* Program explanation :  
    
  **Goal:** run Spark code that **reads an HBase table** and does a tiny bit of analytics (a group-by count) **without** talking to HBase directly from Spark.
* **How we do it:** we go **through Hive**. We first create a **Hive external table** that uses Hive’s **HBaseStorageHandler** (you already did this). That table (e.g., hbase\_emp) is just a *schema and mapping* that points to the real HBase table/columns.  
  Then this Spark program uses **HiveContext** (Spark 1.6) to run normal **SQL** on that Hive table. Hive takes care of reading from HBase behind the scenes.
* **Why this path:** it’s the simplest and most reliable path on the Cloudera QuickStart VM (Spark 1.6), because it avoids extra HBase/Spark connector jars and version mismatches.

**Line-by-line explanation**

# task14\_hbase\_via\_hive.py

# Read HBase table via Hive storage handler (Spark 1.6 + HiveContext)

* Just comments: filename and a reminder that we’re using the **Hive storage handler** route on **Spark 1.6** (so we’ll use HiveContext, not the newer SparkSession).

from pyspark import SparkContext

from pyspark.sql import HiveContext

* Import Spark’s **driver** entry point (SparkContext) and the **Hive-aware** SQL context (HiveContext).
  + In Spark 1.6, HiveContext extends SQLContext with Hive features (metastore, HiveQL, SerDes, storage handlers).

sc = SparkContext(appName="Task14\_HBase\_via\_Hive")

* Starts the **driver** JVM with the application name (shows in logs and the Spark UI at port 4040).
* Under the hood this also sets up cluster resources and is required before you do anything with Spark.

hive = HiveContext(sc)

* Wraps that context in a **HiveContext** so Spark SQL can:
  + connect to the **Hive Metastore** (so it can “see” your external table),
  + understand **Hive storage handlers** (like HBase), and
  + run **Hive SQL** statements.

Important: For this to hit the *same* metastore as the Hive CLI, you must launch with the Hive config on the classpath (the hive-site.xml), e.g.:

export HIVE\_CONF\_DIR=/etc/hive/conf.dist

spark-submit \

--conf spark.sql.catalogImplementation=hive \

--conf spark.driver.extraClassPath=/etc/hive/conf.dist \

--conf spark.executor.extraClassPath=/etc/hive/conf.dist \

task14\_hbase\_via\_hive.py

# Ensure we're in the same DB where we created the external table

hive.sql("USE default")

* Tells Hive to use the **default** database (schema).
* Your external table (e.g., hbase\_emp) was created in default, so USE default guarantees we query the correct namespace.

# Read the HBase-backed Hive table

df = hive.sql("SELECT \* FROM hbase\_emp")

* Runs a SQL query **through Hive** to load all rows/columns from your **external table** hbase\_emp.
* Because hbase\_emp is mapped via **HBaseStorageHandler**, Hive reads from **HBase** under the hood and returns a Spark **DataFrame** (df).
* You now have HBase data available as a Spark DataFrame.

print("=== Data from HBase (through Hive) ===")

df.show()

* Prints a label, then df.show() renders the **first 20 rows** in a tabular view in your terminal.
* This is a quick sanity check that the HBase→Hive mapping is working.

print("=== Simple analytics: employees per department ===")

hive.sql("""

SELECT dept, COUNT(\*) AS cnt

FROM hbase\_emp

GROUP BY dept

ORDER BY cnt DESC, dept

""").show()

* A small **aggregation** done in SQL:
  + SELECT dept, COUNT(\*) — compute the number of employees per department.
  + FROM hbase\_emp — read from the HBase-backed Hive table again.
  + GROUP BY dept — group rows by department.
  + ORDER BY cnt DESC, dept — show the biggest departments first; if counts tie, order alphabetically by dept.
* The final .show() prints the aggregated result to the terminal.

sc.stop()

* Cleanly shuts down the SparkContext/driver JVM and releases resources.