

Amazon Redshift Reference

AWS's cloud data warehouse — provisioned and serverless deployment, data distribution, loading from S3, performance tuning, and integration with the data lake.

Last reviewed 29 Sep 2026 ·

Prerequisites: [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) · [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/)

Related: [Snowflake](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) · [BigQuery](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/) · [Azure and Fabric](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/azure-fabric/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/azure-fabric/) · [Data Modeling](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) · [Terraform](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/terraform-for-de/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/terraform-for-de/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

Challenge: Teams running analytics on AWS need a warehouse that integrates with S3, IAM, and the rest of the AWS ecosystem, handles terabytes to petabytes, and supports both steady scheduled workloads and bursty ad hoc queries.

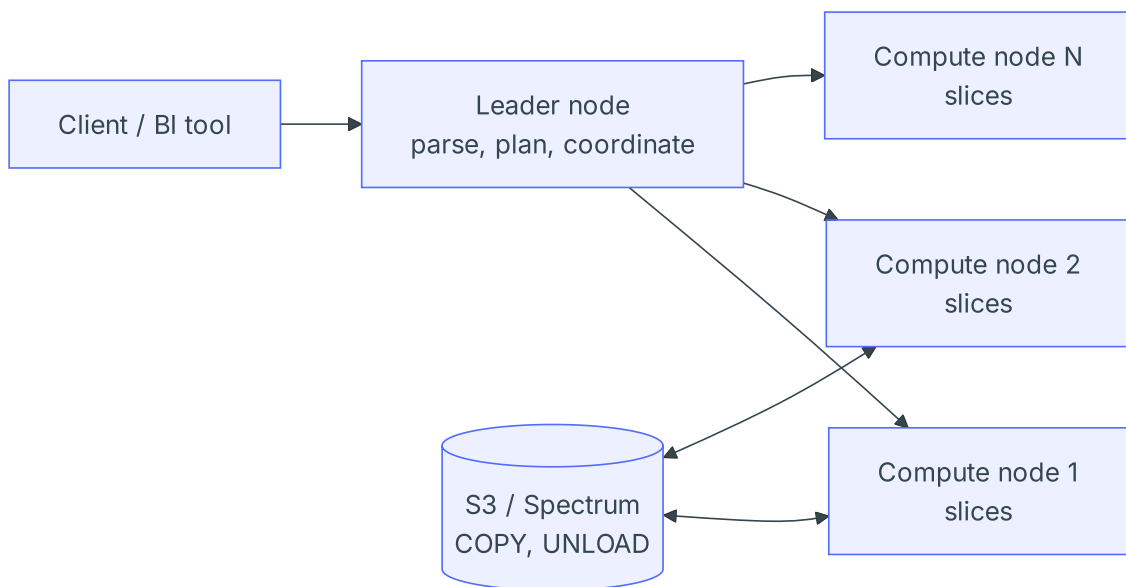
Solution: Amazon Redshift is a columnar, massively parallel processing (MPP) warehouse. It runs either as **provisioned** clusters (RA3 nodes with managed storage in S3) or as **Redshift Serverless** (capacity in Redshift Processing Units, billed per second of use). It reads and writes S3 directly, queries the data lake through Spectrum, and shares data across clusters without copying.

```
graph TD; Leader[Leader node - parses SQL, builds the plan, coordinates] --- Compute1[Compute node]; Leader --- Compute2[Compute node]; Leader --- Compute3[Compute node]; Leader --- Dots[...];
```

each node holds slices of every table

(how rows are spread = distribution style)
Redshift Managed Storage (S3-backed)
Spectrum → S3 data lake

Relevance to data engineering: Redshift performance depends on data layout decisions — distribution and sort keys — and on loading data in bulk. Newer automatic features (auto distribution, auto sort, automatic vacuum and analyze) reduce but do not remove the need to understand them.



On this page

Basic - [Deployment Options](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#deployment-options) (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#deployment-options>) - [Tables, Distribution, and Sort Keys](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#tables-distribution-and-sort-keys) (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#tables-distribution-and-sort-keys>) - [Loading and Unloading](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#loading-and-unloading) (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#loading-and-unloading>)

Intermediate - [Semi-Structured Data \(SUPER\)](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#semi-structured-data-super) (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#semi-structured-data-super>) - [MERGE and Incremental Loads](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#merge-and-incremental-loads) (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#merge-and-incremental-loads>) - [Querying the Data Lake with Spectrum](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#querying-the-data-lake-with-spectrum) (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#querying-the-data-lake-with-spectrum>) - [Connecting from Python](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#connecting-from-python) (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#connecting-from-python>)

Advanced - [Workload Management and Concurrency](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#workload-management-and-concurrency) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#workload-management-and-concurrency) -

[Performance Tuning](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#performance-tuning) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#performance-tuning) - [Data Sharing and Zero-ETL](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#data-sharing-and-zero-etl) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#data-sharing-and-zero-etl) - [Security](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#security) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#security)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/#further-reading)

Deployment Options

	Provisioned (RA3)	Serverless
Capacity	Choose node type and count; resize or pause	Base capacity in RPUs; scales automatically
Billing	Per node-hour (reserved instances available) + managed storage	Per RPU-second while queries run + managed storage
Best for	Steady, predictable, heavy workloads	Variable or intermittent workloads; new projects
Operations	Resize, pause/resume, maintenance windows	Minimal

Both use Redshift Managed Storage, so storage scales independently of compute.

Tables, Distribution, and Sort Keys

```
CREATE TABLE analytics.fct_orders (  
  order_id      BIGINT      NOT NULL,  
  customer_id   BIGINT      NOT NULL,  
  order_date    DATE        NOT NULL,  
  amount        DECIMAL(12,2),  
  status        VARCHAR(20)
```

```

)
DISTSTYLE KEY
DISTKEY (customer_id)          -- co-locate rows joined on
customer_id
SORTKEY (order_date);          -- range filters on date
skip blocks via zone maps

CREATE TABLE analytics.dim_region (
    region_id INT,
    region_name VARCHAR(50)
)
DISTSTYLE ALL;                  -- small dimension: a full
copy on every node, no redistribution in joins

```

Distribution style	How rows are placed	Use for
<code>AUTO</code> (default)	Redshift chooses and adapts (<code>ALL</code> → <code>EVEN</code> / <code>KEY</code> as the table grows)	Most tables — start here
<code>KEY</code>	By hash of one column	Large tables frequently joined on that column
<code>ALL</code>	Full copy on every node	Small, slowly changing dimensions
<code>EVEN</code>	Round-robin	Large tables with no dominant join key

Sort keys determine physical row order; zone maps (min/max per block) let filters skip blocks. Use a compound sort key led by the most common filter (usually a date), or `SORTKEY AUTO`. Interleaved sort keys are rarely worth their maintenance cost.

Loading and Unloading

```

-- Bulk load from S3 (parallel across all slices)
COPY analytics.stg_orders
FROM 's3://my-lake/orders/order_date=2024-03-15/'
IAM_ROLE 'arn:aws:iam::123456789012:role/redshift-s3-
access'
FORMAT AS PARQUET;

```

```

-- CSV with options
COPY raw.customers
FROM 's3://my-lake/customers/'
IAM_ROLE 'arn:aws:iam::123456789012:role/redshift-s3-
access'
CSV IGNOREHEADER 1 TIMEFORMAT 'auto' GZIP;

-- Export query results to S3 as Parquet
UNLOAD ('SELECT * FROM analytics.fct_orders WHERE
order_date = ''2024-03-15''')
TO 's3://my-lake/exports/fct_orders/order_date=2024-03-15/'
IAM_ROLE 'arn:aws:iam::123456789012:role/redshift-s3-
access'
FORMAT AS PARQUET;

-- Inspect load errors
SELECT * FROM sys_load_error_detail ORDER BY start_time
DESC LIMIT 20;

```

- Load with **COPY from many files** (a multiple of the number of slices) rather than `INSERT` statements
- Ingest continuously with auto-copy from S3, streaming ingestion from Kinesis/MSK, or zero-ETL integrations

Semi-Structured Data (SUPER)

```

CREATE TABLE raw.events (
    event_id VARCHAR(64),
    event_ts TIMESTAMP,
    payload SUPER -- schemaless JSON
value
);

INSERT INTO raw.events
SELECT 'e1', GETDATE(), JSON_PARSE('{"campaign": "spring",
"items": [{"sku": "A1", "qty": 2}]}');

-- Navigate with PartiQL dot/bracket notation
SELECT event_id,

```

```
        payload.campaign::VARCHAR AS campaign,
        item.sku::VARCHAR          AS sku,
        item.qty::INT              AS qty
FROM raw.events AS e, e.payload.items AS item; -- unnest
an array
```

MERGE and Incremental Loads

```
MERGE INTO analytics.customers
USING staging.customers_changes AS s
ON analytics.customers.customer_id = s.customer_id
WHEN MATCHED THEN UPDATE SET email = s.email, country =
s.country, updated_at = s.updated_at
WHEN NOT MATCHED THEN INSERT VALUES (s.customer_id,
s.email, s.country, s.updated_at);

-- Partition-style reload inside a transaction
BEGIN;
DELETE FROM analytics.fct_orders WHERE order_date = '2024-
03-15';
INSERT INTO analytics.fct_orders SELECT * FROM
staging.orders WHERE order_date = '2024-03-15';
COMMIT;
```

Deletes and updates leave space to reclaim and rows out of sort order; automatic vacuum handles this in the background, and `VACUUM` / `ANALYZE` can be run explicitly after very large changes.

Querying the Data Lake with Spectrum

```
-- Map an AWS Glue Data Catalog database as an external
schema
CREATE EXTERNAL SCHEMA lake
FROM DATA CATALOG
DATABASE 'lake_db'
IAM_ROLE 'arn:aws:iam::123456789012:role/redshift-spectrum'
CREATE EXTERNAL DATABASE IF NOT EXISTS;
```

```
-- Join lake data (Parquet in S3) with warehouse tables
SELECT d.region_name, SUM(e.amount) AS revenue
FROM lake.orders_history AS e                -- external
table in S3
JOIN analytics.dim_region AS d ON e.region_id = d.region_id
WHERE e.order_date >= '2023-01-01'          -- partition
pruning in S3
GROUP BY d.region_name;
```

Keep hot, frequently joined data in Redshift and cold history in the lake; Spectrum is billed by bytes scanned in S3, so partitioned Parquet matters. Redshift can also read Apache Iceberg tables registered in the Glue catalog.

Connecting from Python

```
import redshift_connector    # pip install
redshift_connector

conn = redshift_connector.connect(
    host="analytics.123456789012.eu-west-1.redshift-
serverless.amazonaws.com",
    database="dev",
    iam=True,                    # IAM auth
    -- no stored passwords
    region="eu-west-1",
)
cur = conn.cursor()
cur.execute("SELECT order_date, SUM(amount) FROM
analytics.fct_orders GROUP BY 1 ORDER BY 1 DESC LIMIT 7")
df = cur.fetch_dataframe()
```

```
import time
import boto3

# Data API – HTTPS, no drivers or persistent connections
(good for Lambda and orchestrators)
client = boto3.client("redshift-data")
resp = client.execute_statement(
    WorkgroupName="analytics",          # or
    ClusterIdentifier=... for provisioned
```

```
Database="dev",
Sql="CALL analytics.refresh_daily_aggregates()",
)
while client.describe_statement(Id=resp["Id"])["Status"]
not in ("FINISHED", "FAILED", "ABORTED"):
    time.sleep(2)
```

Workload Management and Concurrency

- **Automatic WLM** (recommended) allocates memory and concurrency dynamically; assign **query priorities** to queues (e.g. ETL high, ad hoc normal)
- **Query monitoring rules** abort or log runaway queries (e.g. more than N rows returned or M seconds of runtime)
- **Concurrency scaling** adds transient capacity for bursts of read queries (and some writes)
- **Short query acceleration** runs short queries ahead of long ones
- Serverless handles concurrency by scaling RPU's; set a **maximum capacity** and usage limits to cap cost

Performance Tuning

```
-- Tables with skew or unsorted data
SELECT "table", diststyle, skew_rows, unsorted, tbl_rows,
size AS size_mb
FROM svv_table_info
ORDER BY skew_rows DESC NULLS LAST
LIMIT 20;

-- Slowest recent queries
SELECT query_id, status, elapsed_time / 1000000.0 AS
seconds, LEFT(query_text, 120) AS query
FROM sys_query_history
WHERE start_time >= DATEADD(day, -1, GETDATE())
ORDER BY elapsed_time DESC
LIMIT 20;
```

Symptom	Likely cause	Fix
Large <code>DS_BCAST_INNER</code> / <code>DS_DIST_BOTH</code> steps in <code>EXPLAIN</code>	Data redistributed across nodes for a join	Matching <code>DISTKEY</code> on join columns, or <code>DISTSTYLE ALL</code> for small dimensions
One slice much busier than others	Distribution skew on a low-cardinality or null-heavy key	Choose a higher-cardinality key or <code>EVEN</code> / <code>AUTO</code>
Range filters scan everything	Missing or wrong sort key; high unsorted percentage	Sort key on the filter column; let auto vacuum sort, or run <code>VACUUM SORT ONLY</code>
Bad plans after big loads	Stale statistics	<code>ANALYZE</code> (automatic analyze usually handles it)
Repeated heavy dashboard queries	Recomputing the same aggregates	Materialized views with <code>AUTO REFRESH YES</code>

Data Sharing and Zero-ETL

- **Data sharing:** share live, read-only data between Redshift clusters, workgroups, and accounts without copying — e.g. a producer ETL cluster and separate consumer clusters per team
- **Zero-ETL integrations:** replicate data from operational databases (such as Amazon Aurora and RDS) into Redshift without building pipelines
- **Federated queries:** query live data in RDS/Aurora PostgreSQL and MySQL from Redshift
- **Redshift ML:** train and run models with SQL (`CREATE MODEL`) backed by SageMaker

Security

Area	Mechanism
Authentication	<u>IAM</u> (temporary credentials), identity provider federation / <u>IAM</u> Identity Center
Authorization	Users, groups, and roles (<u>RBAC</u>); <code>GRANT</code> on schemas, tables, columns

Area	Mechanism
Row-level security	<u>RLS</u> policies attached to tables and roles
Masking	Dynamic data masking policies
Encryption	At rest with <u>KMS</u> ; TLS in transit
Network	VPC, security groups, private endpoints; enhanced VPC routing for COPY/UNLOAD
Audit	Audit logging to S3 or CloudWatch; <code>SYS_*</code> monitoring views

Common Pitfalls

Pitfall	Symptom	Fix
Row-by-row <code>INSERT</code> s	Very slow loads	<code>COPY</code> from S3 in bulk, from multiple files
<code>DISTKEY</code> on a skewed or low-cardinality column	One node does most of the work	High-cardinality join key, <code>EVEN</code> , or <code>AUTO</code>
Mismatched distribution on large joins	Heavy redistribution (<code>DS_DIST_BOTH</code>)	Same <code>DISTKEY</code> on both sides of frequent large joins
No sort key on date-filtered fact tables	Scans every block	Compound sort key led by the date column
Too many small files for <code>COPY</code> or Spectrum	Slow loads and scans	Consolidate into files of ~100 MB–1 GB
Long-running ad hoc queries blocking ETL	Missed SLAs	Workload priorities, query monitoring rules, concurrency scaling or separate workgroups
Using Redshift as an OLTP store	Poor performance for single-row lookups and updates	Keep operational workloads in an OLTP database
Static passwords in pipelines	Credential sprawl	<u>IAM</u> authentication and the Data API

Pitfall	Symptom	Fix
Serverless without usage limits	Unexpected bills	Maximum RPU capacity and usage limits with alerts

Cheat Sheet

Task	SQL / command
Bulk load	<code>COPY t FROM 's3://...' IAM_ROLE '...' FORMAT AS PARQUET</code>
Export	<code>UNLOAD ('SELECT ...') TO 's3://...' IAM_ROLE '...' FORMAT AS PARQUET</code>
Upsert	<code>MERGE INTO t USING s ON ... WHEN MATCHED ... WHEN NOT MATCHED ...</code>
Table layout	<code>DISTSTYLE AUTO\ KEY\ ALL\ EVEN · DISTKEY (col) · SORTKEY (col)</code>
JSON	<code>SUPER column · JSON_PARSE(text) · payload.field::VARCHAR</code>
Lake tables	<code>CREATE EXTERNAL SCHEMA ... FROM DATA CATALOG ...</code>
Skew / unsorted	<code>SELECT ... FROM svv_table_info</code>
Query history	<code>sys_query_history</code>
Load errors	<code>sys_load_error_detail</code>
Refresh stats	<code>ANALYZE t</code>
Precompute	<code>CREATE MATERIALIZED VIEW mv AUTO REFRESH YES AS SELECT ...</code>

Choosing a distribution style: start with `AUTO` · large fact + large dimension joined on one key → `KEY` on both · small dimension → `ALL` · no dominant join → `EVEN`

Interview Questions

Q: Explain distribution styles in Redshift and how you choose one. A: Distribution decides which slice stores each row. `KEY` hashes a column so rows with the same key land together — ideal when two large tables are joined on that key, since the join needs no data movement. `ALL` copies a small table to every node so any join with it is local. `EVEN`

spreads rows round-robin when there's no dominant join. `AUTO` lets Redshift choose and adapt as tables grow. The goal is to minimize data redistribution during joins while avoiding skew.

Q: What do sort keys do? A: They define the physical order of rows on disk. Redshift keeps min/max values (zone maps) per block, so a filter on the sort key can skip blocks whose ranges don't match. A compound sort key led by the most common filter column — typically a date — gives the biggest benefit for range filters and merge joins.

Q: How do you load data into Redshift efficiently? A: Use `COPY` from S3, which loads in parallel across all slices; split input into multiple compressed files (ideally a multiple of the slice count) of reasonable size, and prefer columnar formats like Parquet. Avoid row-by-row inserts. Load into staging tables and apply with `MERGE` or delete-and-insert in a transaction for idempotency, and check `sys_load_error_detail` for rejected rows.

Q: Provisioned or Serverless — how do you decide? A: Serverless suits variable or intermittent workloads and new projects: no capacity planning, per-second billing while queries run, automatic scaling. Provisioned RA3 clusters suit steady, heavy, predictable workloads, where reserved pricing and fixed capacity are cheaper and performance is more predictable. Both use managed storage, and data sharing lets you mix them — for example provisioned for ETL and serverless for ad hoc consumers.

Q: How do Redshift, BigQuery, and Snowflake differ at a high level? A: All are columnar, separate storage from compute (Redshift via RA3/managed storage), and support standard SQL. Redshift is AWS-native, exposes more physical tuning (distribution and sort keys), and offers provisioned or serverless capacity. BigQuery is fully serverless with slot-based execution and per-byte or capacity pricing on Google Cloud. Snowflake runs on all three major clouds with independently sized virtual warehouses billed per second. The choice usually follows the cloud platform, workload shape, pricing model fit, and team skills.

Further Reading

- [Amazon Redshift documentation](https://docs.aws.amazon.com/redshift/) (https://docs.aws.amazon.com/redshift/)
 - [Amazon Redshift best practices for designing tables](https://docs.aws.amazon.com/redshift/latest/dg/c_designing-tables-best-practices.html) (https://docs.aws.amazon.com/redshift/latest/dg/c_designing-tables-best-practices.html)
 - [Loading data best practices](https://docs.aws.amazon.com/redshift/latest/dg/c_loading-data-best-practices.html) (https://docs.aws.amazon.com/redshift/latest/dg/c_loading-data-best-practices.html)
 - [Amazon Redshift Serverless](https://docs.aws.amazon.com/redshift/latest/mgmt/working-with-serverless.html) (https://docs.aws.amazon.com/redshift/latest/mgmt/working-with-serverless.html)
 - [redshift_connector](https://github.com/aws/amazon-redshift-python-driver) (https://github.com/aws/amazon-redshift-python-driver) — the official Python driver
-

Previous: [BigQuery](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/) · **Next:** [Azure and Fabric](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/azure-fabric/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/azure-fabric/) · **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026