**Step 1 — Start your VM**

* Open Oracle VirtualBox → start **Cloudera Quickstart VM**.
* Open a terminal ([cloudera@quickstart ~]$).

**Command to delete the table if it exists**

Open **HBase shell**:

hbase shell

Inside HBase shell:

disable 'employeesnew'

drop 'employeesnew'

* disable 'employeesnew' → you must disable an HBase table before dropping it (safety rule).
* drop 'employeesnew' → deletes the table and all its data.

Then you can safely re-create it as follows:

**Step 2 — Create an HBase table**

hbase shell

create 'employeesnew', 'details'

put 'employeesnew', '101', 'details:name', 'Anita'

put 'employeesnew', '101', 'details:dept', 'HR'

put 'employeesnew', '101', 'details:salary', '52000'

exit

* **Purpose:** Create a table (employees) with a column family (details) and insert a few sample rows.  
    
  Explanation :  
    
  create 'employees', 'details'

create → command to create a new table.

'employees' → table name. This is the logical container for rows (like a table in SQL).

'details' → column family. In HBase, every table must have at least one column family. All the actual columns (name, dept, salary) will belong inside this family.

After this, we have an empty HBase table:

Table: employees

Column family: details

put 'employees', '101', 'details:name', 'Anita'

put → insert (or update) a single cell.

'employees' → target table name.

'101' → row key (here, employee ID). HBase identifies rows by a unique key (like a primary key).

'details:name' → column family and qualifier:

details = column family (like a folder).

name = column qualifier (like a file inside the folder).

'Anita' → value to store in that cell.

Meaning: In row 101, under column family details, qualifier name, store value Anita.

put 'employees', '101', 'details:dept', 'HR'

Same structure as above.

Row key = 101.

Column = details:dept.

Value = HR.

Now row 101 also has a department value stored.

bash

Copy code

put 'employees', '101', 'details:salary', '52000'

Row key = 101.

Column = details:salary.

Value = 52000.

Now row 101 also has a salary.

So, after these three puts, row 101 in table employees looks like this:

Row Key details:name details:dept details:salary

101 Anita HR 52000

**Step 3 — Link HBase table to Hive**

Inside Hive shell:

hive

Then:

CREATE EXTERNAL TABLE employeesnew (

emp\_id STRING,

name STRING,

dept STRING,

salary STRING

)

STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'

WITH SERDEPROPERTIES (

"hbase.columns.mapping"=":key,details:name,details:dept,details:salary"

)

TBLPROPERTIES ("hbase.table.name"="employeesnew");

SELECT \* FROM employeesnew;

exit;

============================================================================

**Purpose:** Expose the HBase table to Hive SQL using HBaseStorageHandler.  
  
Explanation :  
  
What the statement does (high level)

* **Creates a Hive external table** named employees. “External” means Hive only stores **metadata**; the data stays in HBase.
* Tells Hive to access the table **via HBaseStorageHandler** (so rows come from HBase, not HDFS files).
* **Maps** each Hive column to the correct place in HBase: row key and cells inside the details column family.
* Points the mapping at the **existing HBase table** named employees.
* Lets you immediately SELECT \* FROM employees; and get rows back from HBase.

The statement with detailed explanation

CREATE EXTERNAL TABLE employees (

emp\_id STRING,

name STRING,

dept STRING,

salary STRING

)

STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'

WITH SERDEPROPERTIES (

"hbase.columns.mapping"=":key,details:name,details:dept,details:salary"

)

TBLPROPERTIES ("hbase.table.name"="employees");

SELECT \* FROM employees;

exit;

Line-by-line

**CREATE EXTERNAL TABLE employees ( ... )**

* CREATE ... TABLE employees → defines a new **Hive table name** employees in the current Hive database (usually default unless you did USE some\_db).
* EXTERNAL → important keyword. It tells Hive *not* to own the data. If you later run DROP TABLE employees in Hive, the **HBase table and its data remain intact**; only the Hive metadata is dropped.
* The column list:
* emp\_id STRING  
  Logical Hive column for the **row key** in HBase (we’ll map it to :key below).
* name STRING, dept STRING, salary STRING  
  Logical Hive columns for HBase **cells**. Using STRING keeps it simple; you can cast to numeric types in queries if you need math later.

Tip: Hive column order here must match the mapping order you’ll give in hbase.columns.mapping.

**STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'**

* STORED BY → tells Hive **how** to physically access the data.
* 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' → the built-in **storage handler** that translates Hive reads/writes into HBase Scan/Put operations.  
  This is what makes a Hive table “backed by HBase”.

**WITH SERDEPROPERTIES ( "hbase.columns.mapping" = "...")**

* A **SerDe** (Serializer/Deserializer) converts between Hive rows and HBase cells. For HBase, you must give a **column mapping**:
* "hbase.columns.mapping"=":key,details:name,details:dept,details:salary"
* This is a **comma-separated list** mapping **each Hive column (in order)** to a spot in HBase.
* :key → Map the **first Hive column** (emp\_id) to the **HBase row key**.
* details:name → Map name to the **column family** details, **qualifier** name.
* details:dept → Map dept to details:dept.
* details:salary → Map salary to details:salary.

Be precise: the **family name must match exactly** your HBase table’s family (details in your case), and the order must match the Hive column list.

**TBLPROPERTIES ("hbase.table.name"="employees")**

* Points Hive at the **actual HBase table** name: employees.
* Hive will not create or drop this HBase table; it only references it (because we used EXTERNAL).

**SELECT \* FROM employees;**

* Runs a normal Hive query—but under the hood Hive issues an **HBase Scan** and decodes cells according to your mapping.
* You’ll see your HBase rows as plain columns (emp\_id, name, dept, salary).

**exit;**

* Leaves the Hive shell.

**Step 4 — Run Spark job via Hive bridge**

Save your PySpark script as /home/cloudera/task14\_hive\_bridge.py (via winscp).

From VM terminal:

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 \

/home/cloudera/task14\_hive\_bridge.py

* **Purpose:** Run PySpark, connect to Hive, which fetches from HBase.
* Output: analytics (row count, distinct depts, avg salary, salary per dept).