

Pipeline 1

Read MySQL tables using JDBC

Load `customers`, `orders`, and `order_items` from Amazon RDS MySQL into Spark DataFrames using the JDBC URL.

Perform required joins

Join `customers` → `orders` on `customer_id`, and `orders` → `order_items` on `order_id` to create a single denormalized dataset.

Select final output columns

Project the combined columns:

`customer_id`, `name`, `email`, `city`, `order_id`, `order_date`, `amount`, `item_id`, `product_name`, `quantity`.

Write to Amazon Keyspaces (Cassandra)

Save the final denormalized DataFrame into the table `retail.sales_data` using the Spark Cassandra Connector with `.mode("append")`.

```
[racit@192 ~ % mysql -h database-1.cqr2ksiuua6t.us-east-1.rds.amazonaws.com -u admin -p
[Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 50
Server version: 8.0.43 Source distribution

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

[mysql> create database spark
[    ->;
Query OK, 1 row affected (0.369 sec)

[mysql> use spark
Database changed
mysql> -- Database: retail_db
Query OK, 0 rows affected (0.285 sec)

mysql> CREATE DATABASE IF NOT EXISTS retail_db;
Query OK, 1 row affected, 1 warning (0.381 sec)

mysql> USE retail_db;
Database changed
mysql>
mysql> CREATE TABLE customers (
->     customer_id INT PRIMARY KEY,
->     name VARCHAR(255),
->     email VARCHAR(255),
->     city VARCHAR(100)
-> );
Query OK, 0 rows affected (0.320 sec)
```

The screenshot shows a completed CQL query execution. The results are displayed in a table view with the following data:

customer_id	order_id	amount	city	email	item_id	name	order_date	product_name
1	1001	250	Bengaluru	alice@example.com	5001	Alice	2025-11-29 18:30:00.0+0000	Widget

Pipeline 2

Pipeline 2 — Keyspaces → Spark → Parquet on S3

1. Spark Session Setup

- Configured Spark locally with required **Amazon Keyspaces (Cassandra)** credentials and SSL settings.
- Added **S3A configurations** with correct region, endpoint, and AWS access keys.

2. Read from Amazon Keyspaces

Loaded the `sales_data` table from Keyspace `retail` using the Cassandra Spark connector:

```
spark.read.format("org.apache.spark.sql.cassandra")
```

-

3. Select Required Columns

- Extracted only the necessary fields:
`customer_id, order_id, amount, product_name, quantity.`

4. Write to Amazon S3 as Parquet

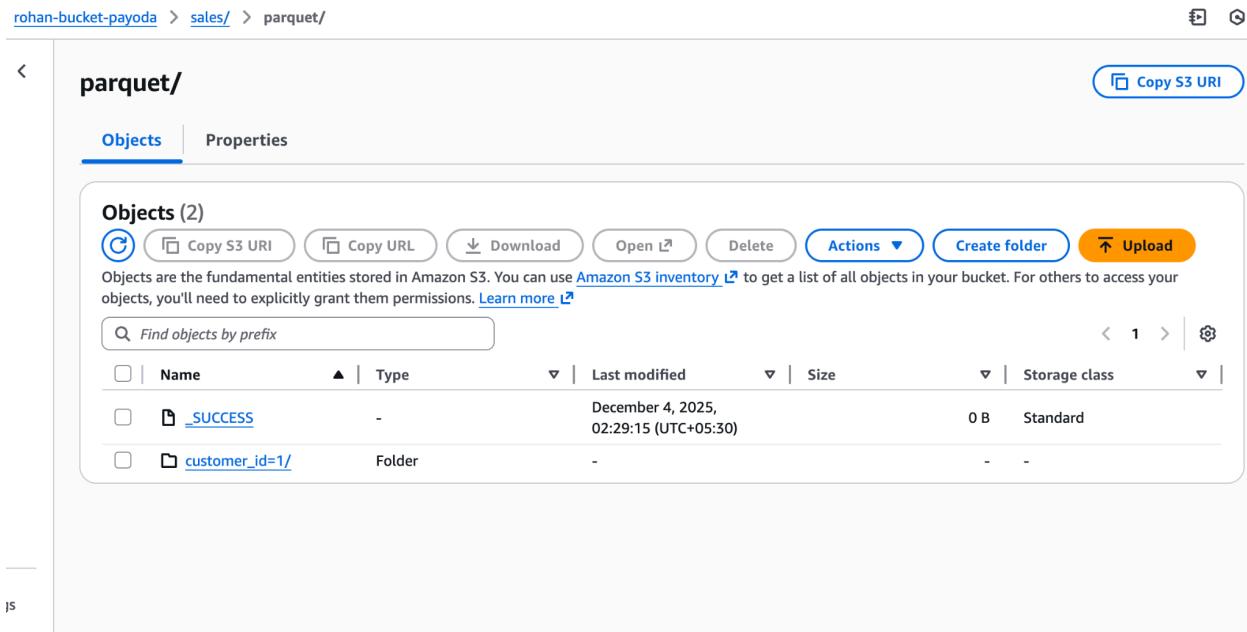
Saved the filtered data to S3 in Parquet format using:

```
partitionBy("customer_id")
```

-
- Output stored at:
`s3a://rohan-bucket-payoda/sales/parquet/`.

5. Result

- Pipeline executes successfully end-to-end:
Keyspaces → Spark processing → S3 Parquet output, partitioned by customer ID.



The screenshot shows the AWS S3 console interface. The path is `rohan-bucket-payoda > sales/ > parquet/`. The 'Objects' tab is selected, showing two items: `_SUCCESS` (a folder) and `customer_id=1/` (a folder). The Actions dropdown menu is open, displaying options such as Copy S3 URI, Copy URL, Download, Open, Delete, Create folder, and Upload. A search bar at the top allows filtering by prefix. The overall layout is clean and modern, typical of AWS web interfaces.

Pipeline 3

Parquet → Spark Aggregation → JSON on S3 (Summary)

Spark Session Setup

- Configured Spark locally with S3A access keys and correct endpoint for S3 (ap-south-2).
- Enabled reading of Parquet files stored in your S3 bucket.

Read Parquet from S3

Loaded the Parquet data generated from Pipeline 2:

```
spark.read.parquet("s3a://rohan-bucket-payoda/sales/parquet/")
```

-

Aggregate Product Metrics

- Computed:
 - **total_quantity = SUM(quantity)**
 - **total_revenue = SUM(amount * quantity)** (rounded to 2 decimals)
- Grouped by **product_name** and sorted by **total_revenue** in descending order.

Write Aggregated JSON to S3

Wrote final aggregated result to S3 as JSON:

```
s3a://rohan-bucket-payoda/aggregates/products/
```

-
- Used `coalesce(1)` to generate a single JSON output file.

Result

- Pipeline runs successfully end-to-end:
S3 Parquet → Spark Aggregation → Final JSON output stored on S3.

Objects (2)

Copy S3 URI Copy URL Download Open Delete Actions Create folder Upload

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

<input type="checkbox"/>	Name	Type	Last modified	Size	Storage class
<input type="checkbox"/>	_SUCCESS	-	December 4, 2025, 02:45:06 (UTC+05:30)	0 B	Standard
<input type="checkbox"/>	part-00000-2d72f1d3-4da8-4bfc-a506-24a7173abaa5-c000.json	json	December 4, 2025, 02:45:04 (UTC+05:30)	67.0 B	Standard

Pipeline 4

- Spark Structured Streaming **polls the new_orders MySQL table every 5 seconds** using `order_id` to detect new rows.
- Any **newly added orders** are captured in real-time.
- Each record is **converted into Avro format** using the provided `orders.avsc` schema.
- The **Avro-encoded records are sent to the Kafka topic `orders_avro_topic`.**
- This enables a **streaming pipeline** from MySQL → Spark → Kafka in near real-time.

```
#advertised.listeners=PLAINTEXT://your.host.name:9092
racit@192 kafka_2.13-3.5.1 % kafka-topics.sh --list --bootstrap-server localhost:9092

--consumer_offsets
orders_avro_topic
visitor.checkin
racit@192 kafka_2.13-3.5.1 % bin/kafka-console-consumer.sh \
--bootstrap-server localhost:9092 \
--topic orders_avro_topic \
--property print.key=true \
--from-beginning

2001    ?Y@2025-12-03T10:00:00.000Z
2006    ??i@2025-12-04T07:08:26.000Z
2007    ???b@2025-12-04T07:08:26.000Z
2008    ???t@2025-12-04T07:08:26.000Z
2009    ???y@2025-12-04T07:08:26.000Z
2010    ?????(\?X@2025-12-04T07:08:26.000Z
```

```
[mysql]> select * from new_orders
[   -> ;
+-----+-----+-----+-----+
| order_id | customer_id | amount | created_at
+-----+-----+-----+-----+
| 2001 | 1 | 100 | 2025-12-03 10:00:00 |
| 2006 | 106 | 200.5 | 2025-12-04 07:08:26 |
| 2007 | 107 | 150.75 | 2025-12-04 07:08:26 |
| 2008 | 108 | 330 | 2025-12-04 07:08:26 |
| 2009 | 109 | 410.25 | 2025-12-04 07:08:26 |
| 2010 | 110 | 99.99 | 2025-12-04 07:08:26 |
+-----+-----+-----+-----+
6 rows in set (0.248 sec)
```

```
mysql> █
```

```
[mysql]>
[mysql]> select * from new_orders
[   -> ;
+-----+-----+-----+-----+
| order_id | customer_id | amount | created_at
+-----+-----+-----+-----+
| 2001 | 1 | 100 | 2025-12-03 10:00:00 |
| 2006 | 106 | 200.5 | 2025-12-04 07:08:26 |
| 2007 | 107 | 150.75 | 2025-12-04 07:08:26 |
| 2008 | 108 | 330 | 2025-12-04 07:08:26 |
| 2009 | 109 | 410.25 | 2025-12-04 07:08:26 |
| 2010 | 110 | 99.99 | 2025-12-04 07:08:26 |
+-----+-----+-----+-----+
6 rows in set (0.248 sec)
```

```
mysql> █
```

Pipeline 5

Spark Streaming Setup: A Spark Structured Streaming job reads messages from the Kafka topic `orders_avro_topic` using `readStream` with proper Kafka options (`bootstrap.servers`, `startingOffsets`, etc.).

Avro Decoding: Each Kafka message in Avro format is decoded into a DataFrame with proper columns (`order_id`, `customer_id`, `amount`, `created_at`) using `from_avro` and an Avro schema.

Data Transformation: A `processing_time` column is added to track ingestion, and the DataFrame schema is printed for verification.

JSON Write to S3: Using `foreachBatch`, each micro-batch is written to the configured S3 path in JSON format with `batchDF.write.json(...)`, optionally coalescing files to reduce small files.

Reliability: S3 Hadoop configurations and checkpointing are configured to ensure continuous streaming and fault tolerance, so every batch of Kafka messages is successfully persisted as JSON in S3.

The screenshot shows the AWS S3 console for the bucket 'rohan-bucket-payoda'. The 'Objects' tab is active, showing two objects: 'aggregates/' and 'sales/'. The objects are listed in a table with columns for Name, Type, Last modified, Size, and Storage class. Both objects are of type 'Folder' and have a size of '-' and storage class of '-'. The table has headers for Name, Type, Last modified, Size, and Storage class. There are also columns for selection (checkbox), sorting (up/down arrows), and filtering (dropdown). At the top of the table, there are buttons for Copy S3 URI, Copy URL, Download, Open, Delete, Actions, Create folder, and Upload. A search bar at the top left says 'Find objects by prefix'.

	Name	Type	Last modified	Size	Storage class
<input type="checkbox"/>	aggregates/	Folder	-	-	-
<input type="checkbox"/>	sales/	Folder	-	-	-