Spark-Connect-Cassandra

June 28, 2024

0.0.1 1. In Apache Spark, you can create and use a database using Spark SQL. Here are the steps to do this:

1. **Creating a Database**: Use the `CREATE DATABASE` statement to create a new database. You can also specify the location for the database files if needed.

   ```
   CREATE DATABASE IF NOT EXISTS my_database
   LOCATION 'path/to/database/location';
   ```

2. **Using a Database**: Once the database is created, you can switch to it using the `USE` statement. This sets the specified database as the current database for the session.

   ```
   USE my_database;
   ```

3. **Creating Tables in the Database**: After switching to the desired database, you can create tables within it.

   ```
   CREATE TABLE IF NOT EXISTS my_table (
       id INT,
       name STRING,
       age INT
   )
   USING parquet;
   ```

4. **Inserting Data into the Table**: You can insert data into the table using the `INSERT INTO` statement.

   ```
   INSERT INTO my_table (id, name, age)
   VALUES (1, 'John Doe', 30),
          (2, 'Jane Doe', 25);
   ```

5. **Querying Data from the Table**: You can query the data from the table using the `SELECT` statement.

   ```
   SELECT * FROM my_table;
   ```

Here is a complete example using PySpark to demonstrate these steps:

```
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder \
    .appName("Create and Use Database") \
    .enableHiveSupport() \
```

```
    .getOrCreate()

# Create a database
spark.sql("CREATE DATABASE IF NOT EXISTS my_database LOCATION 'path/to/database/location'")

# Use the database
spark.sql("USE my_database")

# Create a table
spark.sql("""
    CREATE TABLE IF NOT EXISTS my_table (
        id INT,
        name STRING,
        age INT
    )
    USING parquet
""")

# Insert data into the table
spark.sql("""
    INSERT INTO my_table (id, name, age)
    VALUES (1, 'John Doe', 30),
           (2, 'Jane Doe', 25)
""")

# Query data from the table
result = spark.sql("SELECT * FROM my_table")
result.show()
```

This example covers the basic operations of creating and using a database in Spark SQL. Adjust the path, table schema, and data as per your requirements.

0.0.2 2 .To load external data sources like CSV or JSON files into Apache Cassandra, you typically follow these steps:

1. **Prepare the Cassandra Environment**: Ensure you have Apache Cassandra running and the necessary keyspace and table(s) created.
2. **Use a Tool for Data Loading**: You can use tools like `cqlsh`, Apache Spark with the Cassandra connector, or Python with `cassandra-driver` to load data into Cassandra.

Here's a detailed guide on how to do this using different methods:

(a) Using Apache Spark with the Cassandra Connector Apache Spark provides a robust way to load large datasets into Cassandra. Here's how you can do it:

1. **Setup Spark and Cassandra Connector**: Ensure you have Apache Spark and the Spark Cassandra Connector installed.

2. **Load Data Using PySpark**:

```
from pyspark.sql import SparkSession

# Create Spark session
spark = SparkSession.builder \
    .appName("Load Data into Cassandra") \
    .config("spark.cassandra.connection.host", "your_cassandra_host") \
    .getOrCreate()

# Load CSV data
df_csv = spark.read.csv("path/to/yourfile.csv", header=True, inferSchema=True)

# Load JSON data
df_json = spark.read.json("path/to/yourfile.json")

# Write data to Cassandra
df_csv.write \
    .format("org.apache.spark.sql.cassandra") \
    .mode('append') \
    .options(table="your_table", keyspace="your_keyspace") \
    .save()

df_json.write \
    .format("org.apache.spark.sql.cassandra") \
    .mode('append') \
    .options(table="your_table", keyspace="your_keyspace") \
    .save()
```

(b) Using Python with `cassandra-driver` You can also use Python's `cassandra-driver` to load data directly into Cassandra.

1. **Install `cassandra-driver`**:

```
pip install cassandra-driver pandas
```

2. **Load Data Using Python**:

```
from cassandra.cluster import Cluster
import pandas as pd

# Create a connection to the Cassandra cluster
cluster = Cluster(['your_cassandra_host'])
session = cluster.connect('your_keyspace')

# Load CSV data using pandas
df_csv = pd.read_csv('path/to/yourfile.csv')

# Load JSON data using pandas
```

```
df_json = pd.read_json('path/to/yourfile.json')

# Insert data into Cassandra
for _, row in df_csv.iterrows():
    session.execute(
        """
        INSERT INTO your_table (column1, column2, column3)
        VALUES (%s, %s, %s)
        """,
        (row['column1'], row['column2'], row['column3'])
    )

for _, row in df_json.iterrows():
    session.execute(
        """
        INSERT INTO your_table (column1, column2, column3)
        VALUES (%s, %s, %s)
        """,
        (row['column1'], row['column2'], row['column3'])
    )
```

(c) Using `cqlsh` and `COPY` Command The `COPY` command in `cqlsh` is a straightforward way to import CSV data into Cassandra.

1. **Ensure Data Format**: Make sure your CSV file matches the table schema in Cassandra.
2. **Use `cqlsh` to Load Data**:

```
cqlsh your_cassandra_host

USE your_keyspace;

COPY your_table (column1, column2, column3) FROM 'path/to/yourfile.csv' WITH HEADER = TRUE
```

Note: The `COPY` command is suitable for smaller datasets. For larger datasets, it's recommended to use Apache Spark or custom scripts with `cassandra-driver`.

Summary

- For large datasets, use Apache Spark with the Cassandra connector.
- For smaller datasets or quick imports, use `cqlsh` with the `COPY` command.
- For flexible scripting, use Python with `cassandra-driver`.

Choose the method that best fits your data size and workflow requirements.

0.0.3 3. Using `cqlsh` in the terminal to load external data like CSV files into Cassandra involves the following steps:

1. **Ensure Your Environment is Set Up**: Make sure Apache Cassandra is running, and you have the necessary keyspace and table created.
2. **Prepare Your CSV File**: Ensure your CSV file matches the schema of the table in Cassandra, and it has a header row if you intend to use the `HEADER = TRUE` option.
3. **Use `cqlsh` to Load the Data**:

Here is a step-by-step guide:

0.0.4 Step 1: Start `cqlsh`

Open your terminal and start `cqlsh` by running:

```
cqlsh your_cassandra_host
```

Replace `your_cassandra_host` with the IP address or hostname of your Cassandra node.

0.0.5 Step 2: Select the Keyspace

Once you are in `cqlsh`, select the keyspace you want to use:

```
USE your_keyspace;
```

Replace `your_keyspace` with the name of your keyspace.

0.0.6 Step 3: Prepare the Table

Ensure you have a table that matches the schema of your CSV file. For example:

```
CREATE TABLE IF NOT EXISTS your_table (
    column1 int,
    column2 text,
    column3 double,
    PRIMARY KEY (column1)
);
```

0.0.7 Step 4: Load the CSV Data

Use the `COPY` command to load the CSV data into the table. Here is the syntax:

```
COPY your_table (column1, column2, column3) FROM 'path/to/yourfile.csv' WITH HEADER = TRUE;
```

Replace `your_table` with the name of your table, and `path/to/yourfile.csv` with the path to your CSV file.

0.0.8 Example

Assume you have a CSV file named `data.csv` with the following content:

```
column1,column2,column3
1,John Doe,29.5
2,Jane Doe,34.2
```

And a table `your_table` in the keyspace `your_keyspace` with the following schema:

```
CREATE TABLE IF NOT EXISTS your_table (
    column1 int,
    column2 text,
    column3 double,
    PRIMARY KEY (column1)
);
```

You would load the data as follows:

1. Start `cqlsh`:

   ```
   cqlsh localhost
   ```

2. Select the keyspace:

   ```
   USE your_keyspace;
   ```

3. Load the CSV data:

   ```
   COPY your_table (column1, column2, column3) FROM 'path/to/data.csv' WITH HEADER = TRUE;
   ```

Note: Ensure the path to the CSV file is correct and accessible from the location where you are running `cqlsh`.

0.0.9 Handling Errors and Data Validation

- **Data Type Mismatches**: Ensure that the data types in the CSV file match the table schema.
- **File Accessibility**: Ensure the CSV file is accessible from the node where `cqlsh` is running.
- **Escaping Special Characters**: If your CSV contains special characters, make sure they are properly escaped.

By following these steps, you can successfully load external CSV data into Cassandra using `cqlsh`.

When using PySpark to load data into Cassandra, the schema of your Cassandra table does not necessarily have to be exactly the same as the schema of your external data source (like a CSV or JSON file), but there are important considerations to ensure successful data loading:

0.0.10 Key Points to Consider:

1. **Column Names and Data Types**:
   - The column names in your external data source must match the column names in your Cassandra table.

- The data types of the columns in your external data source should be compatible with the data types in your Cassandra table.

2. **Primary Key**:
   - Your Cassandra table must have a defined primary key (partition key and optional clustering keys). This is crucial for writing data into Cassandra.
3. **Handling Extra or Missing Columns**:
   - If your external data source has extra columns that are not present in the Cassandra table, you can exclude those columns while writing the data.
   - If your external data source is missing columns that are present in the Cassandra table, you need to ensure that the missing columns can accept null values or provide default values.

0.0.11 Example Scenario:

Assume you have a CSV file with the following content:

```
id,name,age,salary
1,John Doe,30,50000
2,Jane Doe,25,60000
```

And a Cassandra table with the following schema:

```
CREATE TABLE your_keyspace.your_table (
    id int PRIMARY KEY,
    name text,
    age int
);
```

In this case, the `salary` column is an extra column in the CSV file that does not exist in the Cassandra table. You can handle this by selecting only the necessary columns when writing data to Cassandra.

0.0.12 PySpark Code Example:

Here's how you can load the CSV data into Cassandra using PySpark, while excluding the extra column:

```
from pyspark.sql import SparkSession

# Create Spark session
spark = SparkSession.builder \
    .appName("Load Data into Cassandra") \
    .config("spark.cassandra.connection.host", "your_cassandra_host") \
    .getOrCreate()

# Load CSV data
df_csv = spark.read.csv("path/to/yourfile.csv", header=True, inferSchema=True)

# Select necessary columns (excluding 'salary')
```

```
df_csv = df_csv.select("id", "name", "age")

# Write data to Cassandra
df_csv.write \
    .format("org.apache.spark.sql.cassandra") \
    .mode('append') \
    .options(table="your_table", keyspace="your_keyspace") \
    .save()
```

0.0.13 Important Notes:

- **Ensure Schema Compatibility**: Ensure that the data types in your DataFrame match the data types expected by the Cassandra table.
- **Primary Key Constraint**: Ensure that the primary key columns (in this case, `id`) are included and correctly populated in your DataFrame.

By following these guidelines, you can successfully load external data sources into Cassandra using PySpark, even if the schemas do not match exactly.

0.0.14 4.The process for loading different file formats like JSON, Parquet, ORC, and Avro into Cassandra using PySpark is similar to loading CSV files. Here's how you can handle these different file formats:

Prerequisites

1. **Ensure Apache Spark and Spark Cassandra Connector are installed**.
2. **Ensure Cassandra is running and accessible**.
3. **Ensure the Spark Cassandra Connector is added to your Spark session**.

Example PySpark Code for Different File Formats Here's how to read and write different file formats to Cassandra using PySpark:

1. JSON Files

```
from pyspark.sql import SparkSession

# Create Spark session
spark = SparkSession.builder \
    .appName("Load JSON into Cassandra") \
    .config("spark.cassandra.connection.host", "127.0.0.1") \
    .getOrCreate()

# Load JSON data
df_json = spark.read.json("path/to/yourfile.json")

# Write data to Cassandra
df_json.write \
```

```
    .format("org.apache.spark.sql.cassandra") \
    .mode('append') \
    .options(table="your_table", keyspace="your_keyspace") \
    .save()
```

2. Parquet Files

```
from pyspark.sql import SparkSession

# Create Spark session
spark = SparkSession.builder \
    .appName("Load Parquet into Cassandra") \
    .config("spark.cassandra.connection.host", "127.0.0.1") \
    .getOrCreate()

# Load Parquet data
df_parquet = spark.read.parquet("path/to/yourfile.parquet")

# Write data to Cassandra
df_parquet.write \
    .format("org.apache.spark.sql.cassandra") \
    .mode('append') \
    .options(table="your_table", keyspace="your_keyspace") \
    .save()
```

3. ORC Files

```
from pyspark.sql import SparkSession

# Create Spark session
spark = SparkSession.builder \
    .appName("Load ORC into Cassandra") \
    .config("spark.cassandra.connection.host", "127.0.0.1") \
    .getOrCreate()

# Load ORC data
df_orc = spark.read.orc("path/to/yourfile.orc")

# Write data to Cassandra
df_orc.write \
    .format("org.apache.spark.sql.cassandra") \
    .mode('append') \
    .options(table="your_table", keyspace="your_keyspace") \
    .save()
```

4. Avro Files To read Avro files, you need to include the Avro package. You can do this by specifying the package when starting PySpark or when submitting your Spark application.

```
pyspark --packages org.apache.spark:spark-avro_2.12:3.0.1
```

Then, you can use the following code to read and write Avro data:

```
from pyspark.sql import SparkSession

# Create Spark session
spark = SparkSession.builder \
    .appName("Load Avro into Cassandra") \
    .config("spark.cassandra.connection.host", "127.0.0.1") \
    .getOrCreate()

# Load Avro data
df_avro = spark.read.format("avro").load("path/to/yourfile.avro")

# Write data to Cassandra
df_avro.write \
    .format("org.apache.spark.sql.cassandra") \
    .mode('append') \
    .options(table="your_table", keyspace="your_keyspace") \
    .save()
```

0.0.15 Key Considerations

- **Column Names and Data Types**: Ensure that the column names and data types in the data files are compatible with the Cassandra table schema.
- **Primary Key**: Ensure the DataFrame includes the primary key columns required by the Cassandra table.
- **Data Cleansing**: Perform any necessary data cleansing or transformation before writing to Cassandra to ensure data integrity and consistency.

0.0.16 Summary

1. **JSON**: Use `spark.read.json` to load JSON files.
2. **Parquet**: Use `spark.read.parquet` to load Parquet files.
3. **ORC**: Use `spark.read.orc` to load ORC files.
4. **Avro**: Use `spark.read.format("avro").load` to load Avro files (ensure the Avro package is included).

By following these steps, you can load various file formats into Cassandra using PySpark.

To load data into Cassandra using `cqlsh`, the process is most straightforward with CSV files. Here's how to do it for various formats:

0.0.17 CSV Files

1. Ensure Your Environment is Set Up Make sure Apache Cassandra is running, and you have the necessary keyspace and table created.

2. Prepare Your CSV File Ensure your CSV file matches the schema of the table in Cassandra, and it has a header row if you intend to use the `HEADER = TRUE` option.

3. Use `cqlsh` to Load the Data Here is a step-by-step guide for loading a CSV file using `cqlsh`:

Example Assume you have a CSV file named `data.csv` with the following content:

```
id,name,age
1,John Doe,30
2,Jane Doe,25
```

And a table `your_table` in the keyspace `your_keyspace` with the following schema:

```
CREATE TABLE your_keyspace.your_table (
    id int PRIMARY KEY,
    name text,
    age int
);
```

You would load the data as follows:

1. Start `cqlsh`:

   ```
   cqlsh localhost
   ```

2. Select the keyspace:

   ```
   USE your_keyspace;
   ```

3. Load the CSV data:

   ```
   COPY your_table (id, name, age) FROM 'path/to/data.csv' WITH HEADER = TRUE;
   ```

Replace `path/to/data.csv` with the correct path to your CSV file.

0.0.18 JSON, Parquet, ORC, and Avro Files

`cqlsh` does not natively support loading JSON, Parquet, ORC, or Avro files directly into Cassandra. However, you can convert these files to CSV format and then use the `COPY` command to load the data.

Convert JSON to CSV You can use tools like `pandas` in Python to convert JSON to CSV:

```
import pandas as pd

# Read JSON file
df = pd.read_json('path/to/yourfile.json')

# Save to CSV
df.to_csv('path/to/yourfile.csv', index=False)
```

Convert Parquet to CSV

```
import pandas as pd

# Read Parquet file
df = pd.read_parquet('path/to/yourfile.parquet')

# Save to CSV
df.to_csv('path/to/yourfile.csv', index=False)
```

Convert ORC to CSV

```
import pandas as pd

# Read ORC file
df = pd.read_orc('path/to/yourfile.orc')

# Save to CSV
df.to_csv('path/to/yourfile.csv', index=False)
```

Convert Avro to CSV You can use the `fastavro` library to read Avro files and convert them to CSV:

```
import pandas as pd
import fastavro

# Read Avro file
with open('path/to/yourfile.avro', 'rb') as f:
    reader = fastavro.reader(f)
    df = pd.DataFrame([record for record in reader])

# Save to CSV
df.to_csv('path/to/yourfile.csv', index=False)
```

0.0.19 Loading the Converted CSV File

Once you have converted your data to CSV format, use the `COPY` command in `cqlsh` as shown in the previous section to load the data into Cassandra.

0.0.20 Summary

- **CSV**: Directly load using `COPY` command in `cqlsh`.
- **JSON, Parquet, ORC, Avro**: Convert to CSV first, then load using `COPY` command in `cqlsh`.

By converting your data to CSV format and then using the `COPY` command in `cqlsh`, you can efficiently load data from various file formats into Cassandra.

[ ]: