AWS PostgreSQL RDS

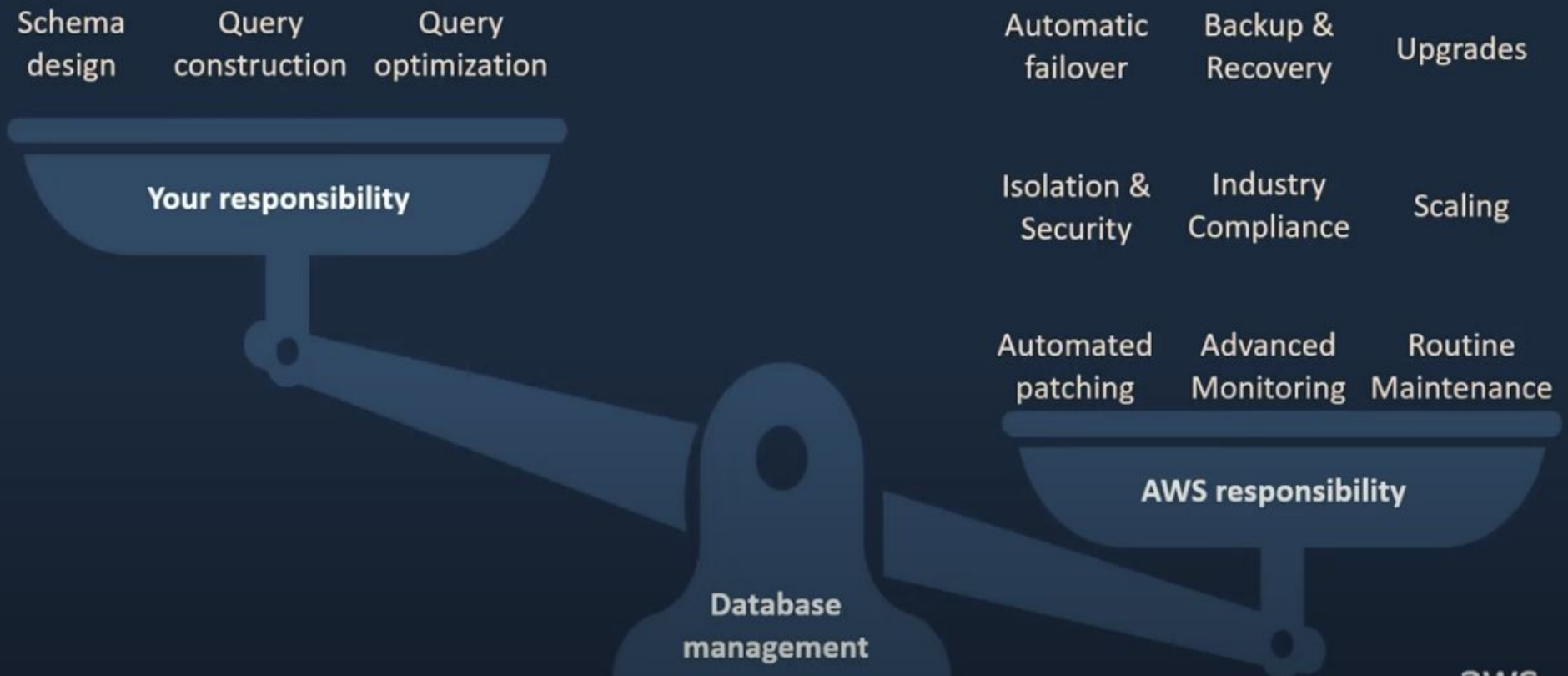
Partition

PostgreSQL – Open Source Database

- PostgreSQL is a powerful, open source object relational database system
- Origins of PostgreSQL date back to 1986 as part of the POSTGRES project at the University of California at Berkeley
- More than 30 years of active development on the core platform
- Rich features and extensions, reliability and standards compliance, open source license

Accelerate path to innovation with managed databases

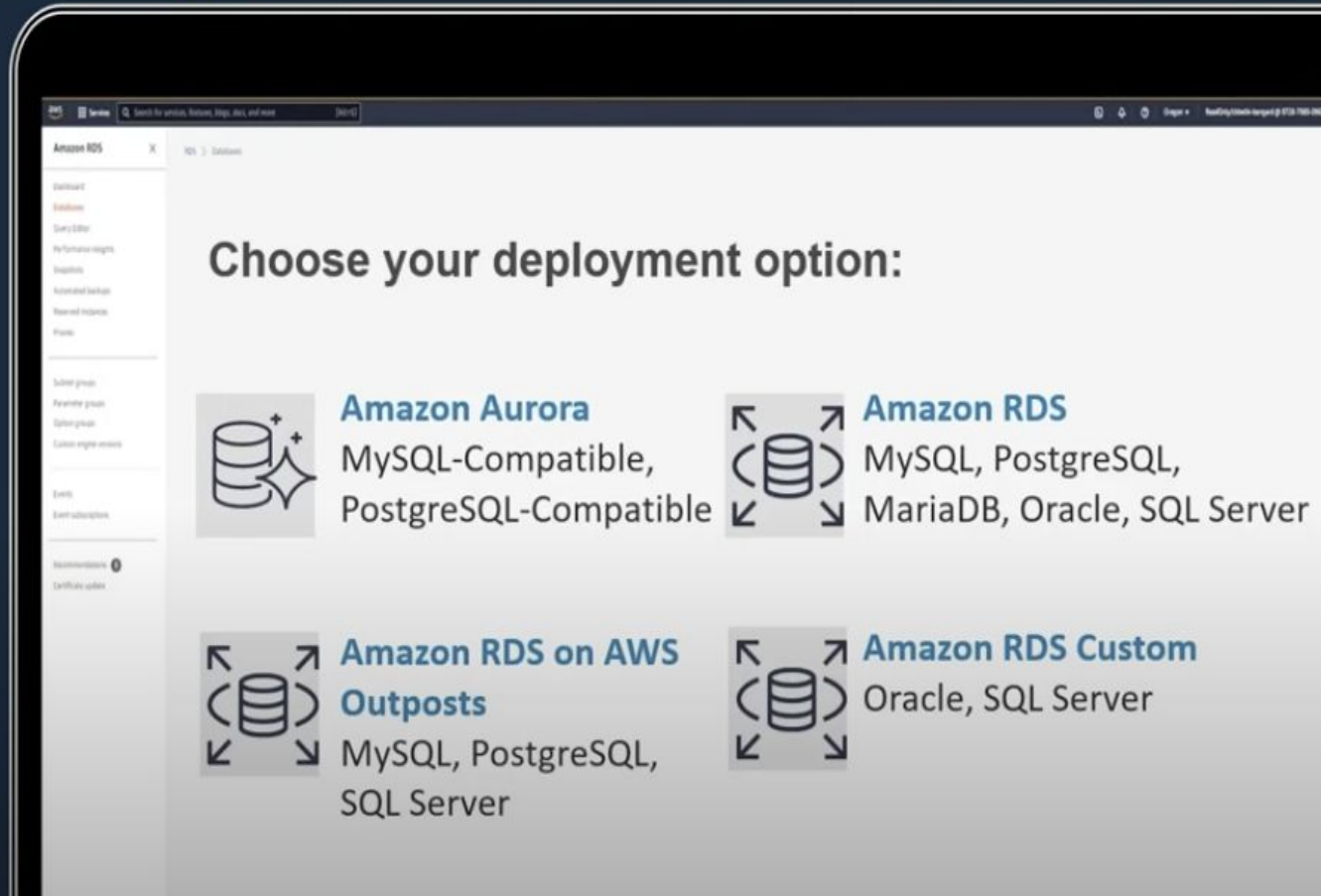
Spend time innovating and building apps, not managing infrastructure



Deployment options with Amazon RDS

Get started with your choice of 7 popular relational databases across 4 deployment options

Unlock better performance, availability, scalability, and security



Partition

Challenges of hyper-scale customers

New growth pattern: sudden, unanticipated, and sustained

Challenges

- Rapid growth of data size creating large size table(s)
- Increase on-disk footprint approach infrastructure limits
- Increase query response time
- Operational challenges for vacuum, index management, upgrades, and etc.
- Increase recovery time size of large table

The solution – Table partitioning

Divide a large table into multiple small segments, called partitions

Benefits

- **Availability**: partition level backup / restore
- **Manageability**:
  - Partition-based maintenance (index, vacuum)
  - Easy add / remove partitions via partitioning "sliding window" for data lifecycle management
- **Performance**:
  - Allow query execution to target a partition / subset of partitions (partition pruning)
  - Partition wise join

Table partitioning in PostgreSQL

<= PG v9.6

- Table Inheritance
- CHECK Constraints
- Trigger for routing

PG v10

- Declarative Table Partitioning

PG v11

- Hash Partitioning
- Indexes on partitioned tables
- FOR EACH ROW trigger
- Partition wise join
- Partition wise aggregates

PG v12

- Partition Pruning enhancements
- Foreign keys to reference partitioned tables
- Faster Copy for partitioned tables
- Non-blocking ATTACH Partition

PG v13

- Partition wise join improvements
- Partition Pruning improvements
- Before row-level triggers
- Support logical replication

PG v14

- Detach partition concurrently
- Improve update / delete performance

Declarative table partitioning

- A partitioned table is a virtual table without any physical storage
- Physical data is stored in partition leaf tables
- Data is split into partitions based on the partition strategy defined
  - Partitioning method
  - Partitioning key
- The subset of data in a partition is defined by Partition Bounds
- A partition may contain sub-partitions
- A default partition stores data that does not belong to any partitions

Partitioning strategy – Partitioning method

Decision point: **partitioning method**

Partitioning methods

- **Range**: Data is placed in partitions based on a range of values
    - Lower end: inclusive, upper end: exclusive
- **List**: Data is placed in partitions based on a list of discrete values
- **Hash (PG v11+)**: Data is placed in partitions based on a hash algorithm applied to a key
- **Composite**: Combining one of more of partitioning types, e.g. list + range, hash + list, and etc…

Partitioning strategy – Partition key

Decision point: **partition key**

Know your data & workload access patterns

- The key must be made up of columns and/or immutable expression
- The key value must be a literal
- The partition key can include up to 32 columns or expressions
    - List partitioning is limited to a single column

Time-series data

- Data is treated as immutable append-only log, table grows continuously overtime
- Old rows may be deleted if purging is needed, but they are not changed
- Typical access to data is specific for a time range
- User cases: Internet of things device streaming data, transaction history table in an OLTP database

Relational database with partitioning vs. purpose-built database

Amazon RDS for PostgreSQL

- Need to perform join with other relational data
- Data can be injected in batches

Purpose-built Amazon Timestream

- Time-series data to be used in standalone application
- Data is loaded directly from many locations or sensors

Example

Scenario: **online delivery app experiencing sudden, rapid growth of demands, 10 TB transaction history table growing at 2 TB a week.**

- Need to control size of the table
- Vacuum process is not able to keep up w/ the change rate
- Increasing query response time
- Customers access to orders in the last 90 days only, orders older than 90 days can be archived

```
CREATE TABLE transaction_history (
    time                timestamptz,
    transaction_id      bigint,
    total               money,
    status              text,
    transaction_details jsonb
);
```

The solution – Range partitioning

Decision on partitioning strategy

- Partition method: Range
- Partition key: time
- Partition boundary: day (transaction history older 90 days will be archived)

```
CREATE TABLE transaction_history (
    time                timestamptz,
    transaction_id      bigint,
    total               money,
    status              text,
    transaction_details jsonb
) PARTITION BY RANGE(time);
```

Manage partitions (native PostgreSQL)

Native PostgreSQL does not provide ways to manage partition automatically

- Create partitions manually
- Use cron scripts

```
CREATE TABLE transaction_history_p2022_01_01
  PARTITION OF transaction_history
    FOR VALUES FROM ('2022-01-01 00:00:000’) TO ('2022-01-02 00:00:000');

CREATE TABLE transaction_history_p2022_01_01
  PARTITION OF transaction_history
    FOR VALUES FROM ('2022-01-02 00:00:000’) TO ('2022-01-03 00:00:000');

# repeat to create partitioning for 90 days …
```

Automate partition management

pg_partman

- An extension to create and manage both time-based and serial-based table partition sets
- RDS for PostgreSQL support starts w/ PG v12.5

pg_cron

- An extension that allows you to use cron syntax to schedule PostgreSQL commands directly within your database
- RDS for PostgreSQL support starts w/ PG v12.5

pg_partman

Automate operations for

- Create partition parent table
- Continuous creation of partition child tables, attach to partitioned table
- Attach new partitions
- Data lifecycle management (detach/drop aged partitions)

Enable pg_partman separately for each database:

```
CREATE SCHEMA partman;

CREATE EXTENSION pg_partman WITH SCHEMA partman;
```

Automate partition creation

Policy-based automatic partition management

- Create_parent() function call
- Define policy for creating new partitions continuously

```
SELECT partman.create_parent(
    'public.transaction_history',
    'time',
    'native',
    'daily',
    p_start_partition := (now() - interval '90 days')::date::text );
```

Ongoing partition maintenance

Manage partitions in rolling window: create new / remove old partitions

- Set retention policy with part_config
- Define actions on “aged” partitions
- Call run_maintenance_proc() to trigger maintenance

```
UPDATE partman.part_config
  SET retention = '90 days',
  retention_keep_table=true
  WHERE parent_table = 'public.transaction_history';
```

Automate maintenance with pg_cron

- To enable pg_cron on RDS for PostgreSQL
  - Add pg_cron to shared_preload_libraries parameter in the DB instance's parameter group
  - Create extension
- Set scheduled maintenance with calling run_maintenance_proc()

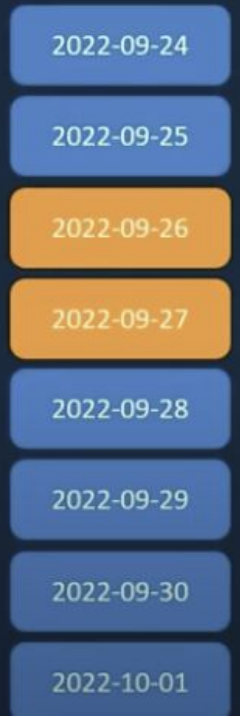
```
CREATE EXTENSION pg_cron;
SELECT cron.schedule(
  'transaction_history maintenance',
  '0 22 * * *',
  $$CALL partman.run_maintenance_proc()$$
);
```

Results – Reduce query response time

Results of query pruning – a query optimization technique that improves performance for declaratively partitioned tables

If the query is constructed properly,
PostgreSQL can eliminate whole partitions
from being scanned

```
SELECT sum(total)
FROM transaction_history
WHERE time
BETWEEN '2022-09-26'
AND '2022-09-27';
```



Results – Faster bulk loads and deletes

Bulk loads

- Create stand-alone table without indexes or constraints
- Perform bulk load
- Create indexes or constraints
- ALTER TABLE ATTACH PARTITION … to add to partitioned table
- Benefits: reduce locking and impacts to parent table during bulk loads

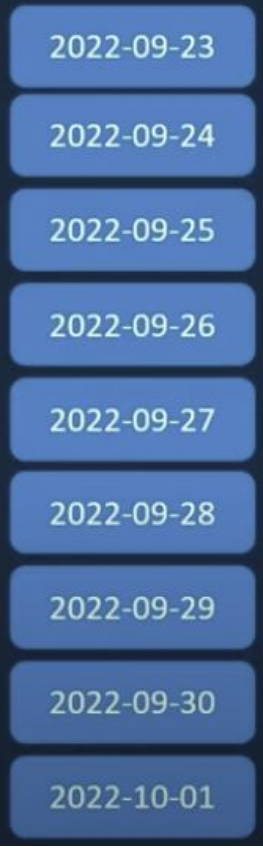
Bulk deletes

- ALTER TABLE DETACH PARTITION, then Drop TABLE
- Benefits: avoid the VACUUM overhead caused by a bulk DELETE

Results – Data lifecycle management

Migrate less frequent used data to cheaper storage

- Rolling windows of data allows easy archival of cold data
- PostgreSQL v14+ supports DETACH PARTITION CONCURRENTLY
- Does not create bloat with deleting expired rows



How effective is the solution?

- ✓ Manage size of the table
  - Data divided into child tables with less than 3 GB per partition
  - Partitioned table stores most recent 90 days of data
- ✓ Reduce query response time, enhance overall performance
  - Allow query execution to target a subset of partitions with query pruning
  - Partition wised join
  - Partition-based index maintenance, vacuum, backup / restore
- ✓ Enable easy archival of historical data
  - Automatic archival of data older than 90 days
  - Concurrent detach of partitions with aged data without creating bloat

When to use partitioning?

For very large tables (e.g. > 100 million rows)

- Size of table unable to fit in memory
- For time-based of series-based data
  - Continuously streaming of data in append only fashion
  - Historical data can be archived or discarded
- Data access pattern
  - Querying data by time range, or discrete values
  - Data injection in batches

What partitioning is not?

Not a magic bullet

- Not for every table, partition has performance and operational overhead
- No substitute for good schema design or operation best practices
- Not substitute for query tuning
- Not sharding

Best practices – Partition key

know your data & workload access patterns

- Column(s) which most commonly appear in WHERE clauses for effective query pruning
- Column(s) to support rolling windows of data for archiving
- Column(s) that do not change often, to avoid moving rows among partitions

Best practices – Number of partitions

- Too few: individual partitions is too big
- Too many: increase query planning time
- Use sub-partitioning to divide data into multi-levels for very large data sets

Best practices

- Start partitioning early before size of table become unmanageable
- Get on the latest major and minor version releases for partitioning enhancements
- Use pg_partman and pg_cron extension to automate partition management

Limitations

- Primary keys must contain the partition key
- Unique Indexes: cannot create unique index across partitions.
- When an UPDATE causes a row to move from one partition to another, there is a chance that another concurrent UPDATE or DELETE will get a serialization failure error

Summary

- Partitioning benefits
  - Improved query performance (query pruning, partition-wise join)
  - Faster bulk load and bulk delete operations
  - Data lifecycle management
- Knowing your data access pattern helps to create effective partitioning strategy
- Implementing partitioning is transparent to clients
- Partitioning adds additional operational overhead and maintenance burden, evaluate if partitioning is the right solution for you