

BigQuery Reference

Google Cloud's serverless data warehouse — architecture, loading, partitioning, cost control, and security.

Last reviewed 29 Sep 2026 ·

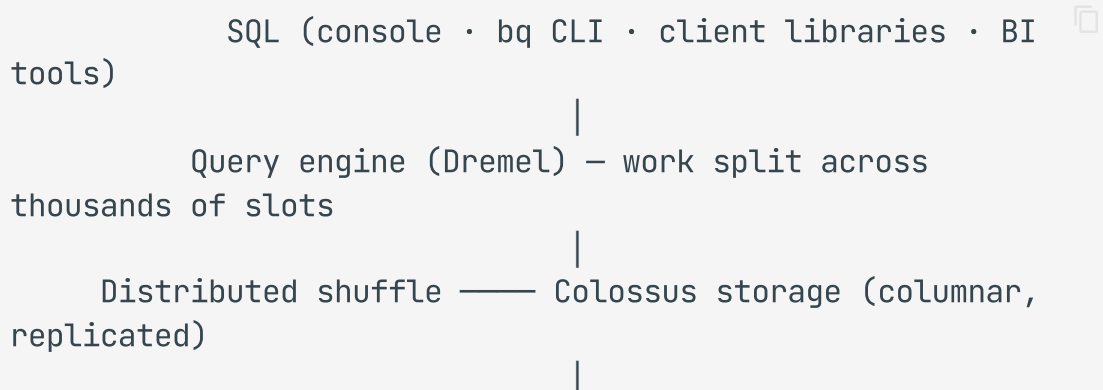
Prerequisites: [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) · [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) (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/) · [Amazon Redshift](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/) · [Data Modeling](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) · [Cost Optimization](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

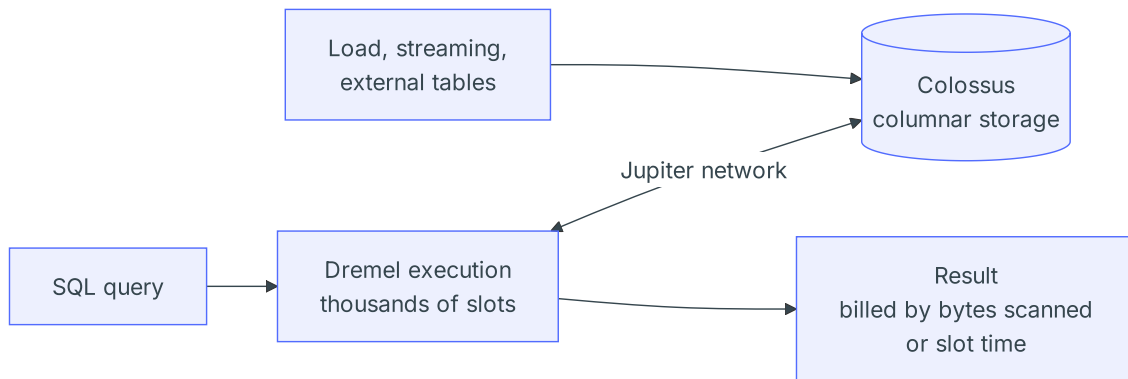
Challenge: Running a warehouse usually means sizing clusters, tuning indexes, and paying for idle capacity. Teams want to query terabytes with SQL without managing infrastructure.

Solution: BigQuery is a serverless warehouse: storage and compute are separate, there are no clusters to manage, and each query is executed on a shared pool of compute units called **slots**. Compute is billed either per byte scanned (on-demand) or per slot capacity (editions with autoscaling), and storage is billed separately.



Native tables • external tables (Cloud Storage, BigLake / Iceberg) • streaming ingestion

Relevance to data engineering: BigQuery's cost and performance depend almost entirely on how much data each query reads, so partitioning, clustering, and query discipline matter more than any server tuning.



On this page

Basic - [Resource Hierarchy](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#resource-hierarchy) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#resource-hierarchy) - [Querying](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#querying) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#querying) - [Nested and Repeated Data](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#nested-and-repeated-data) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#nested-and-repeated-data)

Intermediate - [Loading Data](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#loading-data) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#loading-data) - [Partitioning and Clustering](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#partitioning-and-clustering) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#partitioning-and-clustering) - [DML and MERGE](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#dml-and-merge) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#dml-and-merge) - [Python Client](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#python-client) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#python-client)

Advanced - [Pricing Models and Cost Control](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#pricing-models-and-cost-control) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#pricing-models-and-cost-control) - [Performance](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#performance) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#performance) - [Time Travel, Snapshots, and Clones](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#time-travel-snapshots-and-clones) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#time-travel-snapshots-and-clones) - [Security and Governance](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#security-and-governance) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/#security-and-governance)

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

Resource Hierarchy

```
Organization
└─ Project (billing, quotas, IAM)
    └─ Dataset (a location: US, EU, or a region;
        default table settings)
        └─ Tables · views · materialized views ·
            routines (UDFs, procedures) · models
```

- Fully qualified name: ``project.dataset.table``
- A dataset's **location** is fixed at creation; queries can only join datasets in the same location
- Access is granted with IAM on the project, dataset, or table

```
CREATE SCHEMA IF NOT EXISTS analytics
OPTIONS (location = "EU", default_table_expiration_days =
NULL);
```

Querying

BigQuery uses GoogleSQL (standard SQL). Most of the [SQL guide](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/>) applies directly.

```
-- Latest record per key with QUALIFY
SELECT *
FROM `my-project.analytics.orders`
WHERE order_date >= "2024-03-01"
QUALIFY ROW_NUMBER() OVER (PARTITION BY order_id ORDER BY
updated_at DESC) = 1;

-- Useful functions
SELECT
    DATE_TRUNC(order_date, MONTH)           AS month,
    SAFE_DIVIDE(SUM(refunds), SUM(revenue)) AS refund_rate,
    -- NULL instead of divide-by-zero
    APPROX_COUNT_DISTINCT(customer_id)      AS
approx_customers, -- fast, ~1% error
```

```

COUNTIF(status = "cancelled") AS cancelled
FROM `my-project.analytics.orders`
GROUP BY month;

-- Parameterized query (from client libraries)
SELECT * FROM `my-project.analytics.orders` WHERE
order_date = @run_date;

```

Before running an expensive query: check the estimated bytes in the console, or run `bq query --dry_run`.

Nested and Repeated Data

BigQuery stores nested (`STRUCT`) and repeated (`ARRAY`) fields natively — often better than joining child tables.

```

CREATE TABLE analytics.orders_nested (
  order_id    STRING,
  customer    STRUCT<id STRING, country STRING>,
  line_items  ARRAY<STRUCT<sku STRING, quantity INT64,
price NUMERIC>>,
  order_ts    TIMESTAMP
);

-- Explode line items into rows
SELECT o.order_id, li.sku, li.quantity * li.price AS
line_total
FROM analytics.orders_nested AS o,
      UNNEST(o.line_items) AS li
WHERE o.customer.country = "DE";

-- Aggregate back into an array
SELECT customer.id, ARRAY_AGG(order_id ORDER BY order_ts
DESC LIMIT 5) AS last_orders
FROM analytics.orders_nested
GROUP BY customer.id;

-- Semi-structured JSON column
SELECT JSON_VALUE(properties, "$.campaign") AS campaign
FROM analytics.events;

```

Loading Data

Method	Use for	Notes
Batch load jobs (<code>bq load</code> , <code>LOAD DATA</code>)	Files in Cloud Storage	Free to load (you pay for storage)
Storage Write API	Streaming and high-throughput programmatic writes	Exactly-once semantics with committed streams
External tables / BigLake	Query files in place (Parquet, ORC, Avro, CSV, JSON, Iceberg)	No load step; performance depends on file layout
BigQuery Data Transfer Service	Scheduled imports from SaaS and other clouds	Managed connectors
Datastream	<u>CDC</u> from operational databases	Near-real-time replication

```
-- Load Parquet files from Cloud Storage
LOAD DATA INTO analytics.orders
FROM FILES (
  format = "PARQUET",
  uris    = ["gs://my-lake/orders/order_date=2024-03-15/*.parquet"]
);
```

```
-- External table over files in Cloud Storage
CREATE EXTERNAL TABLE raw.clickstream
OPTIONS (format = "PARQUET", uris = ["gs://my-lake/clickstream/*.parquet"]);
```

```
bq load --source_format=PARQUET analytics.orders "gs://my-lake/orders/order_date=2024-03-15/*.parquet"
bq load --source_format=CSV --skip_leading_rows=1 --autodetect raw.customers gs://my-lake/customers.csv
```

Partitioning and Clustering

```

CREATE TABLE analytics.events (
  event_id    STRING NOT NULL,
  user_id     STRING,
  event_type  STRING,
  event_ts    TIMESTAMP,
  properties  JSON
)
PARTITION BY DATE(event_ts)           -- one
partition per day
CLUSTER BY user_id, event_type       -- up to 4
columns, sorted within each partition
OPTIONS (
  partition_expiration_days = 730,    -- drop
  partitions older than 2 years
  require_partition_filter = TRUE     -- reject
  queries without a partition filter
);

```

Partitioning type	Syntax	Use for
Time-unit column	<code>PARTITION BY DATE(ts) / TIMESTAMP_TRUNC(ts, HOUR)</code>	Event and transaction data (most common)
Ingestion time	<code>PARTITION BY _PARTITIONDATE</code>	Data without a reliable event timestamp
Integer range	<code>PARTITION BY RANGE_BUCKET(customer_id, GENERATE_ARRAY(0, 10000000, 10000))</code>	Tables filtered by numeric ranges

- **Partitioning** prunes whole partitions and makes bytes-scanned estimates accurate before a query runs
- **Clustering** sorts data within partitions so filters and aggregations on clustered columns read fewer blocks; BigQuery re-clusters automatically at no charge
- Keep partitions reasonably large (avoid hourly partitions on small tables); there is a per-table partition limit

DML and MERGE

```

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

```

- DML is billed by bytes scanned (on-demand) like queries — filter on the partition column in the `ON / WHERE` clause
- Prefer batching changes into periodic `MERGE`s over many small single-row updates
- Multi-statement transactions are supported: `BEGIN TRANSACTION; ... COMMIT TRANSACTION;`

Python Client

```

from google.cloud import bigquery

client = bigquery.Client(project="my-project") #
uses Application Default Credentials

job_config = bigquery.QueryJobConfig(
    query_parameters=
    [bigquery.ScalarQueryParameter("run_date", "DATE", "2024-
03-15")],
    maximum_bytes_billed=50 * 1024**3, #
    fail instead of scanning more than 50 GiB
)
sql = """
    SELECT customer_id, SUM(amount) AS revenue
    FROM `my-project.analytics.orders`
    WHERE order_date = @run_date
    GROUP BY customer_id
"""
df = client.query(sql,
job_config=job_config).to_dataframe()

```

```
# Load a DataFrame into a table (replace the day's
partition)
load_config =
bigquery.LoadJobConfig(write_disposition="WRITE_TRUNCATE")
client.load_table_from_dataframe(
    df, "my-project.analytics.daily_revenue$20240315",
    job_config=load_config
).result()
```

The `$YYYYMMDD` partition decorator with `WRITE_TRUNCATE` replaces a single partition — an idempotent daily load.

Pricing Models and Cost Control

Model	Billed on	Good for
On-demand	Bytes processed per query (first TiB per month free)	Unpredictable or light workloads
Capacity (editions: Standard, Enterprise, Enterprise Plus)	Slot-hours, with autoscaling and optional commitments	Steady, heavy workloads; predictable spend
Storage	Active vs long-term (not modified for 90 days, cheaper); logical or physical (compressed) billing	Every table

Prices vary by region and change over time — check the current pricing page.

Cost controls - `require_partition_filter = TRUE` on large partitioned tables - `maximum_bytes_billed` on queries and jobs; custom quotas per user or project - Select only needed columns — `SELECT *` reads every column - Materialized views or summary tables for dashboards - Monitor spend with `INFORMATION_SCHEMA.JOBS` :

```
SELECT user_email,
       COUNT(*) AS
queries,
       ROUND(SUM(total_bytes_billed) / POW(1024, 4), 2) AS
tib_billed
FROM `region-eu`.INFORMATION_SCHEMA.JOBS_BY_PROJECT
WHERE creation_time >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(),
INTERVAL 7 DAY)
```

```
AND job_type = "QUERY"
GROUP BY user_email
ORDER BY tib_billed DESC;
```

Performance

- **Read less:** filter on partition and clustering columns; select only needed columns
- **Filter and aggregate early**, before joins; put the largest table first in joins so smaller tables can be broadcast
- **Avoid** `ORDER BY` without `LIMIT` on large results, self-joins that could be window functions, and JavaScript UDFs in hot paths
- **Pre-compute** with materialized views (incrementally maintained) or scheduled summary tables
- **BI Engine** caches data in memory for sub-second dashboard queries
- **Search indexes** speed up needle-in-haystack lookups on text/log columns
- **Inspect** the query execution graph (stages, slot time, bytes shuffled, spilled) to find bottlenecks

Time Travel, Snapshots, and Clones

```
-- Query a table as it was an hour ago (time travel window: 2-7 days, default 7)
SELECT * FROM analytics.orders
FOR SYSTEM_TIME AS OF TIMESTAMP_SUB(CURRENT_TIMESTAMP(),
INTERVAL 1 HOUR);

-- Restore after a bad write
CREATE OR REPLACE TABLE analytics.orders AS
SELECT * FROM analytics.orders
FOR SYSTEM_TIME AS OF TIMESTAMP("2024-03-15 06:00:00+00");

-- Read-only snapshot (pay only for changed data)
CREATE SNAPSHOT TABLE backups.orders_20240315 CLONE
analytics.orders;
```

```
-- Writable, zero-copy clone for development
CREATE TABLE dev.orders CLONE analytics.orders;
```

After time travel expires, a 7-day **fail-safe** period allows recovery only through Google Cloud support.

Security and Governance

Control	Mechanism
Dataset / table access	IAM roles (<code>roles/bigquery.dataViewer</code> , <code>dataEditor</code> , <code>jobUser</code> , ...)
Share query results, not base tables	Authorized views and authorized datasets
Row-level security	Row access policies
Column-level security	Policy tags (Data Catalog taxonomies) + dynamic data masking
Encryption	Google-managed by default; customer-managed keys (CMEK) available
Network perimeter	VPC Service Controls
Audit	Cloud Audit Logs (admin activity and data access)

```
CREATE ROW ACCESS POLICY emea_only
ON analytics.orders
GRANT TO ("group:emea-analysts@example.com")
FILTER USING (region = "EMEA");
```

Common Pitfalls

Pitfall	Symptom	Fix
<code>SELECT *</code> on wide tables	Large bills for small questions	Select only needed columns; <code>LIMIT</code> does not reduce bytes scanned

Pitfall	Symptom	Fix
Queries without a partition filter	Full-table scans	<code>require_partition_filter</code> ; filter directly on the partition column
Wrapping the partition column in a function	No pruning	Filter the raw column (<code>event_ts >= ...</code>) or partition by the expression you filter on
Many single-row DML statements	Slow pipelines, quota errors, high cost	Batch changes and <code>MERGE</code> periodically
Datasets in different locations	Joins fail	Choose locations deliberately; keep related datasets together
Legacy streaming inserts for high-volume ingestion	Higher cost, weaker guarantees	Storage Write API
Over-partitioning small tables	Partition limits and overhead	Partition by day/month only when tables are large; cluster instead
No cost guardrails	One exploratory query costs more than a month of pipelines	<code>maximum_bytes_billed</code> , quotas, dry runs, spend monitoring
Sharded tables (<code>events_20240315</code> , ...)	Slow wildcard queries, schema drift	One partitioned table

Cheat Sheet

Task	Command / SQL
Estimate cost	<code>bq query --use_legacy_sql=false --dry_run 'SELECT ...'</code>
Run a query	<code>bq query --use_legacy_sql=false 'SELECT ...'</code>
Show schema	<code>bq show --schema --format=prettyjson project:dataset.table</code>
Load files	<code>bq load --source_format=PARQUET dataset.table "gs://bucket/path/*.parquet"</code>

Task	Command / SQL
Replace one partition	Write to <code>table\$YYYYMMDD</code> with <code>WRITE_TRUNCATE</code>
Partition + cluster	<code>PARTITION BY DATE(ts) CLUSTER BY a, b</code>
Explode array	<code>FROM t, UNNEST(t.items) AS item</code>
Safe math	<code>SAFE_DIVIDE(a, b) * SAFE_CAST(x AS INT64)</code>
Latest per key	<code>QUALIFY ROW_NUMBER() OVER (PARTITION BY k ORDER BY ts DESC) = 1</code>
Time travel	<code>FOR SYSTEM_TIME AS OF TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 HOUR)</code>
Dev copy	<code>CREATE TABLE dev.t CLONE prod.t</code>
Spend by user	<code>`region-xx`.INFORMATION_SCHEMA.JOBS_BY_PROJECT → total_bytes_billed</code>

Transformation tools that target BigQuery: SQL in scheduled queries, Dataform (native), dbt, SQLMesh

Interview Questions

Q: How is BigQuery's architecture different from a traditional warehouse? A: It is serverless and separates storage from compute. Data lives in Colossus in a columnar format; queries run on Dremel, which splits work across many slots (units of compute) connected by a fast network and a distributed shuffle. There are no clusters to size or indexes to tune — capacity comes from a shared pool (on-demand) or from reserved, autoscaling slots (editions).

Q: What is the difference between partitioning and clustering in BigQuery? A: Partitioning splits a table into segments (usually by day) so queries filtering on the partition column skip whole partitions, and the bytes to be scanned are known before the query runs. Clustering sorts data within each partition by up to four columns, so filters and aggregations on those columns read fewer storage blocks; its savings are only known after execution. Use both: partition by date, cluster by the most common secondary filters.

Q: How would you control BigQuery costs for a large analytics team? A: Enforce partition filters on large tables, set `maximum_bytes_billed` and per-user or per-project quotas, and teach column selection (no `SELECT *`; `LIMIT` doesn't reduce cost). Pre-aggregate for

dashboards with materialized views or BI Engine. Monitor `INFORMATION_SCHEMA.JOBS` for the most expensive users and queries. For steady heavy usage, compare on-demand with capacity pricing (editions with autoscaling and commitments), and separate workloads with reservations.

Q: How do you load data into BigQuery idempotently? A: Load each run into a specific partition with `WRITE_TRUNCATE` (using the `table$YYYYMMDD` decorator or a `MERGE` scoped to that partition), so reruns replace rather than duplicate data. For upserts, stage the batch and `MERGE` on keys. For streaming, use the Storage Write API with committed streams and offsets for exactly-once writes, or deduplicate by event ID downstream.

Q: When would you use nested and repeated fields instead of separate tables? A: When child records are always accessed with their parent — like order line items or event properties. Storing them as `ARRAY<STRUCT<...>>` keeps them co-located, avoids large joins, and reduces shuffle. Use separate tables when the child data is large, queried independently, or updated on its own.

Further Reading

- [BigQuery documentation](https://cloud.google.com/bigquery/docs) (<https://cloud.google.com/bigquery/docs>)
- [Partitioned tables](https://cloud.google.com/bigquery/docs/partitioned-tables) (<https://cloud.google.com/bigquery/docs/partitioned-tables>) and [clustered tables](https://cloud.google.com/bigquery/docs/clustered-tables) (<https://cloud.google.com/bigquery/docs/clustered-tables>)
- [Optimize query computation](https://cloud.google.com/bigquery/docs/best-practices-performance-compute) (<https://cloud.google.com/bigquery/docs/best-practices-performance-compute>)
- [BigQuery pricing](https://cloud.google.com/bigquery/pricing) (<https://cloud.google.com/bigquery/pricing>)
- *Google BigQuery: The Definitive Guide* — Valliappa Lakshmanan & Jordan Tigani (O'Reilly)

Previous: [Snowflake](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/>) • **Next:** [Amazon Redshift](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/) (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/>) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (<https://sarangambekar1997.github.io/data-engineering-handbook/>)

 September 29, 2026