**Introduction**

Apache Spark and Apache Cassandra are a powerful combination for building scalable, high-performance, and real-time big data applications. Spark’s in-memory distributed computation capabilities complement Cassandra’s scalable, fault-tolerant distributed database design. By integrating the two, you can process, analyze, and store large volumes of data efficiently.

In this tutorial, we will explore how to connect Spark to Cassandra, read data from Cassandra tables, and write data back to Cassandra using Scala. By the end, you will understand the basics of Spark-Cassandra integration and how to perform seamless data exchange between the two systems.

**How does it work?**

The fundamental idea is quite simple: Spark and Cassandra clusters are deployed to the same set of machines. Cassandra stores the data; Spark worker nodes are co-located with Cassandra and do the data processing.

Spark is a batch-processing system, designed to deal with large amounts of data. When a job arrives, the Spark workers load data into memory, spilling to disk if necessary. The important aspect of this is that there is no network traffic. The Spark worker understands how Cassandra distributes the data and reads only from the local node.

Afterwards the “usual” map-reduce style behaviour takes place: certain operations can and will be executed on local nodes, others require data to be exchanged between the Spark workers (in Spark terminology this is called a “shuffle”).

Finally, the results are written back somewhere – again there are multiple storage options ranging from dumping a set of CSV files to local disk, HDFS, or databases such as Cassandra or an RDBMS.

To execute the CQLs statement of this tutorial we will be using a Zeppelin note with the **cassandra**interpreter (**%cassandra**).

%cassandra

desc keyspaces;

**Laying the foundations**

To begin with, let’s explore what is required from the Cassandra perspective to store the data. To do this on a basic level, we only need a very a simple schema. We will be using two tables: one to capture basic customer information, another to log card transactions.

Let’s start with the Keyspace creation.

**Creating KeySpace and Tables**

1. **Create the KeySpace**

Using CQL, we create a keyspace named **tutorial** (if it doesn’t already exists). We will set replication factor to 1 and use the **SimpleStrategy**class.

%cassandra

# Keyspace creation

CREATE KEYSPACE if not exists tutorial

WITH replication = {'class': 'SimpleStrategy', 'replication\_factor': 1};

1. **Create the first Table**

Create the customers table with the required structure, with **id** as the primary key.

%cassandra

# Create the customers table

CREATE TABLE if not exists tutorial.customers (

id text PRIMARY KEY,

county text,

name text

);

1. **Create the second Table**

Create the transactions table with the required structure, with **id** as the primary key.

%cassandra

# Create the transactions table

CREATE TABLE if not exists tutorial.transactions (

customerid text,

year int,

month int,

id timeuuid,

amount int,

card text,

status text,

PRIMARY KEY ((customerid, year, month), id)

);

1. **Load data Into The Table**

Table customers is fairly simple: I’ve added the customer’s home county to allow for some rollup-processing demonstration. The other table, transactions is the primary store for transaction data. It stores transactions per customer, bucketed by time; so it’s directly suitable to query for transaction history based on time criteria.

In this example a transaction status can be PENDING, COMPLETED, FAILED or REPAID; any charges will have a negative amount while repayments will appear as positive ones. Also, a single customer can have multiple cards.Transactions have a unique TimeUUID identifier that is a good clustering key as Cassandra will store them in their natural “wall clock time” order.

Let’s load some sample data into each table.

%cassandra

# Load sample data into customers table

INSERT INTO tutorial.customers (id, county, name) VALUES ('1', 'Orange County', 'Alice Johnson');

INSERT INTO tutorial.customers (id, county, name) VALUES ('2', 'Los Angeles County', 'Bob Smith');

INSERT INTO tutorial.customers (id, county, name) VALUES ('3', 'San Diego County', 'Charlie Brown');

INSERT INTO tutorial.customers (id, county, name) VALUES ('4', 'Riverside County', 'Diana Prince');

INSERT INTO tutorial.customers (id, county, name) VALUES ('5', 'Santa Clara County', 'Eve Adams');

INSERT INTO tutorial.customers (id, county, name) VALUES ('6', 'San Francisco County', 'Frank Castle');

INSERT INTO tutorial.customers (id, county, name) VALUES ('7', 'Alameda County', 'Grace Lee');

INSERT INTO tutorial.customers (id, county, name) VALUES ('8', 'Sacramento County', 'Henry Ford');

INSERT INTO tutorial.customers (id, county, name) VALUES ('9', 'Fresno County', 'Ivy Taylor');

INSERT INTO tutorial.customers (id, county, name) VALUES ('10', 'Ventura County', 'Jack Sparrow');

1. **Check Data Is Loaded**

Use SELECT to verify that the data have been loaded properly. Include LIMIT to retrieve only the first 5 rows.

%cassandra

# Check the table

SELECT \* FROM tutorial.customers LIMIT 5;

Add sample data to the transactions table.

**Sample Data Description**

* **customerid**: Randomly selected from a predefined list of customer IDs.
* **year**: Fixed as 2023 (can be adjusted as needed).
* **month**: Randomized from 1 to 12.
* **id**: A unique time-based UUID for each transaction.
* **amount**: A random transaction amount between 1 and 1000.
* **card**: Randomly selected card type (e.g., VISA, MASTERCARD).
* **status**: Random transaction status (e.g., APPROVED, DECLINED).

%cassandra

# Add sample data to the transactions table

INSERT INTO tutorial.transactions (customerid, year, month, id, amount, card, status) VALUES ('1', 2023, 1, now(), 500, 'VISA', 'APPROVED');

INSERT INTO tutorial.transactions (customerid, year, month, id, amount, card, status) VALUES ('2', 2023, 2, now(), 200, 'MASTERCARD', 'DECLINED');

INSERT INTO tutorial.transactions (customerid, year, month, id, amount, card, status) VALUES ('3', 2023, 3, now(), 750, 'AMEX', 'PENDING');

INSERT INTO tutorial.transactions (customerid, year, month, id, amount, card, status) VALUES ('4', 2023, 4, now(), 400, 'DISCOVER', 'APPROVED');

INSERT INTO tutorial.transactions (customerid, year, month, id, amount, card, status) VALUES ('5', 2023, 5, now(), 300, 'VISA', 'DECLINED');

INSERT INTO tutorial.transactions (customerid, year, month, id, amount, card, status) VALUES ('1', 2023, 6, now(), 900, 'MASTERCARD', 'APPROVED');

INSERT INTO tutorial.transactions (customerid, year, month, id, amount, card, status) VALUES ('2', 2023, 7, now(), 100, 'AMEX', 'PENDING');

INSERT INTO tutorial.transactions (customerid, year, month, id, amount, card, status) VALUES ('3', 2023, 8, now(), 650, 'DISCOVER', 'DECLINED');

INSERT INTO tutorial.transactions (customerid, year, month, id, amount, card, status) VALUES ('4', 2023, 9, now(), 850, 'VISA', 'APPROVED');

INSERT INTO tutorial.transactions (customerid, year, month, id, amount, card, status) VALUES ('5', 2023, 10, now(), 700, 'MASTERCARD', 'PENDING');

Use SELECT to verify that the data have been loaded properly. Include LIMIT to retrieve only the first 5 rows.

%cassandra

# chack the transactions table

SELECT \* FROM tutorial.transactions LIMIT 5;

**Spark-Cassandra Connector**

To connect Spark to Cassandra, you will need to include the DataStax Spark-Cassandra connector in your project. This connector allows Spark to access Cassandra tables as if they were Dataframes. This connector is already installed on the sandbox with all its dependencies and configured so you don’t need to worry about this part.

**Reading Data from Cassandra**

To read data from Cassandra, we need to setup the Spark-Cassandra connector with the configuration details. Below is a step-by-step guide with a code example.

**Steps to Read Data**

1. **Set Up SparkSession**: Configure Spark to use the Cassandra connector. (*automatically created in Zeppelin note*)
2. **Specify Cassandra Options**: Provide the Cassandra keyspace and table details. (*Specifies the keyspace and table to read from.*)
3. **Load Data into a DataFrame**: Use the spark.read method to load data.
4. **Perform Transformations and Actions**: Use Spark APIs to process the data.

%spark

// Load the Cassandra transactions table into a DataFrame

val transactionsDF = spark.read

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

.option("keyspace", "tutorial")

.option("table", "transactions")

.load

**Performing Data Analysis**

A reasonable request from any customer is to understand what is their current balance on each of their cards. When asked the question: given my customer id and card, how much money do I have? In this case our application would need to answer with a single number, the sum of all completed transactions and repayments on that particular card.

**Query 1: Data Aggregation 1**

* Calculate the current balance per customer.

%spark

// Q1: Calculate the current balance per customer

val balanceDF = transactionsDF

.groupBy("customerid")

.agg(sum("amount").as("current\_balance")) // Sum up the 'amount' column for each customer

.orderBy("customerid") // Optional: Sort by customer ID

%spark

​// Show the resulting aggregated data

//aggregatedDF.show(truncate = false)

z.show(aggregatedDF)

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2024/12/q1-Interacting-with-Cassandra-from-Spark.png)

We could also want to calculate the total number of transactions by customer and card type. The following query will group the data by customerid and card columns. We calculate also:

* **transaction\_coun**t: Total number of transactions for each customer per card type.
* **average\_transaction**: Average transaction amount for each customer per card type.
* **total\_transaction**: Total transaction amount for each customer per card type.

**Query 2: Data Aggregation 2**

* Calculate the Total number of transactions and average amount by customer and card type.

%spark

// Q2: Perform aggregation: Total number of transactions and average amount by customer and card type

val aggregatedDF = transactionsDF

.groupBy("customerid", "card") // Group by customer ID and card type

.agg(

count("id").as("transaction\_count"), // Count the number of transactions

avg("amount").as("average\_transaction"), // Calculate the average transaction amount

sum("amount").as("total\_transaction") // Calculate the total transaction amount

)

.orderBy("customerid", "card") // Optional: Sort by customer ID and card type

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2024/12/q2-Interacting-with-Cassandra-from-Spark.png)

Another use-case might be to try and gain an understanding of how much people in different counties spend per month. In a relational database this would be a trivial operation, but would more than likely require a table join. Cassandra does not support joins, and aggregations against the full dataset are not efficient. Again, we can use Spark to generate the data we are interested in. For this particular use-case I will use a different API of Spark: SparkSQL to demonstrate.

**Query 3: Data Aggregation 3**

* Calculate total spending per county per month.

%spark

// Join transactions with customers on customer ID

val joinedDF = transactionsDF.join(customersDF, transactionsDF("customerid") === customersDF("id"), "inner")

%spark

//Q3: Calculate total spending per county per month

val spendingPerCountyDF = joinedDF

.groupBy("county", "year", "month") // Group by county, year, and month

.agg(sum("amount").as("total\_spent")) // Sum up the amount for each group

.orderBy("county", "year", "month") // Optional: Sort the results

%spark

// Show the results

// spendingPerCountyDF.show(truncate = false)

z.show(spendingPerCountyDF)

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2024/12/q3-Interacting-with-Cassandra-from-Spark.png)

To create a Spark SQL script that reports the names of all users having their transactions completed (where status = 'APPROVED'), you first need to ensure the transactions data and customer data are loaded into temporary views. Below is a complete example of a Spark SQL script for this purpose.

%spark

// Create temporary views for SQL querying

transactionsDF.createOrReplaceTempView("transactions")

customersDF.createOrReplaceTempView("customers")

**Query 4: Spark SQL Join**

* Using Spark SQL find names of users with approved transactions.

%spark

//Q4: Write the Spark SQL query to find names of users with approved transactions

val query = """

SELECT DISTINCT c.id, c.name

FROM customers c

JOIN transactions t

ON c.id = t.customerid

WHERE t.status = 'APPROVED'

"""

%spark

// Execute the query

val resultDF = spark.sql(query)

%spark

// Show the result

resultDF.show(truncate = false)

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2024/12/q3-Interacting-with-Cassandra-from-Spark-1.png-1.png)

**Write Data Back to Cassandra**

Writing data back to Cassandra in Spark involves saving a DataFrame to a Cassandra table using the Spark-Cassandra Connector. Here’s how you can do it:

**Steps to Write Data Back to Cassandra**

1. **Ensure the Target Table Exists**:
   * Before writing data, the target table in Cassandra must already exist. Use a CREATE TABLE statement in cqlsh or programmatically create it.
2. **Write the DataFrame**:
   * Use the .write method on the DataFrame with the following configurations:
     + format: Set to "org.apache.spark.sql.cassandra".
     + options: Specify the target keyspace and table.
     + mode: Choose "append" to add data to the table or "overwrite" to replace existing data.

Let’s say we want to write back to Cassandra the names of all users having their transactions completed. Here’s the Cassandra CREATE TABLE script for the approved\_users table to store the results of approved transactions:

* The table resides in the tutorial keyspace.
* **Table Columns**:
  + id: A unique identifier (in text format) for each approved user.
  + name: The name of the approved user.

%cassandra

# Create table for approved users name

CREATE TABLE tutorial.approved\_users (

id text PRIMARY KEY,

name text

);

**Save Data to Cassandra**

Ensure the approved\_users DataFrame is saved correctly after creating the table. Below is an example of how to save the results:

%spark

// write the result back to the Cassandra table

resultDF.write

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

.option("keyspace", "tutorial")

.option("table", "approved\_users")

.mode("append")

.save

**Verify the Data**

After the write operation, verify the saved data using the CQL Select command:

%cassandra

# Verify the data

SELECT \* FROM tutorial.approved\_users;

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2024/12/Interacting-with-Cassandra-from-Spark.wrtiing.png)

**Summary**

Integrating Spark with Apache Cassandra allows you to build efficient, real-time data pipelines for large-scale applications. In this tutorial, we covered the basics of connecting Spark to Cassandra, reading data from Cassandra tables, and writing data back to Cassandra. By applying these concepts, you can leverage the combined power of Spark’s processing capabilities and Cassandra’s storage efficiency to create robust data solutions.

Feel free to extend this setup by adding transformations, aggregations, or integrating other data sources to enrich your pipeline!

Open Zeppelin Notes: [Interacting with Cassandra from Spark](http://localhost:19995/#/notebook/2KJGKGXX6)