

Amazon Redshift

Best Practices

Scott Merrill

Solutions Architect, AWS



Discussion Topics

- Data Models
- Table Design Best Practices
- Query Best Practices
- Data Lake Modelling Best Practices
- Workload Management Best Practices
- Data Loads Best Practices
- Multitenancy Architecture Best Practices

Data Models



Redshift: Use Popular Data Models



Redshift can be used with a number of data models including...

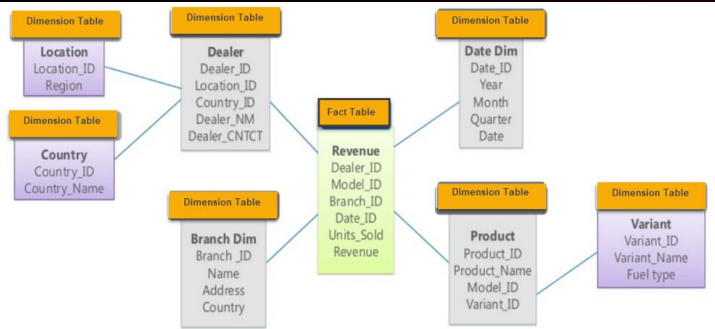
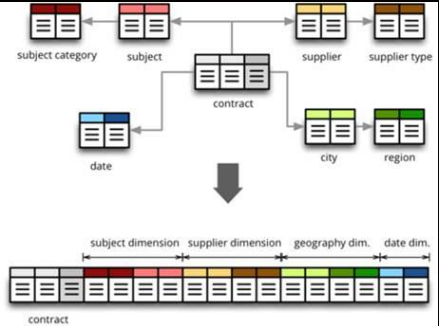
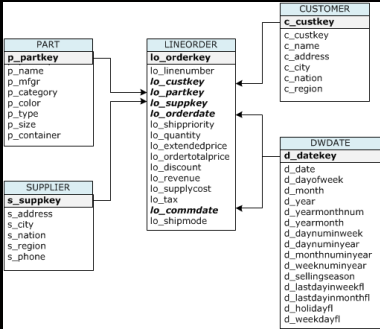
STAR Schema

Most Common

Highly Denormalized

Snowflake Schema

Less Common



Best Practice: Avoid highly normalized models. Models such as 3NF resemble the STAR schema, but has much more table normalization and are typically more appropriate with OLTP systems

A commonly used data model with Amazon Redshift is the STAR schema, which separates data into large fact and dimension (dim) tables:

- Facts refer to specific events (e.g. order submitted.) and **fact tables** hold summary detail for those events. e.g. the high-level attributes of an order submitted such as *order_id*, *order_dt*, *product_id*, & *total_cost* Fact tables use foreign keys to link to dim tables
- The dimensions that make up a *fact* often have attributes themselves that are more efficiently stored in separate **dim tables**. e.g. a fact might contain a *product_id*, but the actual product details would be contained in a separate *products* dim table (e.g. *product_price*, *height_cm*, *width_cm*, & *product_id* are columns that might be found in a *products* dim table)



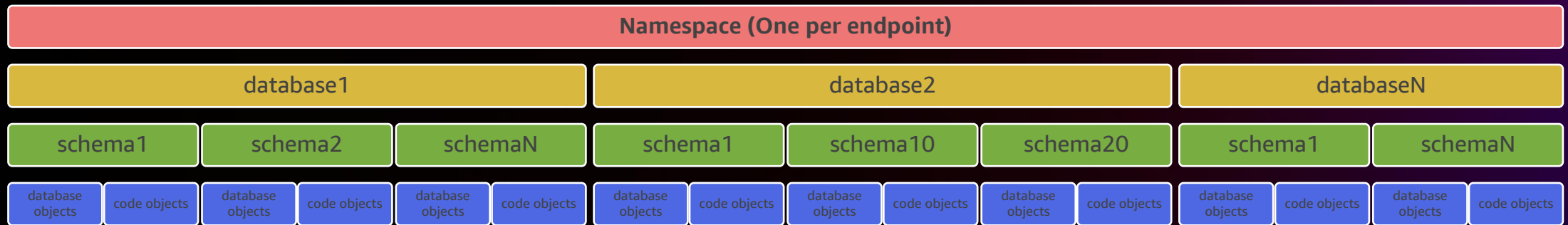
Amazon Redshift

Data Storage & Data Types

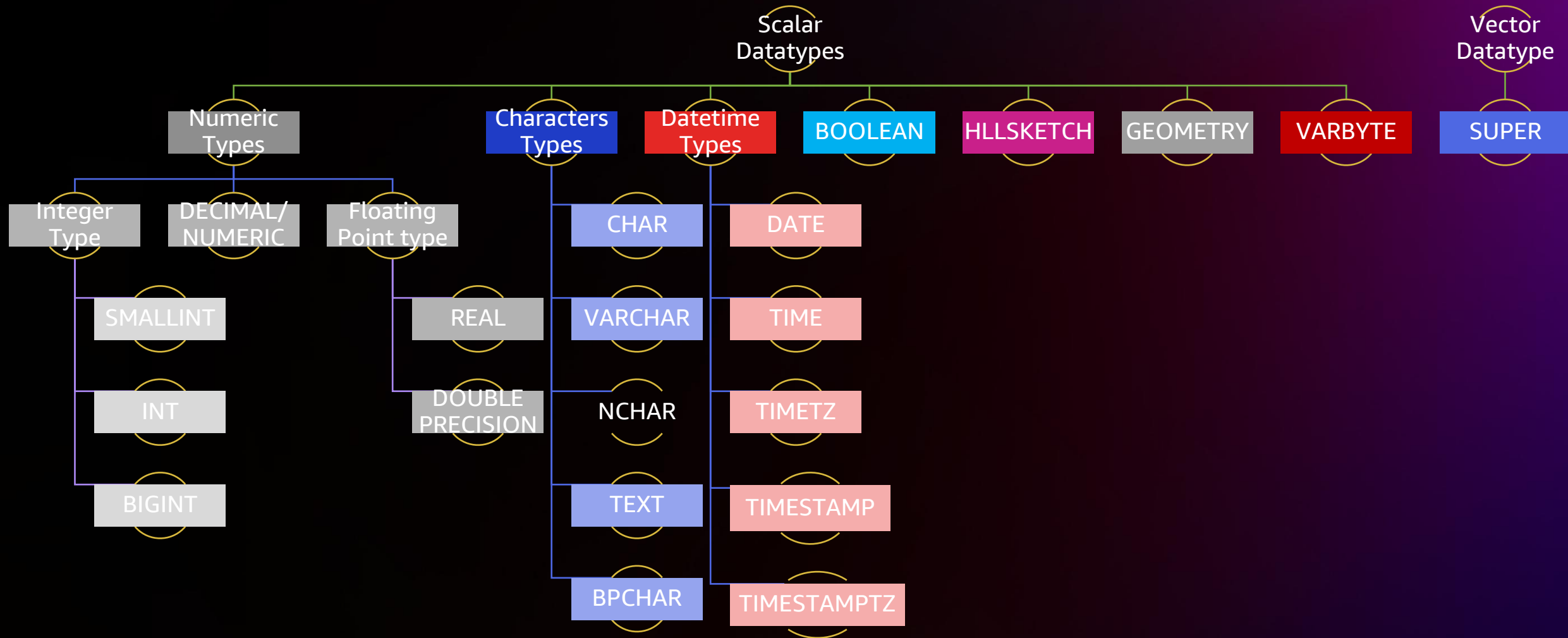


Data storage in Redshift

- Data loaded into Redshift is stored in Redshift Managed Storage (RMS), storage is columnar
- Structured and semi-structured data can be loaded
- Amazon Redshift is ANSI SQL and ACID compliant
- Does not require indexes or db hints. Leverages sort keys, distribution keys, compression instead, to achieve fast performance through parallelism and efficient data storage
- Data is organized as: Namespace > database > schema > objects



Redshift Datatypes



Choosing Data Types: Best Practices

- Make columns only as wide as they need to be. Redshift performance is about efficient I/O. Do not arbitrarily assign maximum length/precision. This can slow down query execution time.
- Use appropriate data types. Eg: Don't store date as varchar
- Multibyte Characters - Use VARCHAR data type for UTF-8 multibyte characters support (up to a maximum of four bytes)
- Use GEOMETRY data type and spatial functions to store, process and analyze Spatial data
- Use SUPER datatype to store semi-structured data and for evolving schema & schema-less data

```
create table sales(  
  listid varchar(255) not null,  
  sellerid varchar(255) not null,  
  buyerid varchar(255) not null,  
  eventid varchar(255) not null,  
  dateid varchar(255) not null,  
  qty sold varchar(255) not null,  
  pricepaid varchar(255) encode delta32k,  
  commission varchar(255) encode delta32k,  
  saletime varchar(255),  
  ...
```

```
create table sales(  
  listid integer not null,  
  sellerid integer not null,  
  buyerid integer not null,  
  eventid integer not null,  
  dateid smallint not null,  
  qty sold smallint not null,  
  pricepaid decimal(8,2) encode delta32k,  
  commission decimal(8,2) encode delta32k,  
  saletime timestamp,  
  ...
```



Additional Documentation

- [Querying Spatial Data in Redshift](#)



Semi-structured data – SUPER datatype

Data type: **SUPER**

Easy, efficient, and powerful JSON processing

Fast row-oriented data ingestion

Fast column-oriented analytics with materialized views over SUPER/JSON

Access to schema-less nested data with easy-to-use SQL extensions powered by the PartiQL query language

Supports up to 16 MB of data for an individual SUPER field or object

id	name	phones
INTEGER	SUPER	SUPER
1	{"given": "Jane", "family": "Doe"}	[{"type": "work", "num": "925550100"}, {"type": "cell", "num": "6505550101"}]
2	{"given": "Richard", "family": "Roe", "middle": "John", "}	[{"type": "work", "num": "5105550102"}]

```
SELECT name.given AS firstname, name.middle as
middlename, ph.num
FROM customers c, c.phones ph
WHERE ph.type = 'work';
```

firstname	middle	num
-----+-----		
"Jane"	null	925550100
"Richard"	"John"	5105550102

SUPER datatype: Best Practices

For low latency inserts or for small batch inserts, insert into SUPER.
Inserts into SUPER datatype are quicker.

If you join frequently using attributes stored in SUPER, create separate scalar datatype columns for those attributes to improve performance

If you filter frequently using attributes stored in SUPER, create separate scalar datatype columns for those attributes to improve performance

Use SUPER when your queries require strong consistency, predictable query performance, complex query support, and ease of use with evolving schemas & schema-less data

Use Redshift Spectrum instead of loading into SUPER, if data requires integration with other AWS services (e.g. EMR)

```
/*customer-orders-lineitem*/  
CREATE TABLE  
customer_orders_lineitem  
(c_custkey bigint  
,c_name varchar  
,c_address varchar  
,c_nationkey smallint  
,c_phone varchar  
,c_acctbal decimal(12,2)  
,c_mktsegment varchar  
,c_comment varchar  
,c_orders super );
```

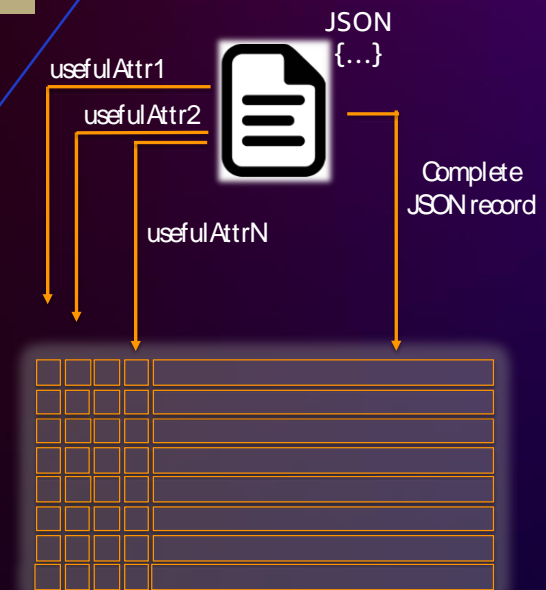


Table Design Best Practices



Redshift table design

THREE MAIN CONCEPTS



Compression
(Column Encoding)



Distribution
style



Sort
keys

Automatic table optimization

TABLE OPTIMIZATION DONE AUTOMATICALLY FOR YOU – NO MANUAL INTERVENTION NEEDED

Continuously scans workload patterns

Automatically adjusts sort key, distribution style and encoding over time to account for changes in workload

Can be enabled or disabled per table/column

Optimizations are applied to tables/columns when load on compute is less

Promotes ease of use, so that you focus on business objectives rather than database management



Column Encryption



Distribution style



Sort keys

Best Practice: Use auto options for compression, distribution and sort keys

Compression/Encoding

Goals

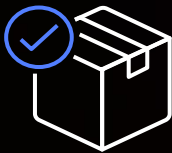
- Allow more data to be stored
- Improve query performance by decreasing I/O

Impact

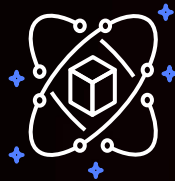
- Allows 3x to 4x times more data to be stored



Column data is persisted to 1 MB immutable blocks



Blocks are individually encoded with 1 of 13 encodings



A full block can contain millions of values after compression

```
CREATE TABLE deep_dive (  
  aid      INT      ENCODE AUTO  
  ,loc     CHAR(3)  ENCODE AUTO  
  ,dt      DATE     ENCODE AZ64  
);
```

aid	loc	dt

Compression Best Practices

- Use default compression
 - By **default** compression encoding is set to **AUTO** for all columns
 - Redshift automatically determines the best compression encoding for that column
- In case you decide to fine-tune by choosing column encoding yourselves:
 - Use **AZ64** where possible
 - Use **ZSTD / LZO** for high cardinality (VAR)CHAR columns
 - Use **BYTEDICT** for low cardinality (VAR)CHAR columns

Distribution Style

Distribution style is a table property that dictates how that table's data is distributed

Goal

- Distribute data evenly for parallel processing
- Minimize data movement during query processing

Impact

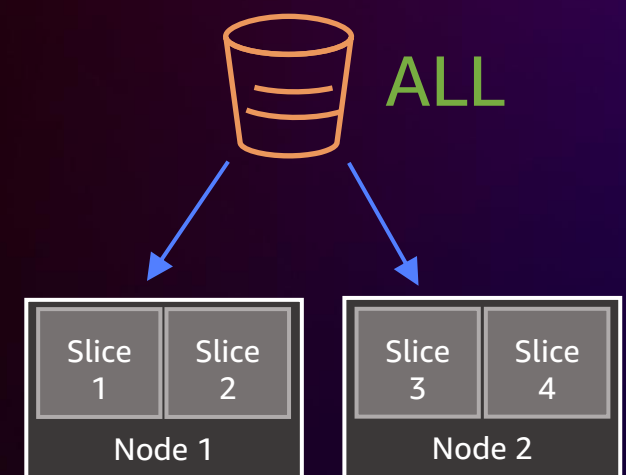
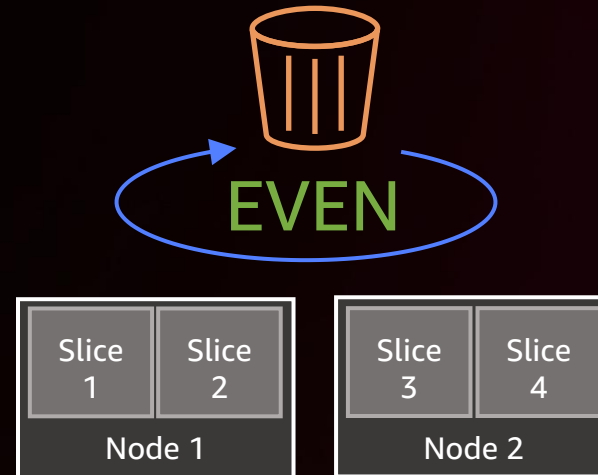
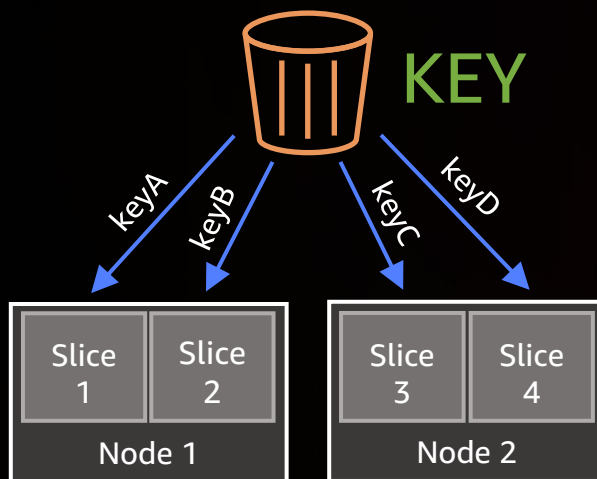
- Minimizes data redistribution by achieving collocation

KEY: Value is hashed; same value goes to same location (slice)

EVEN: Round robin distribution

ALL: Full table data is made available in first slice for all compute node

AUTO: Redshift automatically manages distribution style



Distribution Best Practices

- Use distribution style **AUTO** wherever possible
- Specify the primary key and foreign keys for all your tables. This will help **AUTO** to pick to make efficient decisions
- In case you decide to fine-tune by choosing a distribution style yourselves
 - Use **KEY** style distribution for tables that are frequently joined
 - Use **high cardinality join column** as the distribution key.
 - **Avoid date columns** as the distribution key.
 - When joining fact table with multiple dimension tables, use the same distribution key for fact table and the large dimension table for co-located join.
 - Create a **composite key using FNV_HASH function** for the multi-join columns and use it as the distribution key.
 - Use **ALL** style distribution for small tables, ≤ 5 Million rows
 - Use **EVEN** style distribution if a table is largely denormalized and does not participate in joins, or if you don't have a clear choice for another distribution style

Data sorting

Redshift uses sort keys to physically order data on disk

Goal

- Make queries run faster by increasing the effectiveness of zone maps and reducing I/O




Impact

- Enables range-restricted scans to prune blocks by leveraging zone maps

```
SELECT count(*) FROM deep_dive WHERE dt = '06-09-2020';
```

Unsorted table

3x I/O

	MIN: 01-JUNE-2020 MAX: 20-JUNE-2020
	MIN: 08-JUNE-2020 MAX: 30-JUNE-2020
	MIN: 12-JUNE-2020 MAX: 20-JUNE-2020
	MIN: 02-JUNE-2020 MAX: 25-JUNE-2020

Sorted by "dt"

1x I/O

	MIN: 01-JUNE-2020 MAX: 06-JUNE-2020
	MIN: 07-JUNE-2020 MAX: 12-JUNE-2020
	MIN: 13-JUNE-2020 MAX: 21-JUNE-2020
	MIN: 21-JUNE-2020 MAX: 30-JUNE-2020

Sort Key Best Practices

- Use Sort Key **AUTO** for large tables > 5 Million rows
- **Don't use sort keys for small tables** <= 5 Million rows
- In case you decide to fine-tune by choosing a sort key yourselves for a large table,
 - Pick the column/s that are most commonly used in filters as SORT KEY. For eg: If you query most recent data frequently, date is the most appropriate sort key
 - If you frequently join a table, specify the join column as both the sort key and the distribution key on both tables. This results in merge join which is faster than the otherwise hash join.
 - Don't pick more than 4 columns to be in SORT KEY. When there are more than 4, there is no added benefit from the additional columns
 - When there are more than one column in SORT KEY, their order matters
 - Effective sort key order is **lower to higher cardinality**
 - Low cardinality columns come first high cardinality columns come last
 - Always use the **leading sort key column** in the filter condition
 - Don't apply compression encoding on sort key columns
 - Don't apply functions in queries when using SORT KEY in filters. For Eg: If business_date column is SORT KEY, don't apply a filter **to_char(business_date,'YYYY') = '2023'**

Query Best Practices



Query Best practices

- Avoid using `select *`. Include only the columns you specifically need to reduce I/O
- Use a [CASE expression](#) to perform complex aggregations instead of selecting from the same table multiple times.
- Use subqueries in cases where one table in the query is used only for predicate conditions and the subquery returns a small number of rows (less than about 200). The following example uses a subquery to avoid joining the LISTING table.

Use

```
select sum(sales.qty sold) from sales
where salesid in (
    select listid from listing where listtime > '2023-12-26'
);
```

Instead of

```
select sum(sales.qty sold) from sales
Join listing on sales.salesid = listing.listed
Where listing. listtime > '2023-12-26';
```

Query Best practices

Join:

- Don't use cross-joins unless absolutely necessary
 - Use distribution keys as join columns
-

Aggregate

- Use sort keys in the GROUP BY clause so the query planner can use more efficient aggregation
- If you use both GROUP BY and ORDER BY clauses, make sure that you put the columns in the same order in both

Use `group by a, b, c`
`order by a, b, c`

Instead of

`group by b, c, a`
`order by a, b, c`

Query Best practices

Query Predicate:

- Use predicates to restrict the dataset as much as possible and use sort keys in the predicates
- In the predicate, use the least expensive operators that you can.
 - Comparison condition operators are preferable to LIKE operators.
 - LIKE operators are still preferable to SIMILAR TO or POSIX operators.
- Avoid using functions in query predicates.
- Add predicates to filter tables that participate in joins, even if the predicates apply the same filters

Use

```
select listing.sellerid, sum(sales.qtysold)
from sales, listing
where sales.salesid = listing.listid
and listing.listtime > '2008-12-01'
and sales.saletime > '2008-12-01'
group by 1 order by 1;
```

Instead of

```
select listing.sellerid, sum(sales.qtysold)
from sales, listing
where sales.salesid = listing.listid
and listing.listtime > '2008-12-01'
group by 1 order by 1;
```

Materialized Views

- Improve performance of complex, SLA sensitive, predictable and repeated queries using Materialized views
- Materialized view **persists the result set** of the associated SQL
- Materialized views can be **refreshed automatically** or **manually**
- Redshift automatically determines best way to update data in the materialized view (**incremental** **full refresh**)
- **Automatic query rewrite** leverages relevant materialized views and can improve query performance by order(s) of magnitude
- **Automated materialized views**: Redshift continuously monitors workload to identify queries that will benefit from having a MV and automatically creates and manages MVs for them



Redshift Materialized Views

Materialized views can be created using the **CREATE** statement, and can be included (default) or excluded from Redshift backups. Materialized views can also have table attributes such as *dist style* and *sort keys*, and be refreshed at any time

```
CREATE MATERIALIZED VIEW mv_name  
[ BACKUP { YES | NO } ]  
[ table_attributes ]  
AS query
```

```
REFRESH MATERIALIZED VIEW mv_name;
```


Materialized Views – Best Practices

- Rely on [automated materialized views feature](#), instead of creating your own.
- In case you decide to fine-tune and create materialized views on your own, follow these best practices:
 - Create materialized views that can be [incrementally refreshed](#) in order to avoid full refresh.
 - Schedule [manual refresh](#) for nested materialized views or those not eligible for auto refresh.
 - Follow [query best practices](#) when writing Materialized View queries.
 - Follow [table design best practices](#) on distribution style and sort key when creating the Materialized View.

Nasdaq Uses AWS to Pioneer Stock Exchange Data Storage in the Cloud



“We were able to easily support the jump from 30 billion records to 70 billion records a day because of the flexibility and scalability of Amazon S3 and Amazon Redshift.”

Robert Hunt

Vice President of Software Engineering, Nasdaq

<https://aws.amazon.com/solutions/case-studies/nasdaq-case-study>

Data Lake Modeling Best Practices



Data modeling for Data Lakes Queries

- With Data Lakes, tables are **collections of files**

Data Lake Partitions

- Files can be organized as partitions

For example: Given a table mytable in the s3 location **s3://mybucket/prefix/mytable:** With data organized under prefixes, 2023, 2022, 2021, 2020 etc, The years become partitions for mytable

- Partitions are based on S3 prefix
- Tables may have thousands of partitions

File type, file size and the way files are organized, significantly impacts performance of data lake queries.

Redshift supports read data in:

- **Open file formats:** Parquet, ORC, JSON, CSV, etc.
- **Open table formats:** Apache Hudi, Apache Icerberg, Delta, etc.



Partition:

s3://mybucket/prefix/mytable/

s3://mybucket/prefix/mytable/yyyy=2023
s3://mybucket/prefix/mytable/yyyy=2022
s3://mybucket/prefix/mytable/yyyy=2021
s3://mybucket/prefix/mytable/yyyy=2020

Data Lake Queries Best practices

- Consider **columnar format** (e.g. Parquet, ORC) for performance and cost.
 - With columnar formats, Amazon Redshift reads only the needed columns thereby reducing query cost.
- Set the table statistics (**numRows**) manually for Amazon S3 external tables.
- **Avoid very large size files** (> than 512 MB) and large number of small KB sized files.
 - Supports parallel reads – between 128 MB and 1 GB.
 - Does not support parallel reads – between 64 MB and 128 MB.
- **Partition data on S3** and use frequently filtered columns as partition key.
 - Avoid excessively granular partitions.
 - Columns that are used as common filters are good candidates.
 - Multilevel partitioning is encouraged if you frequently use more than one predicate. Eg: you can partition based on both SHIPDATE and STORE.
 - Create Glue partition Indexes to improve performance of partition pruning.

Data Lake Queries Best practices

- Optimize query cost using **query monitoring rules (QMR)** such as **spectrum_scan_size_mb** or **spectrum_scan_row_count** and also set query performance boundaries on data lake queries .
- **Use GROUP BY clause** - Replace complex DISTINCT operations with GROUP BY in your queries.
- **Choose the right datatype** when creating external tables.
 - Choose **varchar(<<appropriate_length>>)** instead of **varchar(max)**.
 - Choose the **datatype date** instead of **varchar** for dates.
- Monitor and control your Amazon Redshift Spectrum usage and costs using usage limits.

Workload Management Best Practices



Workload management

Allows for the separation of different query workloads

Goal

Prioritize important queries

Throttle / abort less important queries

Queue

Service class that users interact with

Logical separation of user workloads

If you run ETL, dashboards and adhoc queries, create 3 queues, one for each

SQA

Automatically detect short-running queries and runs them within the short query queue, if queuing occurs

Concurrency scaling

When enabled, Amazon Redshift automatically adds transient clusters, in seconds, to serve sudden spike in concurrent requests with consistently fast performance

Query Monitoring Rules

Protects against wasteful use of the compute resources

Rules applied to a WLM queue allow queries to be: LOGGED, ABORTED, HOPPED

Types of workload management

☒ Enable Short Query Acceleration [Learn more](#)

Max Concurrency Scaling clusters: **Manual WLM**

Queue 1 [Delete](#) [^](#) [v](#)

Memory (%) Concurrency on main Concurrency Scaling mode Timeout (ms) User groups ☐ Match wildcards ☐ Match wildcards [+](#) [+](#)

[v](#) Query Monitoring Rules (0) [Add rule from templates](#) [Add custom rule](#)

No rules have been defined.

Queue 2 [Delete](#) [^](#) [v](#)

Memory (%) Concurrency on main Concurrency Scaling mode Timeout (ms) User groups ☐ Match wildcards ☐ Match wildcards [+](#) [+](#)

[v](#) Query Monitoring Rules (0) [Add rule from templates](#) [Add custom rule](#)

No rules have been defined.

Default queue [Delete](#) [^](#) [v](#)

Memory (%) Concurrency on main Concurrency Scaling mode Timeout (ms) [v](#) [Auto](#)

[v](#) Query Monitoring Rules (0) [Add rule from templates](#) [Add custom rule](#)

Dashboard **Auto WLM**

Memory (%) **Concurrency on main** **Concurrency scaling mode** **Query priority**

Auto Auto Auto Highest

User groups dashboard Query groups

[v](#) Query monitoring rules (0)

ETL

Memory (%) Concurrency on main Concurrency scaling mode Query priority

Auto Auto - Normal

User groups etl Query groups

[v](#) Query monitoring rules (0)

Adhoc

This is the default queue.

Memory (%) Concurrency on main Concurrency scaling mode Query priority

Auto Auto - Low

Memory allocation

Concurrency

Prioritization

De-prioritization

SQA

Concurrency Scaling and QMR

Manual and static

Manual and static

Cannot be done

Cannot be done

Can be enabled manually

Configurable for each queue

Automatic and dynamic

Automatic and dynamic

Can be done at queue level

Can be done at query level using QMR

Automatically enabled

Configurable for each queue

Workload management: Best Practices

- Use Auto WLM if your workload is highly unpredictable, or you are using default WLM.
- Use QMR on `query_execution_time`, `query_temp_blocks_to_disk` and `spectrum_scan_size_mb` or `spectrum_scan_row_count`.

Only for Manual WLM

- Use manual WLM if you want to manually fine-tune and completely understand your workload patterns or require throttling certain types of queries depending on the time of day.
- Keep #WLM queues to a minimum, typically just three queues, to avoid having unused queues.
- Limit ingestion/ELT concurrency to two to three.
- To maximize query throughput, use ensure the total concurrent queries is 15 or less.
- Save the superuser queue for administration tasks and canceling queries.

Data Load Best Practices



Data Ingestion: AWS Services

	Batch	Near Real Time
S3 file	<ul style="list-style-type: none">• COPY command• Redshift Spectrum	<ul style="list-style-type: none">• COPY job – S3 triggered
Streaming sources		<ul style="list-style-type: none">• Streaming Ingestion (Amazon Kinesis, Amazon MSK)
Transactional databases	<ul style="list-style-type: none">• AWS Database Migration Service	<ul style="list-style-type: none">• Zero ETL Integration from Aurora• AWS Database Migration Service
Amazon Data Exchange (ADX)	<ul style="list-style-type: none">• Third Party Data available in ADX	<ul style="list-style-type: none">• Third Party Data available in ADX
SaaS Applications	<ul style="list-style-type: none">• Amazon AppFlow	<ul style="list-style-type: none">• Amazon AppFlow

COPY Command

Used to ingest data from

- Amazon S3 (most common source)

- Amazon EMR

- Amazon DynamoDB

- Remote host (SSH)

Can ingest files in various formats, compression schemes

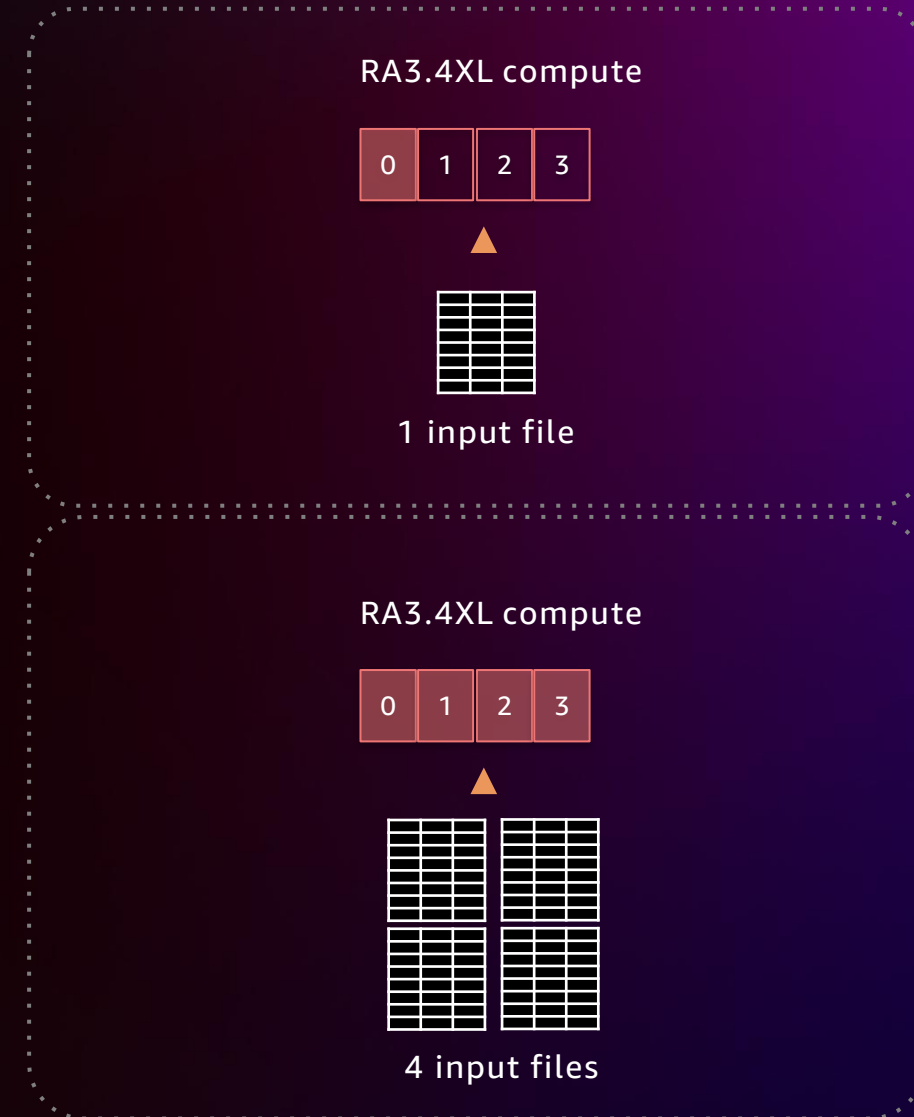
- File format: CSV, JSON, Avro, Parquet, ORC etc

- Compression Options: BZIP2, GZIP, LZOP, ZSTD

- Encryption

One compute slice can process one file

COPY continues to scale linearly as you add more compute



COPY Command Best Practices

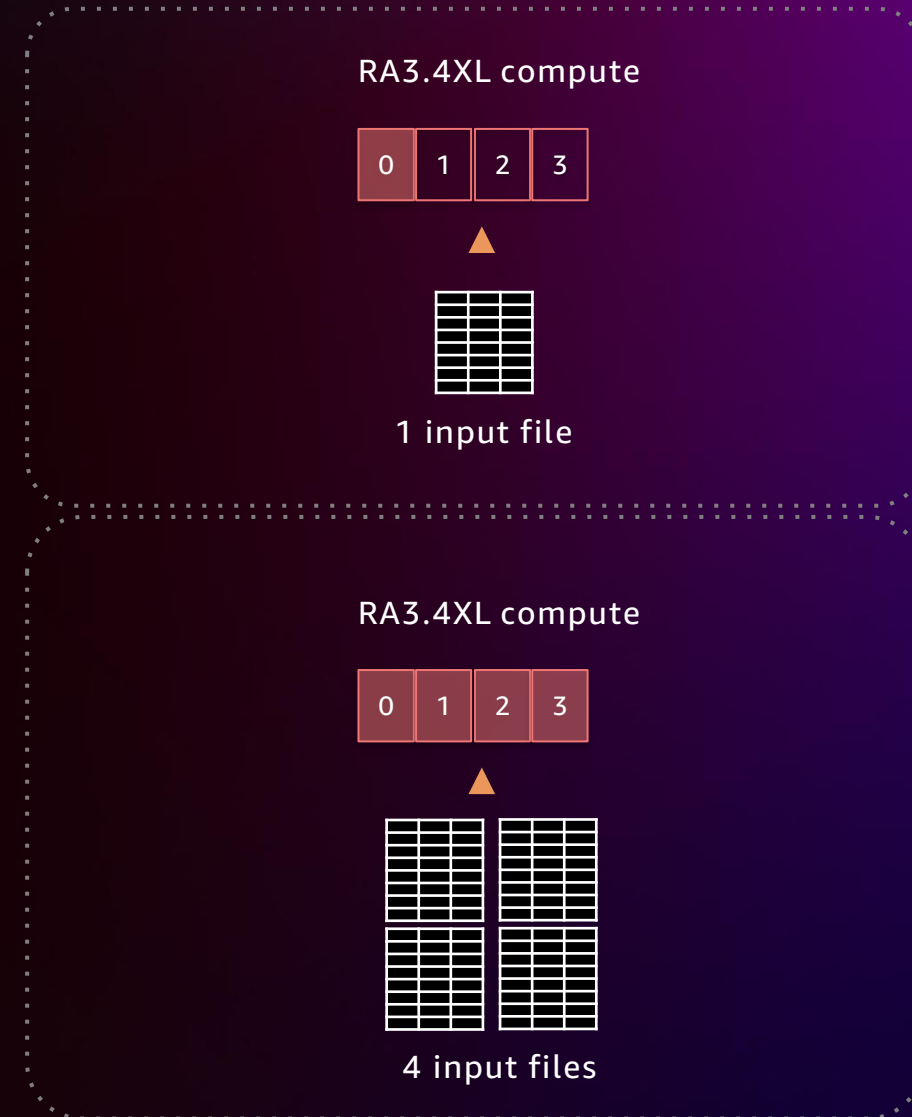
Use COPY command to load data whenever possible.

Use a single COPY command per table.

When using COPY, avoid loading from many small files or large non-splittable files.

If COPY is not possible, do bulk inserts using `INSERT` statement. Avoid single row inserts.

- In using COPY command, [optimal file size](#) are:
 - For [non-splittable file](#) when each file is between 1MB-1GB each after compression
 - For [splittable file](#) when each file size is:
 - Between 128MB-1GB for [columnar files](#), specifically Parquet and ORC.
 - Between 64MB-10GB for [row-oriented \(CSV\) data](#) that do not use any these keywords – REMOVEQUOTES, ESCAPE and FIXEDWIDTH.



Data Loading Best Practices

Load your data in **sort key order** to avoid needing to vacuum.

For **large amounts** of data, load in **small sequential blocks** according to sort order:
eliminates the need to **vacuum**.

you use much **less intermediate sort space** during each load, and makes it **easier to restart** if the COPY fails and is rolled back.

For data with **fixed retention period**, organize your data as a sequence of **time-series tables**.

Use **MERGE** statement to perform upserts.

Enforce Primary, Unique or Foreign Key constraints in ETL.

Wrap workflow/statements in an explicit transaction.

Consider using **TRUNCATE** instead of **DELETE**.

Post Data Load Best Practices



(AUTO) VACUUM

- The VACUUM process runs either manually or automatically in the background
- **Goals**
 - VACUUM will remove rows that are marked as deleted
 - VACUUM will globally sort tables
 - For tables with a sort key, ingestion operations will locally sort new data and write it into the unsorted region
- **Best practices**
 - VACUUM should be run only as necessary
 - For the majority of workloads, **AUTO VACUUM DELETE** will reclaim space and **AUTO TABLE SORT** will sort the needed portions of the table
 - In cases where you know your workload – VACUUM can be run manually
 - Run vacuum operations on a regular schedule
 - Perform vacuum re-cluster on large tables

(AUTO) ANALYZE

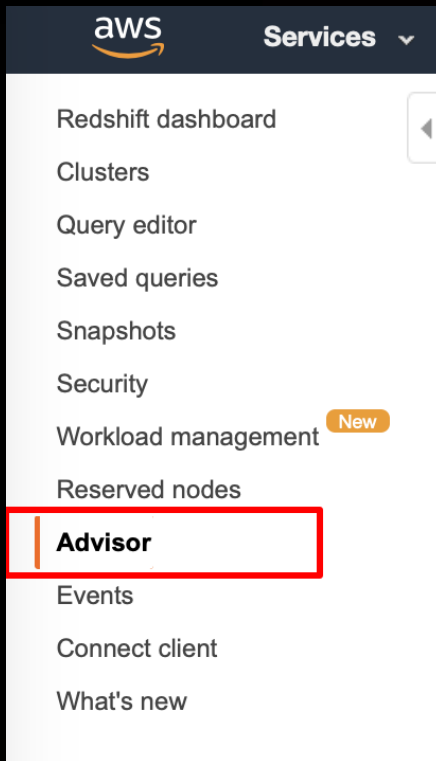
- The ANALYZE process collects table statistics for optimal query planning
- In the vast majority of cases, AUTO ANALYZE automatically handles statistics gathering
- Best practices
 - For the majority of workloads, **AUTO ANALYZE will collect statistics**
 - ANALYZE can be run periodically after ingestion on just the columns that WHERE predicates are filtered on
 - Analyze after VACUUM
 - Utility to manually run VACUUM and ANALYZE on all the tables in the cluster:
<https://bit.ly/34ZR3PP>

Amazon Redshift Advisor

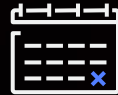


Amazon Redshift Advisor

To improve the performance and decrease the operating costs, Amazon Redshift Advisor offers specific recommendations by analyzing performance and usage metrics. Advisor ranks recommendations by order of impact.



- Amazon Redshift Advisor available in Amazon Redshift console



- Runs daily scanning operational metadata



- Observes with the lens of best practices

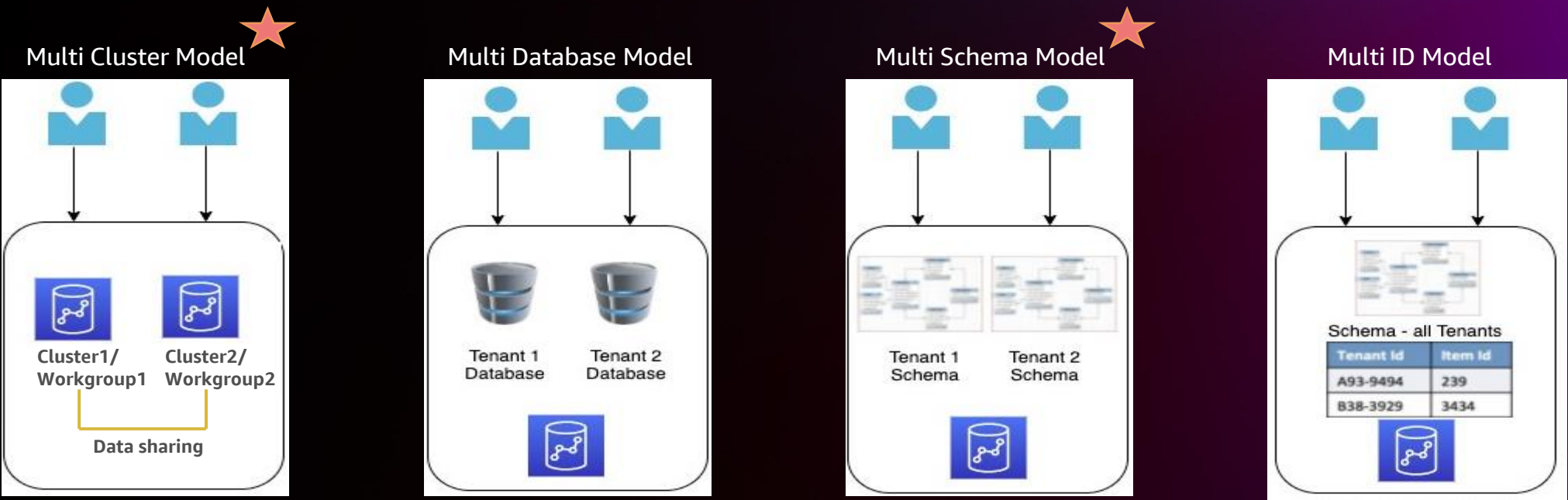


- Provides tailored, high-impact recommendations to optimize your Amazon Redshift cluster for performance and cost savings

Best Practices for Multi-Tenancy Architectures



Multi Tenant Strategy



Workload Type	<ul style="list-style-type: none">• Completely Isolated workloads, yet share data among tenants• Distributed data repositories	Workloads having different <ul style="list-style-type: none">• security policies• Isolation levels, collation	Workloads requiring <ul style="list-style-type: none">• common security policies• same isolation & collation• frequent queries across tenants	Workloads requiring <ul style="list-style-type: none">• same storage constructs. Access control with RLS• Same tables, views across tenants.
Scalability	Highly scalable model	Scaling is limited to cluster	Scaling is limited to cluster	Scaling is limited to cluster
Cross-tenant R/W	Supported for Reads. No writes	Supported for Reads. No writes	Both Reads/Writes	Both Reads/Writes
Examples	<ul style="list-style-type: none">• Separating ETL, BI workloads• Dev, QA, PROD DBs	<ul style="list-style-type: none">• Independent business units with no/limited cross querying• Banking DB and Insurance DB	<ul style="list-style-type: none">• Multiple departments, functional units often cross query data• Sales, Finance, Marketing	Each tenant is a persona accessing from same storage structure



Thank you!

