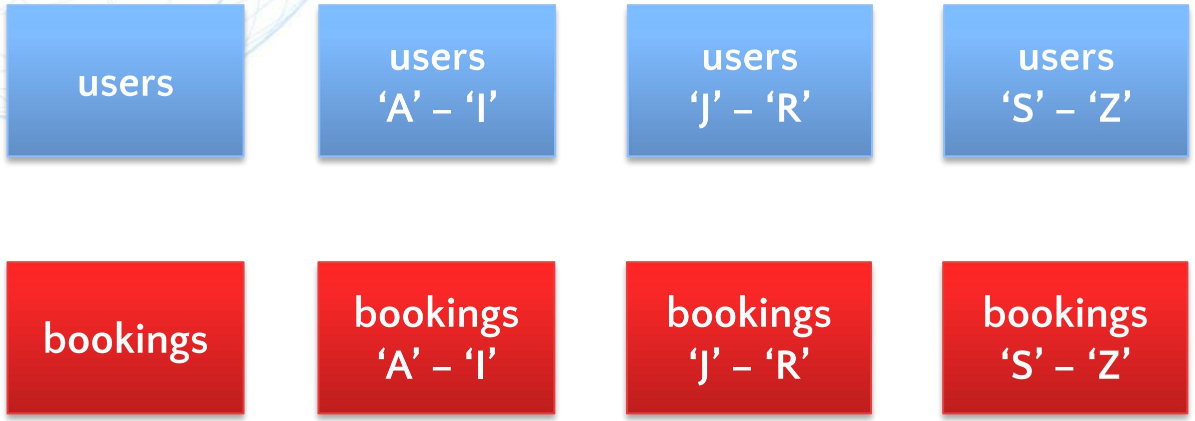
- Note that the database is still going to have to store the same amount of data as before, but the individual operations on those data can refer to only the partitions of interest. For example, to look up a user whose name starts with 'A', the application may issue the operation to only the relevant partition.



- In one approach, the partitioning is defined and implemented entirely in the application code. Database only has to deal with multiple smaller tables.

Partitioning

- In another approach, partitioning is local to the database, whereby the application references the logical entities which the database maps to their partitions. For example, an update to the booking of a user whose name starts with ‘A’ is converted by the database into an update of the relevant bookings partition.

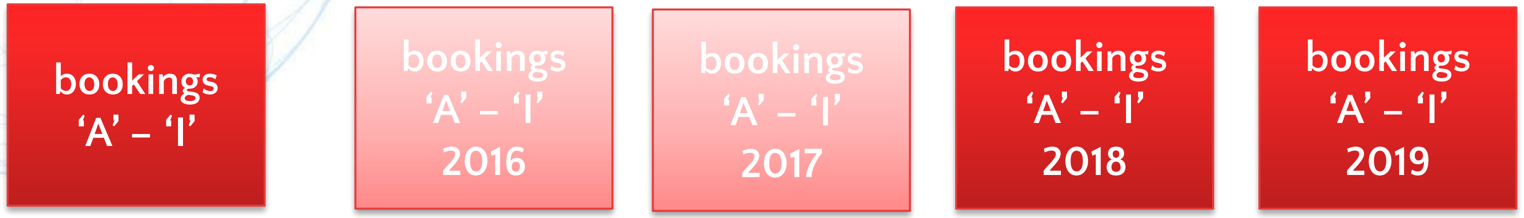


- In this approach, the partitioning is defined by the application code and mostly implemented in the database.

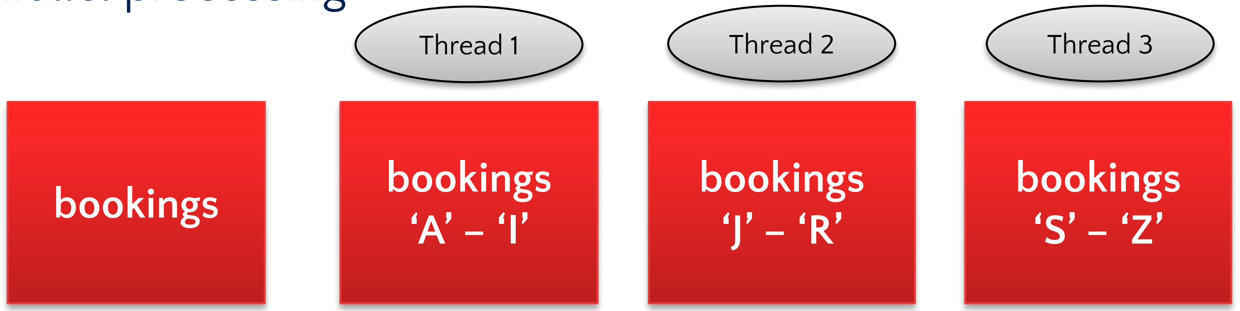
Partitioning

Other benefits of partitioning include, among others:

- Efficient archiving.

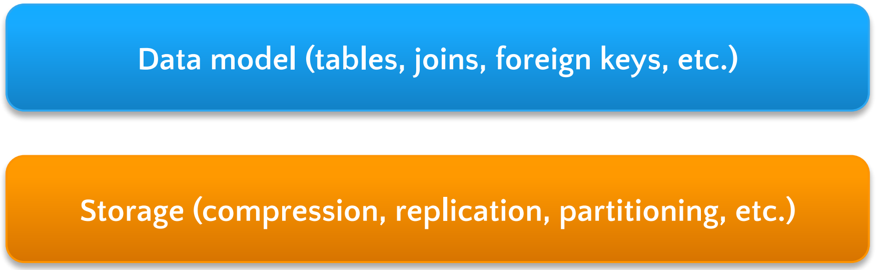


- Parallel processing



Intrinsic partitioning

- Some natively clustered 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.


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

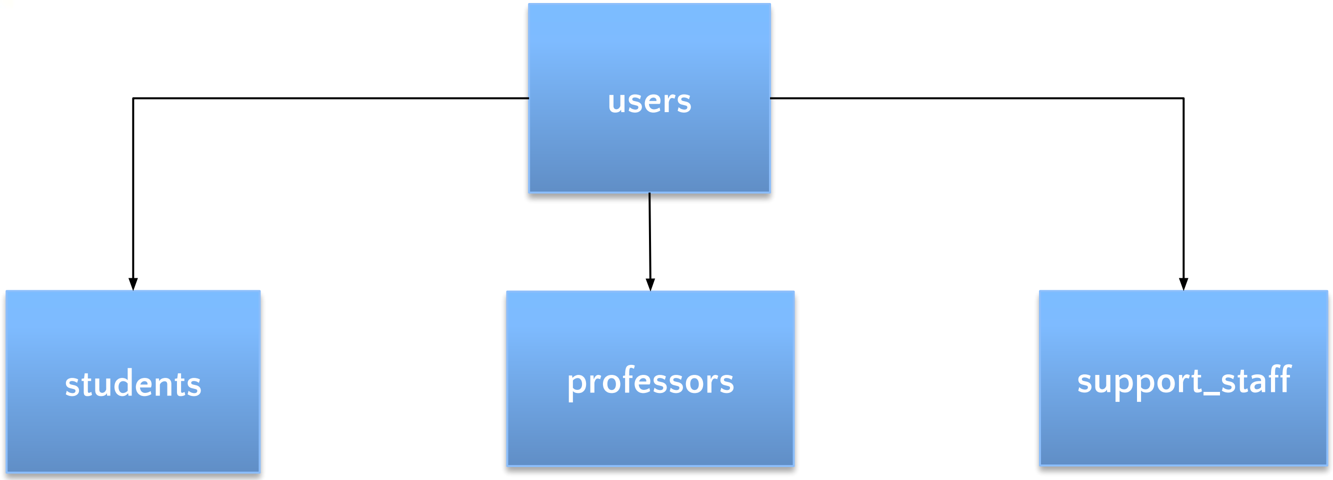
- The storage layer does try so that the partitioning is optimal for the application's workload, but it's mostly automatic, that is not in the application's control.
- The application directly interacts with only the data model layer

Outline

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

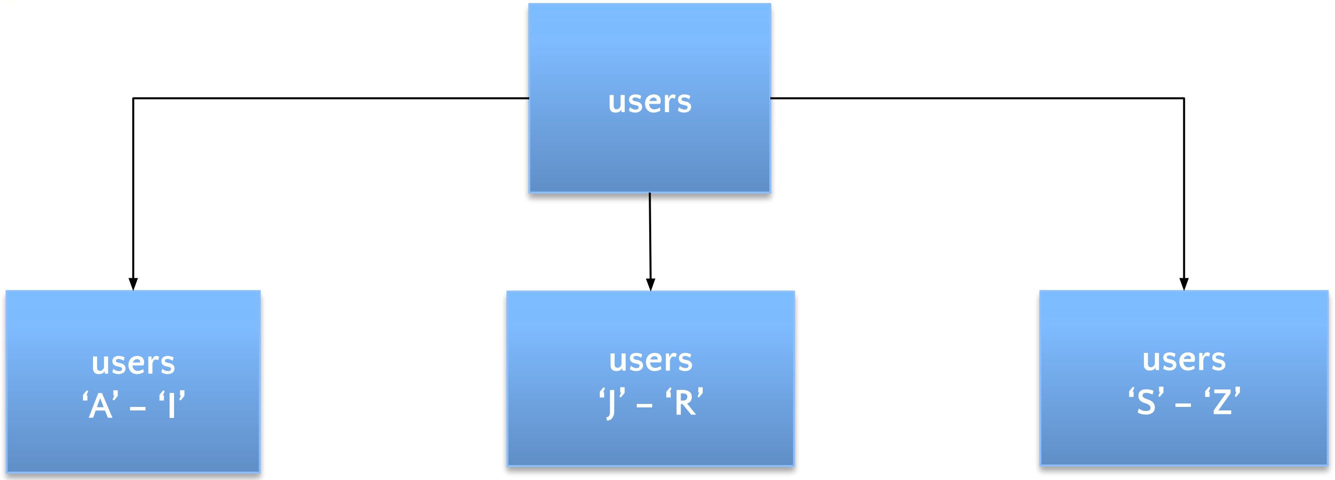
Partitioning in Postgres: the "old" way

- Postgres has supported in-database partitioning for a long time, even though the optimizations for it came around much later (14 years ago).
- Traditional implementation of partitioning borrows from table inheritance, which allows grouping related tables by inheriting from the same parent table/class.
- Operations on the generic parent table are internally applied to the more specific child tables, even though the application may operate directly on the child tables.



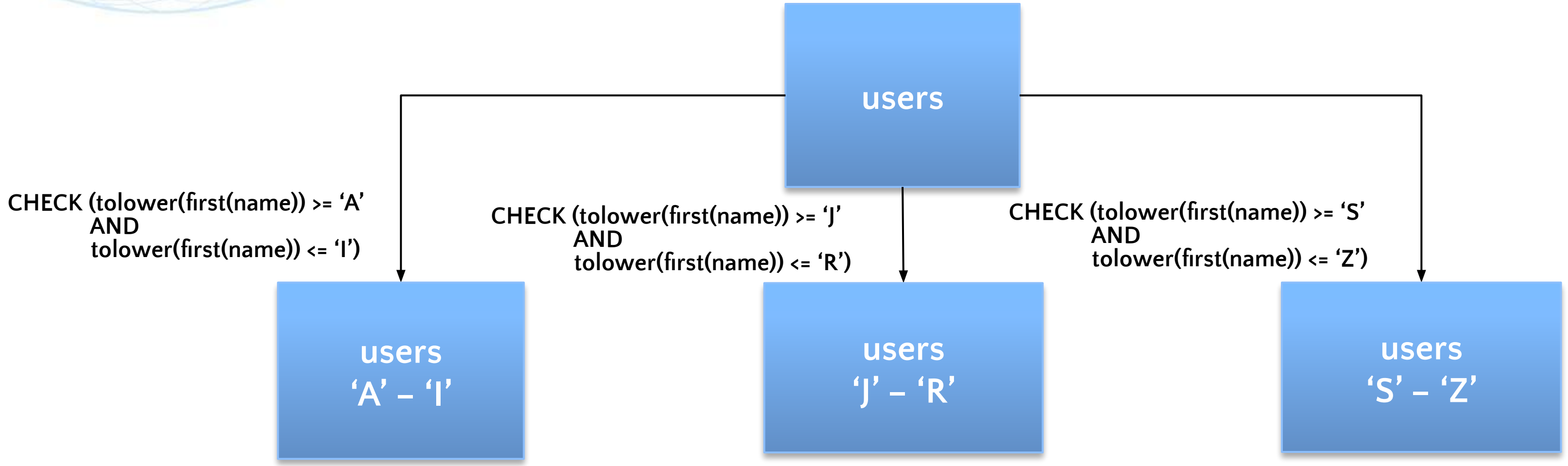
Partitioning in Postgres: the “old” way

- The ability to group related tables under one name is very useful for the requirements of partitioning, namely refer to multiple tables by one name.
- The child tables are partitions, which contain the actual data, whereas the parent table is just a placeholder for the application code.
- The application issues operations on the parent table which Postgres internally applies to the child tables.



Partitioning in Postgres: the "old" way

- Moreover, Postgres can avoid processing irrelevant child tables with some additional setup.
- To do so, application writer 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.

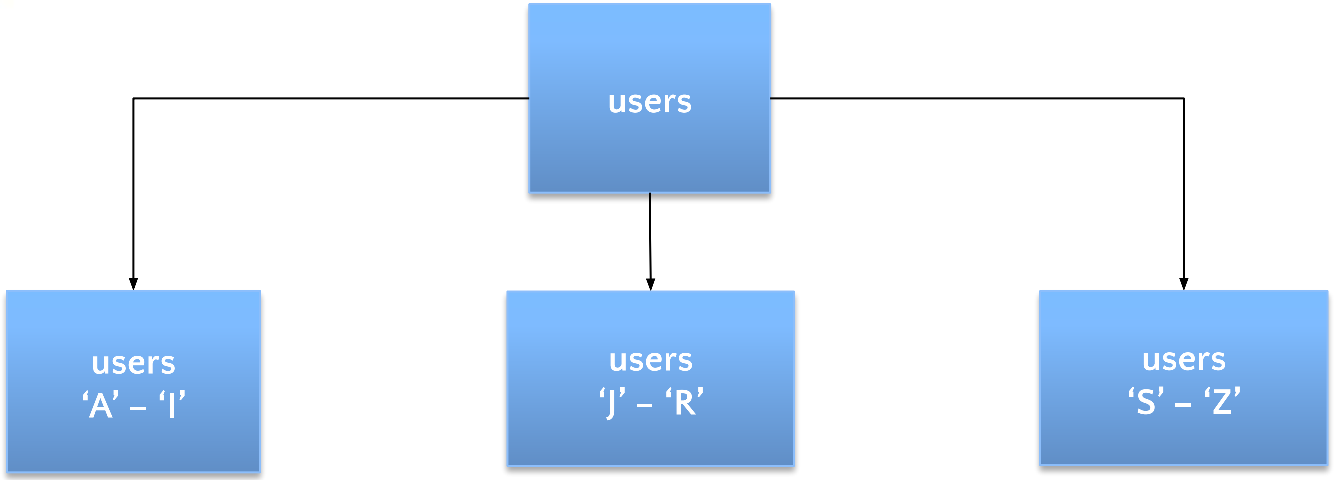

users
CHECK (tolower(first(name)) >= 'A' AND tolower(first(name)) <= 'I')
CHECK (tolower(first(name)) >= 'J' AND tolower(first(name)) <= 'R')
CHECK (tolower(first(name)) >= 'S' AND tolower(first(name)) <= 'Z')
users 'A' – 'I'
users 'J' – 'R'
users 'S' – 'Z'

Partitioning in Postgres: the “old” way

- While Postgres translates SELECT, UPDATE, and DELETE performed on the parent to child tables, it doesn’t translate INSERT. Table inheritance proper doesn’t need such a feature, because the child tables are typically directly manipulated in the application code.
- A workaround for that is to write the application code to insert into child tables directly or to use database triggers for INSERT redirection – 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 by inspecting the input data.

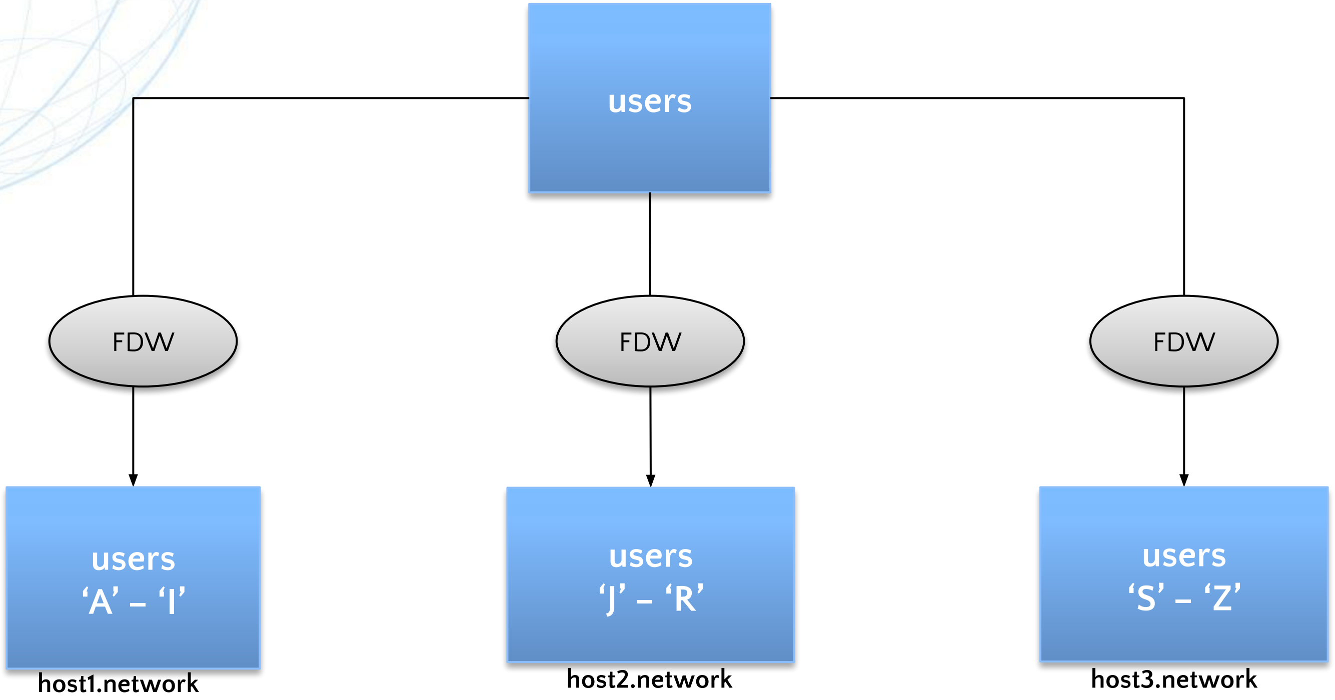
Partitioning in Postgres: the “old” way

```
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:

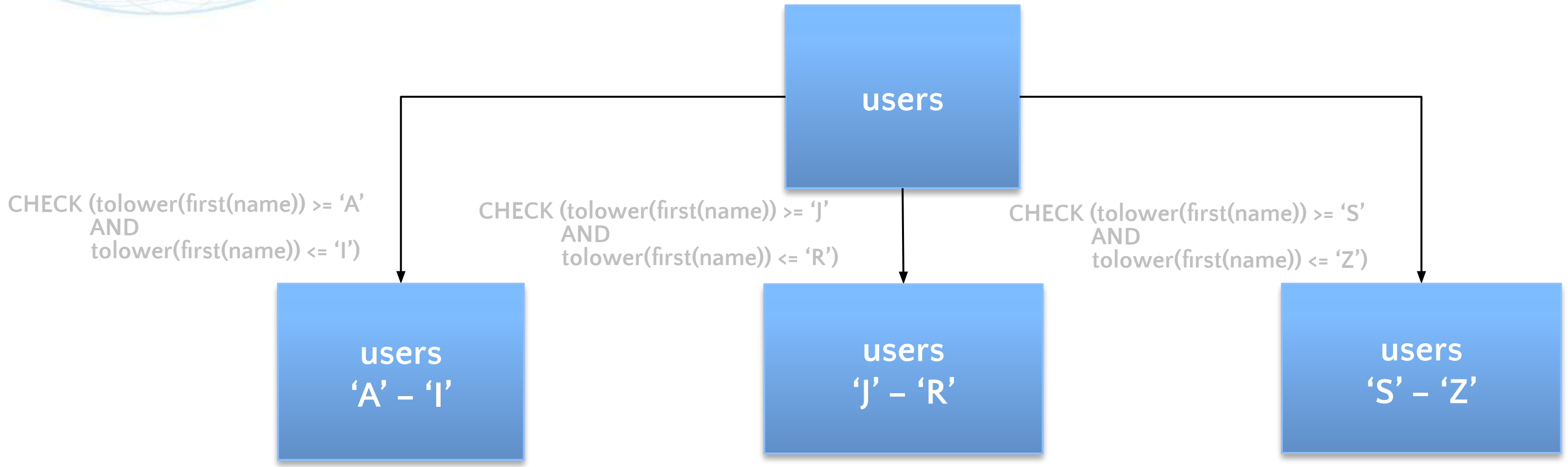
- Poor usability, because the application writer still has to bear most of the responsibility for making sure that partitioning is set up correctly
- Poor performance, especially as the number of partitions increases, because the architecture of Postgres for processing queries is not really optimized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

Outline

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

Declarative partitioning

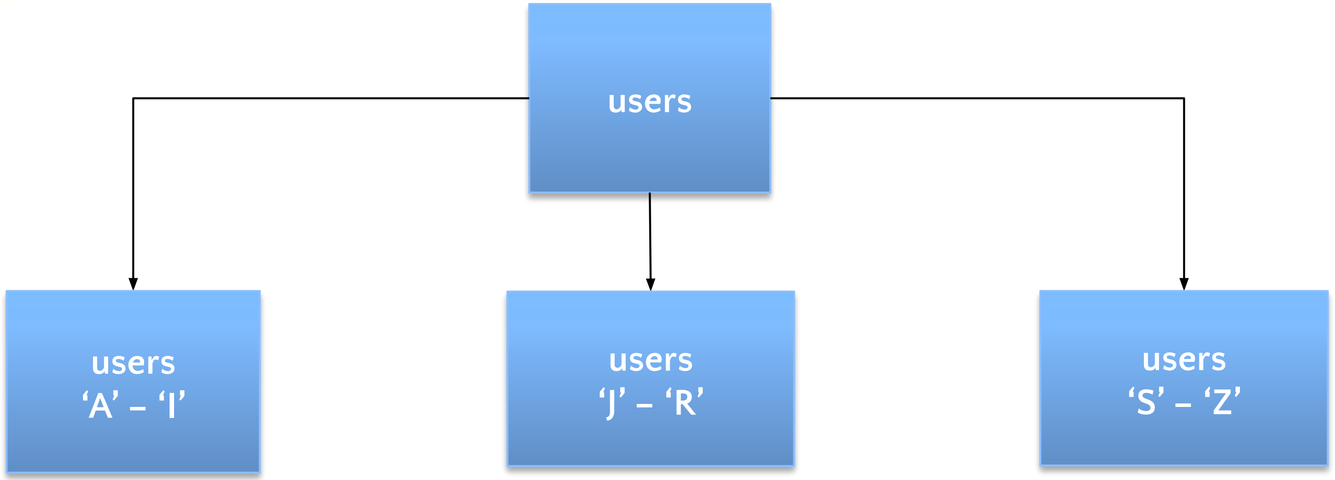
- It's the same old multiple-tables-under-one-name architecture, but with a new syntax to define partitions.
- Users still bear the responsibility for deciding what partitions to create and also for creating them using the new commands (CREATE TABLE + partition clause), but Postgres takes care of the rest


users
CHECK (tolower(first(name)) >= 'A' AND tolower(first(name)) <= 'I')
CHECK (tolower(first(name)) >= 'J' AND tolower(first(name)) <= 'R')
CHECK (tolower(first(name)) >= 'S' AND tolower(first(name)) <= 'Z')
users 'A' – 'I'
users 'J' – 'R'
users 'S' – 'Z'

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 new features around declarative partitions have more to do with easier administration of partitions than improving the basic architecture of partitioning.
- The bright side is that Postgres is able to capture some partitioning-specific information about tables thanks to the new syntax, which is stored in partitioning-specific system catalog tables. That information allows Postgres to better optimize queries over declarative partitions compared to “old”-style partitions.

Declarative partitioning

- The number of partitions that can be reasonably handled is also bigger now, though not because we completely solved the earlier mentioned architectural problems of Postgres that prevents it from handling many tables well, but because the newly enabled optimizations allow those problems to be papered over, for now at least.
- There is some effort focused on solving the fundamental architectural problems, now more than ever.

Outline

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

Postgres 10

- Postgres 10 introduced declarative partitioning, with the basics:
  - The syntax for RANGE and LIST partitioning
  - Automatic partition constraint enforcement
  - INSERT and COPY (except for foreign table partitions)
- Things that can be done aren't new compared to "old"-style partitioning, but the setup consists of much less chores, as described.
- Bulk INSERT operation (including COPY) is at least an order of magnitude faster than with a plpgsql trigger.
- Not all rainbows... With the introduction of a new table type ("partitioned tables"), supporting the definition of various database objects on those tables would require some careful consideration, so it wasn't implemented in Postgres 10. Users get an unsupported feature errors instead.

Postgres 10

- Examples of such unintuitive limitations are the inability to define indexes, index-based constraints, foreign key constraints, row-level triggers, etc. on partitioned tables. Users are not prevented from defining these objects directly on partitions, but that's extra work.
- 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
- People would write triggers and database-external scripts to handle those issues as they'd see fit, but the expectation is that at least some of those things are handled inside the database.

Postgres 11

- A pretty significant release for partitioning
- Most of the restrictions on partitioned table DDL are lifted:
  - Indexes can be defined
  - 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
- For each of the above, like how CHECK and NOT NULL constraints work, the object defined on a partitioned table is also defined on all of the existing partitions, where it is actually enforced.   The partitioned table's copy is simply a template for the future partitions.

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
- The following new techniques are now applied when processing queries involving partitions.
  - Partition pruning (superseding constraint exclusion)
  - Application of partition pruning during execution (in addition to during planning)
  - Partitionwise join (when joining co-partitioned tables, join individual partitions)
  - Partitionwise aggregate (perform aggregates on individual partitions)

Postgres 11

- Overall, 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 puts the upper limit on the number of partitions that can be used somewhere around low 100s, because any given query would need to touch them all.
- This results in an unacceptable performance profile, especially for workloads that contain single-record queries on partitioned tables, which are ideally handled by touching only the partition that contains the record.

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 in processing common queries, 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

- 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 touched 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 couldn’t use certain low-level optimizations like per-partition row-buffering for partitioned tables, which has been fixed. That boosts COPY’s performance significantly for the common set of loading patterns.
- Planner can now avoid doing explicit sorts for queries that need ordered data from certain partitioned tables, mainly range partitioned tables.

Postgres 12

- It wouldn’t be totally inappropriate to say that say that Postgres has decent support for partitioning at this point. 😊
- Many people have recently wished to see partition creation itself be automated, which seems like an area that the next set of usability improvements might come from.
- One more avenue for future improvements is to offer better support for partitioning in scale-out clusters, from both usability and performance standpoints.

Future enhancements

- Auto-creation of partitions
- Global indexes on partitioned tables
- Teach planner to consider partitioned indexes
- Optimizing for non-single-record queries

Summary

In this talk, the following topics were covered:

- Partitioning concepts
- The “old” Postgres partitioning
- Declarative partitioning
- Progress of development of declarative partitioning
- Future enhancements

Thank you!

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