Sure, let's delve deeper into Apache Hive, particularly focusing on its use in a Cloudera environment. I'll outline some complex aspects including steps, code, and scripts.

Deep Dive into Apache Hive

Architecture Overview

- **Components**: Hive consists of three main parts: HiveQL, Metastore, and Execution Engine.
  - **HiveQL**: A SQL-like scripting language for querying data in Hive.
  - **Metastore**: Stores metadata about the Hive tables (like schema and location).
  - **Execution Engine**: Processes HiveQL commands, converts them into MapReduce or Tez jobs, and executes them on the Hadoop cluster.

HiveQL

HiveQL is an SQL-like language that gets converted into MapReduce or Tez jobs. Here's an example of a complex HiveQL query:

```
SELECT customer_id, count(*) as orders_count
FROM orders
WHERE order_date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY customer_id
HAVING count(*) > 5
ORDER BY orders_count DESC;
```

This query selects customers with more than five orders in the year 2023.

Creating and Managing Tables

Hive supports table creation and management similar to traditional databases. Here's an example of creating a complex table with partitions and buckets:

```
CREATE TABLE sales_data (
    sale_id int,
    product_id int,
    quantity_sold int,
    sale_date date
)
PARTITIONED BY (region string)
CLUSTERED BY (product_id) INTO 10 BUCKETS
STORED AS ORC;
```

This creates a table `sales_data`, partitioned by `region`, and clustered into 10 buckets based on `product_id`.

Hive on Cloudera

In a Cloudera environment, you manage Hive services through Cloudera Manager. Here are some steps for complex configurations:

1. **Configuring Hive for High Availability**:
   - Set up a Hive Metastore database that supports replication (like MySQL or PostgreSQL).
   - Configure Hive to use this Metastore in Cloudera Manager.
   - Set up multiple HiveServer2 instances for load balancing.
2. **Integrating with Hadoop Security (Kerberos)**:
   - Enable Kerberos authentication in Cloudera Manager.
   - Configure Hive to use Kerberos for client authentication.

Example: Data ETL Process

Here's a hypothetical complex ETL process using Hive:

1. **Extract Data**: Load data from external sources (like CSV files) into raw Hive tables.

   ```
   LOAD DATA INPATH '/path/to/data.csv' INTO TABLE raw_sales_data;
   ```

2. **Transform Data**: Perform transformations using HiveQL.

   ```
   INSERT INTO TABLE processed_sales_data
   SELECT sale_id, product_id, quantity_sold, sale_date, region
   FROM raw_sales_data
   WHERE sale_date IS NOT NULL;
   ```

3. **Load Data**: Load the transformed data into a final data warehouse table, possibly partitioned by date or region for efficient querying.

Performance Tuning

- **Query Optimization**: Use EXPLAIN to understand the query execution plan and optimize queries.
- **File Formats and Compression**: Use efficient file formats like ORC or Parquet and enable compression to improve I/O performance.
- **Cost-Based Optimization (CBO)**: Enable CBO in Hive for optimal execution plans based on data statistics.

Script Example

A complex Hive script might involve multiple queries, temporary tables, and custom UDFs (User Defined Functions). Here's a simplified example:

```
-- Temporary Table for Aggregated Sales
CREATE TEMPORARY TABLE temp_sales_aggregate AS
SELECT product_id, sum(quantity_sold) as total_sold
FROM sales_data
```

```
GROUP BY product_id;

-- Join with Product Table to Get Product Names
SELECT p.name, t.total_sold
FROM product p JOIN temp_sales_aggregate t ON (p.id = t.product_id)
WHERE t.total_sold > 1000;
```

Notes:

To proceed with the example involving Hive and a `data.csv` file, let's first create a synthesis of what `data.csv` might contain and then outline the steps to create a database and a table in Hive into which this data can be loaded.

Synthesizing `data.csv` Content

Assuming `data.csv` is related to sales data, it might contain columns like:

1. `sale_id` (integer): Unique identifier for each sale.
2. `product_id` (integer): Identifier for the product sold.
3. `quantity_sold` (integer): Number of units sold in the sale.
4. `sale_date` (date): Date of the sale in the format `YYYY-MM-DD`.
5. `region` (string): Region where the sale occurred.

A snippet of `data.csv` might look like this:

```
sale_id,product_id,quantity_sold,sale_date,region
1001,501,3,2024-01-10,North
1002,502,5,2024-01-11,East
1003,503,2,2024-01-12,West
...
```

Creating Database and Table in Hive

Before loading data into Hive, you need to create a database and a table that corresponds to the structure of `data.csv`.

Step 1: Create Hive Database

```
CREATE DATABASE IF NOT EXISTS sales_data_db;
```

This command creates a database named `sales_data_db` in Hive.

Step 2: Create Hive Table

Switch to the newly created database and create a table that matches the structure of `data.csv`.

```
USE sales_data_db;

CREATE TABLE IF NOT EXISTS raw_sales_data (
```

```
    sale_id INT,
    product_id INT,
    quantity_sold INT,
    sale_date DATE,
    region STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

This command creates a table named `raw_sales_data` with columns corresponding to the fields in `data.csv`. The `ROW FORMAT DELIMITED FIELDS TERMINATED BY ','` clause indicates that the data is comma-separated, as is typical in CSV files.

Step 3: Load Data Into Table

After the table is created, you can load the data from `data.csv` into the `raw_sales_data` table. Assuming `data.csv` is stored in HDFS, the command would be:

```
LOAD DATA INPATH '/path/to/data.csv' INTO TABLE raw_sales_data;
```

This command loads the data from `data.csv` located at `/path/to/data.csv` in HDFS into the `raw_sales_data` table in the `sales_data_db` database.