

Passing Arguments in Glue



In AWS Glue, you can **pass arguments to a Glue job** in several ways depending on how you trigger the job (manually, via the AWS Console, AWS CLI, boto3, or from a trigger/another job). These arguments can be accessed in your PySpark script using ``sys.argv`` or ``getResolvedOptions``.

✅ 1. Defining Arguments in the Console (Manual Run)

When you start a job manually via the **AWS Glue Console**:

- Go to the Glue Job.
- Click **Run job**.
- Under **Job parameters - optional**, add parameters in the form:

```
--key1 value1 --key2 value2
```

✅ 2. Accessing Arguments in the Glue Script

Use the ``getResolvedOptions`` method from ``awsglue.utils``:

```
import sys
from awsglue.utils import getResolvedOptions

# Define expected parameters
args = getResolvedOptions(sys.argv, ['JOB_NAME', 'input_path', 'output_path'])
```

```
input_path = args['input_path']
output_path = args['output_path']
```

✅ 3. Passing Arguments via AWS CLI

```
aws glue start-job-run \
  --job-name your_job_name \
  --arguments '{"--input_path":"s3://my-bucket/input/","--output_path":"s3://my-bucket/output/"}'
```

✅ 4. Passing Arguments via boto3 (Python SDK)

```
import boto3

client = boto3.client('glue')

response = client.start_job_run(
    JobName='your_job_name',
    Arguments={
        '--input_path': 's3://my-bucket/input/',
        '--output_path': 's3://my-bucket/output/'
    }
)
```

✅ 5. Common Glue Reserved Arguments

Some AWS Glue jobs require or automatically handle reserved arguments:

Key	Description
<code>--JOB_NAME`</code>	Required for all jobs
<code>--TempDir`</code>	Temp directory for dynamic frame operations
<code>--enable-continuous-cloudwatch-log`</code>	Enables streaming CloudWatch logs
<code>--additional-python-modules`</code>	Add extra PyPI modules

✓ 6. Example: Full PySpark Glue Script with Arguments

```
import sys
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions

# Get arguments
args = getResolvedOptions(sys.argv, ['JOB_NAME', 'input_path', 'output_path'])

# Initialize contexts
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session

# Use the arguments
df = spark.read.json(args['input_path'])
df.show()

df.write.parquet(args['output_path'])
```

Let me know if you want a template job or explanation for Glue Studio, Glue triggers, or Glue workflows too.



To **pass arguments to an AWS Glue job from a Lambda function**, you use `boto3`'s `start_job_run` **API** with the `Arguments` parameter.

✅ Step-by-Step: Start Glue Job from Lambda with Arguments

1. IAM Role Permissions

Ensure the **Lambda's execution role** has permission to start Glue jobs:

```
{
  "Effect": "Allow",
  "Action": "glue:StartJobRun",
  "Resource": "arn:aws:glue:your-region:your-account-id:job/your-job-name"
}
```

2. Lambda Function (Python) to Start Glue Job

```
import boto3

def lambda_handler(event, context):
    glue = boto3.client('glue')

    response = glue.start_job_run(
        JobName='your_glue_job_name',
        Arguments={
            '--input_path': 's3://your-bucket/input/',
            '--output_path': 's3://your-bucket/output/',
            '--run_date': event.get('run_date', '2025-06-26') # dynamic
        }
    )
```

```
argument
    }
)

return {
    'statusCode': 200,
    'body': f"Started Glue job with run ID: {response['JobRunId']}"
}
```

3. Glue Script to Receive Arguments

In your Glue PySpark script:

```
import sys
from awsglue.utils import getResolvedOptions

args = getResolvedOptions(sys.argv, ['JOB_NAME', 'input_path', 'output_path',
    'run_date'])

input_path = args['input_path']
output_path = args['output_path']
run_date = args['run_date']
```

4. Example Event (for Testing Lambda Manually)

```
{
    "run_date": "2025-06-26"
}
```

Let me know if:

- You want to trigger the Lambda from S3, EventBridge, or API Gateway.
- Your Glue job is in Glue Studio/Glue 2.0.
- You need to pass dynamic arguments like file name or timestamp.



In **Amazon Redshift Serverless, 1 RPU (Redshift Processing Unit)** is a unit of **compute power** and memory. While AWS does **not provide an exact equivalence** (like number of vCPUs or GB of RAM per RPU), here's what you should know:

✅ Approximate Characteristics of 1 RPU

Resource	Value (Approx.)
vCPUs	~1-2 vCPUs (AWS does not disclose exact count)
Memory (RAM)	~2-4 GiB RAM
Pricing (us-east-1)	\$0.375 per RPU-hour
Concurrency	Auto-handled with memory scaling

"Think of 1 RPU as a unit of abstract compute (similar to 1 DWU in Synapse or 1 CU in BigQuery)."

🧠 Analogy

- 8 RPU's can roughly match a **dc2.large** Redshift cluster (~2 nodes).
 - 32-64 RPU's are typically used for production-sized analytics with complex queries.
-

How to Choose RPU Size?

Workload Type	Suggested RPU Range
Development / Test	8-16 RPU's
Medium ETL / BI workloads	32-64 RPU's
Heavy analytical workloads	128-512+ RPU's

Tip:

Redshift Serverless **automatically scales** up to the maximum RPU limit you define (say, 64 or 128). You're billed **only for what's used per second**, not for the max limit.

Would you like help estimating RPU usage for a specific workload or converting a provisioned cluster to RPU's?



To **write data from AWS Glue PySpark to Amazon Redshift**, you can use the ``glueContext.write_dynamic_frame.from_jdbc_conf()`` method, which simplifies writing data to Redshift via JDBC.

✓ Pre-requisites

1. **IAM Role:** Glue job's IAM role must have access to Redshift and S3.
 2. **Redshift Connection:** Create a Redshift connection in AWS Glue (JDBC with username/password or IAM).
 3. **VPC Configuration:** Glue job and Redshift cluster must be in the same VPC/subnet (for provisioned Redshift).
 4. **TempDir:** Provide an S3 temporary directory for staging data.
-

🔧 Sample Code: Glue PySpark → Redshift

```
import sys
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.dynamicframe import DynamicFrame

args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session

# Sample data
df = spark.createDataFrame([
    (1, "Alice"),
    (2, "Bob")
], ["id", "name"])

dyf = DynamicFrame.fromDF(df, glueContext, "dyf")

# Write to Redshift
```



```

glueContext.write_dynamic_frame.from_jdbc_conf(
    frame=dyf,
    catalog_connection="redshift-conn", # Name of Glue connection
    connection_options={
        "database": "dev",
        "dbtable": "public.sample_table",
        "aws_iam_role": "arn:aws:iam::123456789012:role/AWSGlueServiceRole",
        "preactions": "DROP TABLE IF EXISTS public.sample_table;", # optional
        "postactions": "ANALYZE public.sample_table;", # optional
    },
    redshift_tmp_dir="s3://your-temp-bucket/glue/temp/"
)

```

Explanation

Part	Description
<code>`catalog_connection`</code>	Glue Connection to Redshift
<code>`dbtable`</code>	Target Redshift table (must exist unless <code>`preactions`</code> creates it)
<code>`aws_iam_role`</code>	IAM role with Redshift + S3 access
<code>`redshift_tmp_dir`</code>	Temporary staging S3 location (used internally by Glue to load data)

Alternative: Spark JDBC (Advanced)

If you don't want to use Glue DynamicFrame API, you can use Spark's JDBC like this:

```

df.write \
    .format("jdbc") \
    .option("url", "jdbc:redshift://<endpoint>:5439/dev") \

```

```
.option("dbtable", "public.sample_table") \  
.option("user", "<username>") \  
.option("password", "<password>") \  
.option("tempdir", "s3://your-temp-bucket/glue/temp/") \  
.option("aws_iam_role", "arn:aws:iam:...") \  
.save()
```

“But DynamicFrame-based method is more optimized and recommended in Glue.”

Would you like a reusable template to write PySpark DataFrames to Redshift dynamically with table/connection as parameters?



Here's a clear explanation of **distribution keys**, **sort keys**, and **materialized views** in **Amazon Redshift**, along with **examples** for each.

◆ 1. Distribution Keys (DISTKEY)

📌 What:

- Redshift is **distributed**, meaning it splits data across multiple nodes.
- `DISTKEY` tells Redshift **how to distribute rows** of a table across nodes.
- Choosing the right `DISTKEY` can **reduce data movement** and improve **join performance**.

✅ Use When:

- You join large tables frequently on a common column.
- Co-locating data on the same node reduces shuffling.

Example:

```
CREATE TABLE orders (  
  order_id      INT,  
  customer_id   INT,  
  amount        DECIMAL(10,2)  
)  
DISTKEY (customer_id);
```

- If you also have a `customers` table with `DISTKEY (customer_id)`, then joins on `customer_id` will be **faster** because matching rows are on the **same node**.

2. Sort Keys (SORTKEY)

What:

- Redshift stores data on disk **sorted** by the column(s) you define as `SORTKEY`.
- Helps **optimize range queries**, filtering, and **query pruning**.
- Can be **single-column** or **compound/interleaved** sort keys.

Use When:

- You often filter, group, or order by a particular column (e.g., `date`, `id`).

Example:

```
CREATE TABLE events (  
  event_id    INT,  
  event_type  VARCHAR,  
  event_date  DATE  
)  
SORTKEY (event_date);
```

- Queries like `WHERE event_date BETWEEN '2024-01-01' AND '2024-01-31'` are much faster.

Compound vs Interleaved Sort Keys

Type	Best For
Compound	Left-most key always used in WHERE clause
Interleaved	Multiple columns used equally in filtering

3. Materialized Views

What:

- A **precomputed result set** stored on disk.
- Unlike normal views, they **store data** and can be **refreshed** manually or automatically.
- Speeds up complex queries like joins and aggregations.

✅ Use When:

- You frequently run heavy queries with joins or aggregations on large data.

🧠 Example:

```
CREATE MATERIALIZED VIEW total_sales_mv AS
SELECT customer_id, SUM(amount) AS total_spent
FROM orders
GROUP BY customer_id;
```

Then:

```
SELECT * FROM total_sales_mv WHERE total_spent > 1000;
```

- Runs **much faster** than querying the base `orders` table each time.
- You can **refresh** it using:

```
REFRESH MATERIALIZED VIEW total_sales_mv;
```

Or configure it to refresh **automatically**.

🔄 Summary Table

Feature	What it Does	Best Use
<code>DISTKEY</code>	Controls row distribution across nodes	Large joins on common keys
<code>SORTKEY</code>	Controls row sort order on disk	Filtering, range scans
<code>MATERIALIZED VIEW</code>	Stores precomputed query results	Speeding up heavy queries

Would you like a diagram or cost/performance tips when choosing keys and materialized views?



To **COPY data from Amazon S3 to Amazon Redshift**, you use the `COPY` command in SQL. This is the **fastest and most efficient** way to load bulk data into Redshift.

✓ 1. Basic Syntax of COPY

```
COPY schema.table_name
FROM 's3://your-bucket/path/'
IAM_ROLE 'arn:aws:iam::account-id:role/YourRedshiftRole'
FORMAT AS PARQUET;
```

📦 2. Common File Formats Supported

Format	COPY Syntax
CSV	<code>`FORMAT AS CSV`</code>
JSON	<code>`FORMAT AS JSON 'auto'`</code> or provide a JSONPaths file
PARQUET	<code>`FORMAT AS PARQUET`</code>
AVRO	<code>`FORMAT AS AVRO`</code>
DELIMITED	<code>`DELIMITER ','`</code> (for CSV-like files)

3. Examples

Example 1: Load CSV

```
COPY public.orders
FROM 's3://my-bucket/orders/'
IAM_ROLE 'arn:aws:iam::123456789012:role/MyRedshiftRole'
FORMAT AS CSV
IGNOREHEADER 1;
```

Example 2: Load Parquet

```
COPY public.sales
FROM 's3://my-bucket/sales-data/'
IAM_ROLE 'arn:aws:iam::123456789012:role/MyRedshiftRole'
FORMAT AS PARQUET;
```

Example 3: Load JSON

```
COPY public.events
FROM 's3://my-bucket/events/'
```

```
IAM_ROLE 'arn:aws:iam::123456789012:role/MyRedshiftRole'  
FORMAT AS JSON 'auto';
```

4. IAM Role Permissions

Make sure the IAM role used:

- Has access to S3 (`s3:GetObject``)
- Is **attached to Redshift** (visible in console)

5. Optional Parameters

Option	Description
<code>`IGNOREHEADER 1`</code>	Skip header row (CSV)
<code>`DELIMITER ','`</code>	Specify delimiter
<code>`TIMEFORMAT`</code>	Parse datetime formats
<code>`MAXERROR`</code>	Tolerate bad records
<code>`TRUNCATECOLUMNS`</code>	Truncate data if wider than column

Best Practices

- Split large files into **multiple smaller files** for parallel loading.

- Use **gzip** or **snappy** compression.
 - Use **Parquet** for better performance and smaller size.
 - Always load into **staging tables** first.
-

Refresh Materialized View (if needed)

If your data impacts materialized views, remember to:

```
REFRESH MATERIALIZED VIEW your_mv_name;
```

Would you like a Python boto3 example to trigger COPY from a Lambda or Glue job?



Amazon Redshift handles data distribution by **spreading table data across compute nodes** in a cluster to **maximize parallel processing and performance**. Choosing the right **distribution style** is critical for optimizing **joins, aggregations, and query performance**.

How Redshift Handles Data Distribution

- Redshift divides a table's data into **slices**, with each slice assigned to a CPU core.
- Rows are **assigned to slices** based on the **distribution style** of the table.

- When two tables are joined, performance is best if matching rows are **on the same slice** (i.e., no data shuffling between nodes).

Distribution Styles in Redshift

Distribution Style	Description	Best Use Case
KEY	Distributes rows based on a hash of a column	Large fact tables joined on a key
ALL	Replicates entire table to every node	Small dimension tables used in joins
EVEN (or ROUND ROBIN)	Distributes rows evenly across slices	Unrelated or staging tables
AUTO	Redshift chooses based on size and query patterns	Recommended unless you have a specific reason

KEY Distribution

- Uses a **hash of a column** value to determine the slice.
- Rows with the same key go to the **same slice**.
- Improves join performance between large tables.

```
CREATE TABLE orders (  
  order_id INT,  
  customer_id INT,  
  amount DECIMAL(10,2)
```

```
)  
DISTSTYLE KEY  
DISTKEY (customer_id);
```

→ Join `orders` with a `customers` table also distributed on `customer_id` for efficient joins.

2 ALL Distribution

- Table is **fully copied** to each node.
- Great for **small, frequently joined dimension tables**.

```
CREATE TABLE countries (  
    country_code CHAR(2),  
    country_name VARCHAR(50)  
)  
DISTSTYLE ALL;
```

→ Reduces join overhead by avoiding redistribution.

3 EVEN (Round-Robin) Distribution

- Rows are distributed **evenly**, regardless of values.
- Useful for **staging tables** or when no good join key exists.

```
CREATE TABLE logs (  
    log_id INT,  
    message TEXT
```

```
)  
DISTSTYLE EVEN;
```

→ No bias in data distribution, but may require redistribution on joins.

4 AUTO Distribution (default)

- Redshift uses internal logic to **automatically choose** the best style.
- Starts with **ALL** if the table is small, or **EVEN/KEY** if it grows.

```
CREATE TABLE sales (  
    sale_id INT,  
    product_id INT,  
    price DECIMAL(10,2)  
);  
  
-- No DISTSTYLE = AUTO (default)
```

Summary of When to Use What

Scenario	Best Distribution Style
Fact tables with large joins	<code>`DISTSTYLE KEY`</code>
Small dimension tables	<code>`DISTSTYLE ALL`</code>
Staging/temp tables	<code>`DISTSTYLE EVEN`</code>
Unsure or dynamic growth	<code>`DISTSTYLE AUTO`</code>

Would you like help choosing the right distribution style for your specific schema or workload?



◆ What Are Sort Keys in Amazon Redshift?

Sort keys in Redshift define the **physical order** of data on disk. They significantly affect **query performance**, especially for **filtering**, **range scans**, and **joins**.

✅ Key Benefits of Sort Keys

Feature	Benefit
📦 Data Pruning	Redshift can skip entire blocks of data (blocks = 1 MB each)
🚀 Faster WHERE clauses	Efficient access to relevant rows
🧠 Improves joins and aggregations	If sorted on join/filter columns

🔄 Types of Sort Keys

Type	Description
Compound Sort Key	Data is sorted in the order of listed columns (left to right). Most effective when queries use left-most columns.

Type	Description
Interleaved Sort Key	Gives equal weight to all columns. Better when queries use multiple sort key columns independently .

Example: Compound Sort Key

```
CREATE TABLE orders (  
  order_id INT,  
  customer_id INT,  
  order_date DATE,  
  total_amount DECIMAL(10,2)  
)  
SORTKEY (order_date);
```

✅ Efficient for queries like:

```
SELECT * FROM orders  
WHERE order_date BETWEEN '2024-01-01' AND '2024-01-31';
```

Example: Interleaved Sort Key

```
CREATE TABLE events (  
  event_id INT,  
  user_id INT,  
  event_type VARCHAR,  
  event_date TIMESTAMP  
)  
INTERLEAVED SORTKEY (user_id, event_type, event_date);
```

✓ Good for queries filtering on **any** of these columns:

```
SELECT * FROM events WHERE user_id = 123;  
SELECT * FROM events WHERE event_type = 'click';
```

! Impact on Performance

🚀 Faster Queries:

- Sort keys allow **block-level pruning**.
- Reduces data scanned = lower latency and cost.

🐌 Slower Writes:

- Insert performance can degrade if sort key order is not preserved.
- Especially with **interleaved** sort keys.

📌 Best Practices

Tip	Explanation
Choose compound sort key for time-series data	e.g. `event_date`
Use interleaved for multi-dimensional queries	e.g. filtering on multiple unrelated columns
Don't overuse sort keys	Adds complexity and slows inserts
Monitor with `SVL_QLOG`, `STL_SCAN`	See if pruning is helping

Would you like help picking sort keys for your actual Redshift tables or workload?







What is VACUUM in Amazon Redshift?

The `VACUUM` command in Redshift **reclaims space**, **re-sorts rows**, and **improves query performance** by cleaning up tables after operations like `DELETE`, `UPDATE`, or `INSERT`.

Redshift uses a **columnar, append-only** storage model. When you delete or update data, the old rows aren't immediately removed — they're just marked for deletion. This can lead to **bloat**, slower queries, and inefficient storage.

Purposes of VACUUM

Purpose	Description
 Reclaim space	Frees disk space taken by deleted rows
 Re-sort data	Restores sort key order for faster queries
 Optimize performance	Reduces I/O by helping block pruning
 Defragment storage	Organizes table blocks for efficient access

When to Use VACUUM

Scenario	Action Needed
Large DELETE/UPDATE	Run <code>`VACUUM DELETE ONLY`</code> or <code>`FULL`</code>
Insertions break sort order	Use <code>`VACUUM REINDEX`</code> or <code>`FULL`</code>
Sort key performance degrading	Use <code>`VACUUM SORT ONLY`</code>
After ELT pipelines/bulk loads	Consider <code>`VACUUM`</code> (esp. if out of order)

VACUUM Syntax and Types

```
VACUUM [FULL | SORT ONLY | DELETE ONLY | REINDEX] [table_name];
```

Type	Use Case
<code>`FULL`</code>	Reclaims space + re-sorts + re-indexes (slowest but most complete)
<code>`SORT ONLY`</code>	Only re-sorts based on sort key (no space reclaim)
<code>`DELETE ONLY`</code>	Only reclaims deleted space (no re-sort)
<code>`REINDEX`</code>	Re-indexes interleaved sort keys

Example

```
-- Reclaim space after deletions
VACUUM DELETE ONLY my_table;

-- Fully vacuum and re-sort table
```

```
VACUUM FULL my_table;
```

```
-- Re-sort based on sort key
```

```
VACUUM SORT ONLY my_table;
```

Notes and Best Practices

- Redshift **autovacuum** runs in the background but **isn't immediate** or aggressive.
- You can **monitor table bloat** using:

```
SELECT * FROM SVV_TABLE_INFO WHERE unsorted > 20;
```

- Use **VACUUM during off-peak hours** — it uses resources and locks tables.
- For **interleaved sort keys**, use ``VACUUM REINDEX`` regularly.

Summary

Feature	Value
Purpose	Clean up deleted/updated data, re-sort rows
Improves	Storage efficiency, query performance
When to use	After large deletes/updates or insert-heavy loads
Type to use	Depends on issue: <code>`DELETE ONLY`</code> , <code>`SORT ONLY`</code> , or <code>`FULL`</code>

Would you like a script to automate VACUUM based on table health or size?



Amazon Redshift uses **columnar storage** and applies **column-level compression** (also called **encodings**) to reduce storage size and improve query performance.

✅ Why Compression Matters

- Saves **disk space** (cost-effective)
 - Improves **I/O performance**
 - Reduces **query latency**
 - Redshift **automatically decompresses** data during query execution
-

1234 Types of Column Compression Encodings in Redshift

Here's a summary of the most common Redshift compression types:

Encoding Type	Best For	Description
RAW	No compression	Used when compression isn't possible or needed
BYTEDICT	Low-cardinality string/integer	Maps unique values to small integer codes

Encoding Type	Best For	Description
RUNLENGTH	Columns with repeated values	Great for sorted data or long repeated sequences
TEXT255 / TEXT32K	String columns up to 255 / 32K	Compresses common prefixes in strings
DELTA	Sequential numeric data	Stores differences between consecutive values
DELTA32K	Wider range of delta compression	Like DELTA but supports larger differences
LZO	General-purpose compression	Fast compression/decompression
ZSTD	High compression ratio	Slower to compress, but very space efficient (cost-effective)

Example: Creating Table with Column Compression

```
CREATE TABLE sales (
  sale_id          INT ENCODE DELTA,
  customer_id      INT ENCODE BYTEDICT,
  sale_amount      DECIMAL(10,2) ENCODE ZSTD,
  sale_date        DATE ENCODE DELTA,
  status           VARCHAR(10) ENCODE TEXT255
);
```

“Redshift automatically applies compression during a `COPY` if you use the `COMPUPDATE ON` option.”

How to Choose Compression Encoding?

1. Use **ANALYZE COMPRESSION** to let Redshift suggest optimal encodings:

```
ANALYZE COMPRESSION sales;
```

2. It returns a table with columns, suggested encodings, and estimated savings.

Which Compression Is Most Cost-Effective?

Encoding	Compression Ratio	Use Case
ZSTD	★★★★★ (best)	High compression, good for large, static data
DELTA	★★★☆☆	Sequential IDs, timestamps
BYTEDICT	★★★☆☆	Low-cardinality columns (e.g., country codes)
RUNLENGTH	★★★★★ (if applicable)	Best when sorted and values repeat
LZO	★★☆☆☆	Balanced speed and space (legacy)

→ **ZSTD** and **RUNLENGTH** give **highest space savings** = **more cost-effective** in storage-heavy workloads.

Best Practices

- Use ``ANALYZE COMPRESSION`` after loading sample data.
- Prefer **ZSTD** for large string or numeric columns unless high speed is critical.

- If data is sorted or sequential → use ``DELTA`` or ``RUNLENGTH``.

Would you like help optimizing compression for a specific table or comparing space savings before/after?



To **update rows in Amazon Redshift based on keys**, you use the standard SQL ``UPDATE ... FROM`` syntax.

✅ Basic Syntax for UPDATE with JOIN (key-based update)

```
UPDATE target_table
SET column1 = source.column1,
    column2 = source.column2
FROM source_table AS source
WHERE target_table.key = source.key;
```

🧠 Example

Let's say you have:

- ``orders``: target table to be updated
- ``latest_orders``: source table with new values

```
UPDATE orders
SET amount = latest.amount,
    status = latest.status
FROM latest_orders AS latest
WHERE orders.order_id = latest.order_id;
```

👉 This updates `orders.amount` and `orders.status` where `order_id` matches in both tables.

Update Based on Multiple Keys

```
UPDATE orders
SET amount = latest.amount
FROM latest_orders AS latest
WHERE orders.order_id = latest.order_id
    AND orders.customer_id = latest.customer_id;
```

Notes

- Redshift **does not support** `UPDATE` with subqueries directly — use a `FROM` clause.
 - For **large updates**, consider:
 - Updating in **batches** using a temp staging table
 - Using **UPSERT** via `MERGE` (Redshift supports `MERGE` from 2023)
-



Alternative: MERGE (Redshift-supported)

```
MERGE INTO orders AS tgt
USING latest_orders AS src
ON tgt.order_id = src.order_id
WHEN MATCHED THEN
    UPDATE SET amount = src.amount, status = src.status;
```

Would you like a script for performing batch updates or a pattern for slowly changing dimensions (SCD)?



To **update matched records and insert unmatched records** in Amazon Redshift (i.e., perform an **UPSERT**), you should use the `MERGE` command — introduced in Redshift **from April 2023**.



Redshift `MERGE` Syntax

```
MERGE INTO target_table AS tgt
USING source_table AS src
ON tgt.key_column = src.key_column
WHEN MATCHED THEN
    UPDATE SET
        column1 = src.column1,
        column2 = src.column2
WHEN NOT MATCHED THEN
    INSERT (key_column, column1, column2)
    VALUES (src.key_column, src.column1, src.column2);
```



Example: Upsert into `orders` from `latest_orders`

```
MERGE INTO orders AS tgt
USING latest_orders AS src
ON tgt.order_id = src.order_id
WHEN MATCHED THEN
    UPDATE SET
        amount = src.amount,
        status = src.status
WHEN NOT MATCHED THEN
    INSERT (order_id, customer_id, amount, status)
    VALUES (src.order_id, src.customer_id, src.amount, src.status);
```

Use Case

Condition	Action
`order_id` exists	`UPDATE` values
`order_id` not found	`INSERT` new row

Requirements and Tips

- Requires Redshift version **April 2023 or newer**.
- You must provide **all required columns** in `INSERT`.

- If ``MERGE`` is not available (older version), use a workaround:

1. Run ``UPDATE``
2. Run ``INSERT`` with ``NOT EXISTS`` or ``LEFT JOIN`` manually

Alternative (for older Redshift versions)

```
-- 1. UPDATE existing
UPDATE orders
SET amount = latest.amount,
    status = latest.status
FROM latest_orders AS latest
WHERE orders.order_id = latest.order_id;

-- 2. INSERT new
INSERT INTO orders (order_id, customer_id, amount, status)
SELECT latest.order_id, latest.customer_id, latest.amount, latest.status
FROM latest_orders AS latest
LEFT JOIN orders ON latest.order_id = orders.order_id
WHERE orders.order_id IS NULL;
```

Would you like a reusable stored procedure or Glue-based solution to do this dynamically across tables?



To **upsert (update + insert unmatched records) data from S3 to Amazon Redshift**, follow this process:

✅ Step-by-Step UPSERT from S3 to Redshift

🔧 Assumptions

- You have a **source file** (CSV/Parquet/etc.) in **S3**.
- You want to **merge it into a target table** in Redshift.
- Your **Redshift cluster** has access to S3 via an **IAM Role**.

🔄 Steps Overview

1. **Create a staging table** (to hold new data from S3).
2. **COPY data from S3** into the staging table.
3. Use **MERGE** to upsert data into the target table.
4. (Optional) **DROP or TRUNCATE** the staging table.

📁 Step 1: Create Staging Table (same structure as target)

```
CREATE TEMP TABLE staging_orders (  
    order_id      INT,  
    customer_id   INT,  
    amount        DECIMAL(10,2),  
    status        VARCHAR(20)  
);
```



Step 2: COPY from S3 to Redshift Staging Table

```
COPY staging_orders
FROM 's3://your-bucket/path/orders.csv'
IAM_ROLE 'arn:aws:iam::123456789012:role/YourRedshiftRole'
FORMAT AS CSV
IGNOREHEADER 1;
```

*"If your file is **Parquet**, use:"*

```
FORMAT AS PARQUET;
```



Step 3: MERGE (UPSERT) into Target Table

```
MERGE INTO orders AS tgt
USING staging_orders AS src
ON tgt.order_id = src.order_id
WHEN MATCHED THEN
    UPDATE SET
        customer_id = src.customer_id,
        amount      = src.amount,
        status      = src.status
WHEN NOT MATCHED THEN
    INSERT (order_id, customer_id, amount, status)
    VALUES (src.order_id, src.customer_id, src.amount, src.status);
```



Step 4: Clean Up (Optional)

```
DROP TABLE staging_orders;
```



✅ Best Practices

Tip	Description
Use <code>`TEMP`</code> table	Automatically drops after session ends
Use <code>`MERGE`</code> for atomicity	One command does update + insert
Compress and partition S3 files	For better COPY performance
Use <code>`COPY`</code> with <code>`COMPUPDATE OFF`</code> if encodings are fixed	Speeds up load

Would you like a fully automated script using Python or AWS Glue to orchestrate this end-to-end?



Yes, you **can schedule a Redshift query to run based on file arrival in S3**, but Redshift itself does **not natively detect S3 file events**. Instead, you can build this using **AWS event-driven services** like **S3 + Lambda + EventBridge + Redshift**.

✅ How to Trigger Redshift Query on S3 File Arrival

🔧 Architecture Overview

1. **S3** → new file is uploaded

2. **S3 Event Notification** triggers
 3. **Lambda Function** executes SQL on Redshift via `boto3`
 4. (Optional) Use **EventBridge** or **Glue workflow** for more complex pipelines
-

Step-by-Step Implementation

1 Enable S3 Event Notification

- Go to your **S3 bucket** → **Properties** → **Event Notifications**
 - Create a new event:
 - **Event type:** `PUT`
 - **Prefix:** folder path (e.g., `incoming/`)
 - **Send to:** Lambda
-

2 Create a Lambda Function to Run Redshift Query

```
import boto3

def lambda_handler(event, context):
    redshift_data = boto3.client('redshift-data')

    response = redshift_data.execute_statement(
        ClusterIdentifier='your-redshift-cluster',
        Database='your_db',
        DbUser='your_user',
        Sql="CALL run_upsert_procedure();", # or your custom SQL
        WorkgroupName='your-rs-serverless-wg', # only for Redshift Serverless
        SecretArn='arn:aws:secretsmanager:region:acct:secret:your-secret' #
```

```
for auth
)

return {
  'statusCode': 200,
  'body': 'Query triggered successfully!'
}
```

3 IAM Permissions for Lambda Role

Lambda's execution role must allow:

```
{
  "Effect": "Allow",
  "Action": [
    "redshift-data:ExecuteStatement",
    "redshift-data:GetStatementResult",
    "secretsmanager:GetSecretValue"
  ],
  "Resource": "*"
}
```

4 (Optional) Use a Stored Procedure in Redshift

Instead of hardcoding SQL in Lambda, use:

```
CREATE OR REPLACE PROCEDURE run_upsert_procedure()
AS $$
BEGIN
  -- Run COPY from S3
  COPY staging_table FROM 's3://your-bucket/folder/'
  IAM_ROLE 'arn:aws:iam::123456789012:role/YourRedshiftRole'
```

```
FORMAT AS PARQUET;

-- Run MERGE/UPSERT
MERGE INTO target_table ...
END;
$$ LANGUAGE plpgsql;
```

Summary

Component	Role
S3 Event	Triggers Lambda when file is uploaded
Lambda	Runs SQL query/procedure in Redshift
Redshift Data API	Executes SQL
Secrets Manager	Stores Redshift credentials securely

Would you like a full CloudFormation or Terraform template to set this up automatically?



Certainly! Here's a **set of interview questions and answers** focused on **Slowly Changing Dimensions (SCD)** — a key concept in Data Warehousing and ETL pipelines, especially relevant to Redshift, Glue, and general Data Engineering roles.

✓ Basic SCD Interview Questions

◆ Q1. What is a Slowly Changing Dimension (SCD)?

Answer:

A Slowly Changing Dimension (SCD) is a dimension in a data warehouse that **changes slowly over time**, rather than frequently. It refers to how we manage and track **historical changes** in dimension data.

◆ Q2. What are the types of SCDs?

Answer:

Type	Description
SCD Type 0	No changes are allowed. Data remains as originally inserted.
SCD Type 1	Overwrite old data with new data. No history kept.
SCD Type 2	Create a new row for each change. Full history preserved.
SCD Type 3	Track limited history by adding extra columns (e.g., `prev_city`).

✓ Scenario-Based Questions

◆ Q3. When would you use SCD Type 1 vs Type 2?

Answer:

- Use **Type 1** when **history is not important**, and you only need the most recent value (e.g., correcting a misspelled name).
 - Use **Type 2** when you need to **track historical changes** (e.g., tracking customer's address change over time).
-

◆ Q4. How do you implement SCD Type 2 in SQL (Redshift or other DBs)?

Answer:

Steps to implement SCD Type 2:

1. Load new data into a **staging table**.
2. Compare with the existing dimension table based on **business key** (e.g., `customer_id`).
3. If a change is detected:
 - **Expire** the current record (set `end_date`).
 - **Insert** a new row with updated values and `start_date`.

Example SQL:

```
-- 1. Expire old record
UPDATE dim_customer
SET end_date = CURRENT_DATE
WHERE customer_id = 'C001'
  AND end_date IS NULL
  AND city <> 'New City';

-- 2. Insert new row
```

```
INSERT INTO dim_customer (customer_id, name, city, start_date, end_date)
VALUES ('C001', 'Alice', 'New City', CURRENT_DATE, NULL);
```

◆ Q5. What columns are typically used in SCD Type 2?

Answer:

Column	Purpose
`surrogate_key`	Unique row identifier (PK)
`business_key`	Natural key (e.g., `customer_id`)
`start_date`	When the record became active
`end_date`	When the record was replaced
`is_current`	Flag to indicate current record (`Y/N`)

✓ Advanced Questions

◆ Q6. How do you detect changes between new and existing records?

Answer:

- Use a **LEFT JOIN** between the staging table and the dimension table on the business key.

- Check for changes using ``COALESCE`` or ``IS DISTINCT FROM``.

Example:

```
SELECT s.*
FROM staging_customer s
LEFT JOIN dim_customer d
  ON s.customer_id = d.customer_id AND d.end_date IS NULL
WHERE s.city IS DISTINCT FROM d.city;
```

◆ Q7. How would you handle SCD in Glue using PySpark?

Answer:

1. Read existing and incoming data.
2. Perform a join on business key.
3. Use ``when/otherwise`` logic to detect changes.
4. For updated rows:
 - Mark current rows as expired (``end_date``).
 - Append new rows with ``start_date``.
5. Union unchanged + new + updated rows → overwrite dimension table.

◆ Q8. What are challenges of SCD Type 2?

Answer:

- **Increased storage** due to historical rows.

- Complexity in **queries** (you must filter on ``end_date IS NULL`` or ``is_current = 'Y'``).
 - Needs careful **join logic** and deduplication.
-

Would you like a full PySpark example of SCD Type 2 using AWS Glue?



Certainly! Here are **interview questions and answers** on **Data Warehousing platforms**, covering concepts across **Amazon Redshift, Snowflake, BigQuery, Azure Synapse**, and general architecture principles.

◆ Basic Data Warehousing Questions

◆ Q1. What is a Data Warehouse?

Answer:

A **data warehouse** is a centralized system designed to store, integrate, and analyze large volumes of **historical and analytical data** from multiple sources for reporting and business intelligence. It supports **OLAP (Online Analytical Processing)**.

◆ Q2. Name some popular data warehousing platforms.

Answer:

- **Amazon Redshift**

- **Google BigQuery**
 - **Snowflake**
 - **Azure Synapse Analytics**
 - **Teradata**
 - **Oracle Exadata**
 - **IBM Netezza**
-

◆ Q3. What are the main differences between OLAP and OLTP?

Feature	OLTP	OLAP
Purpose	Transactional (CRUD)	Analytical (Reporting, BI)
Data Volume	Small, frequent transactions	Large, historical data
Queries	Short and simple	Complex, aggregated
Normalization	Highly normalized	Denormalized/star schema

◆ Platform-Specific Questions

◆ Q4. How is data stored in Amazon Redshift?

Answer:

Redshift stores data in a **columnar format** across multiple compute nodes. Each table is divided into **slices**, and data is distributed using **distribution keys/styles**. It supports **compression**, **sort keys**, and **massively parallel processing (MPP)** for performance.

◆ **Q5. What makes Snowflake architecture unique?**

Answer:

Snowflake uses a **multi-cluster, shared-data architecture**, where:

- Storage and compute are **decoupled**
 - Each virtual warehouse (compute cluster) can scale independently
 - Supports **auto-scaling**, **time travel**, and **zero-copy cloning**
-

◆ **Q6. How does Google BigQuery process queries?**

Answer:

BigQuery is a **serverless, fully-managed** data warehouse that processes queries using **Dremel**, a distributed query engine. It auto-scales and charges based on **bytes scanned**, not compute time. Data is stored in **Colossus** (columnar format).

◆ **Performance & Cost Optimization**

◆ Q7. How do you optimize performance in a data warehouse?

Answer:

- Use **partitioning** and **clustering** (BigQuery, Snowflake)
- Choose optimal **sort keys and dist keys** (Redshift)
- Use **materialized views** for repeated aggregations
- Archive unused data to cold storage
- Apply **compression** to reduce I/O and cost

◆ Q8. Compare Redshift and Snowflake.

Feature	Redshift	Snowflake
Architecture	MPP with local storage	Decoupled compute & storage
Scaling	Manual or Elastic Resize	Auto-scaling, multi-cluster
Storage	Amazon S3 (via Redshift-managed)	Cloud object storage (S3, GCS, Azure)
Pricing	Reserved or On-Demand	Per-second, on-demand compute
Semi-structured	Supported via SUPER type	Native (VARIANT data type)
Performance	Manual tuning (sort, dist keys)	Automated optimization

◆ Q9. How does Azure Synapse differ from Redshift?

Answer:

Feature	Azure Synapse	Amazon Redshift
Integration	Strong with MS services	Strong with AWS ecosystem
Language Support	T-SQL, Spark, Python	SQL, Python via Redshift ML
Compute Model	On-demand or provisioned	Provisioned or Serverless
Use Cases	Data lake + DW hybrid	Traditional DW & analytics

◆ Advanced Questions

◆ Q10. What is ELT and how is it used in cloud data warehouses?

Answer:

ELT (Extract-Load-Transform) is a modern approach where:

1. Data is **extracted** from sources
2. **Loaded** into the data warehouse (raw)
3. **Transformed** using SQL or stored procedures **within** the warehouse

Cloud DWHs like Redshift, Snowflake, and BigQuery are designed for **ELT**, as they have powerful SQL engines and scale compute independently.

◆ Q11. What is a materialized view and how is it different from a normal view?

Answer:

Feature	View	Materialized View
Stored Data	No	Yes (precomputed)
Refresh Needed	No	Yes (``REFRESH`` or auto)
Query Speed	Slower (run-time compute)	Faster (precomputed results)

Materialized views are used to **cache complex queries** for better performance.

◆ Q12. How does cost differ across Redshift, Snowflake, and BigQuery?

Platform	Cost Model
Redshift	Per-node/hour (provisioned) or per-RPU (serverless)
Snowflake	Per-second compute + storage
BigQuery	Per-query (bytes scanned) + storage

→ BigQuery is ideal for **ad-hoc, infrequent** queries.







→ Redshift and Snowflake are better for **consistent workloads**.

Would you like mock interview questions with scenarios or questions specific to your experience with Redshift or Glue?



Great question! Understanding **Facts vs Dimensions** is foundational in **data warehousing** and **star schema design**. Here's a clear breakdown:

What Are Fact and Dimension Tables?

Concept	Fact Table	Dimension Table
 Definition	Stores measurable, numeric data (facts)	Stores descriptive attributes related to facts
 Content	Quantities, sales, revenue, clicks, etc.	Customers, Products, Dates, Locations, etc.
 Keys	Foreign keys to dimension tables	Primary keys (used in fact table as FKs)
 Granularity	One row per transaction/event	One row per entity (customer/product/etc.)
 Frequency of Change	Frequently updated (new facts daily)	Slowly changing (e.g., name change, rebranding)
 Examples	<code>`sales_fact`, `orders_fact`</code>	<code>`dim_customer`, `dim_product`, `dim_date`</code>

Fact Table Example

```
CREATE TABLE sales_fact (  
    sale_id          INT,  
    date_key         INT,  
    customer_key     INT,  
    product_key      INT,  
    store_key        INT,  
    quantity         INT,  
    total_amount     DECIMAL(10,2)  
);
```

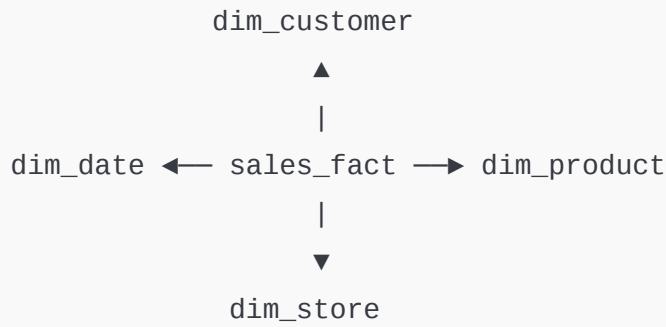
- Contains **foreign keys** to dimension tables.
- Contains **measures**: `quantity`, `total_amount`.

✓ Dimension Table Example

```
CREATE TABLE dim_product (  
    product_key      INT PRIMARY KEY,  
    product_name     VARCHAR,  
    category         VARCHAR,  
    brand            VARCHAR  
);
```

- Describes attributes of `product`.
- Used for filtering, grouping, and joining with fact table.

★ Star Schema Example



- Fact table at center, surrounded by dimensions = **Star Schema**.

Real-Life Analogy

Think of a **sales receipt**:

- **Fact**: What was sold, how many, at what price?
- **Dimensions**: Who bought it? What product? When? Where?

Fact vs Dimension: Quick Summary

Feature	Fact	Dimension
Data Type	Numeric, additive	Descriptive
Volume	Large	Smaller
Purpose	Analysis, metrics	Filtering, grouping, labels
Examples	Sales, Revenue, Clicks	Date, Product, Customer
Schema Location	Center of star schema	Surrounding fact tables

Would you like a sample dataset or ER diagram to visualize this in Redshift or Snowflake?