**What we’re doing in Part 2**

Guys, We will treat **Hive** (already inside your Cloudera QuickStart VM) as our **database**.

**Goal:**

1. Put the same employee\_data.csv into a **Hive table** (our “DB table”).
2. Use **PySpark (HiveContext)** to:
   * Read the table into a **DataFrame**
   * Print **details** (schema, row count, preview, basic stats)
   * **Check missing values** column-by-column
   * **Replace** missing values:
     + Numeric (salary) → **median** (robust)
     + Text (department, name, join\_date) → **mode** (most frequent)
   * **Save back** the cleaned data into the **database** as a **new Hive table**
3. Verify the cleaned table exists and looks good.

We’ll also deliberately insert **two rows with missing values** so the “check & replace” step actually does something you can show in class.

# Step 1 — Reset Hive and Create Table

### Open Hive shell:

hive

This starts the Hive command-line tool, where we can type SQL commands to manage tables.

### 1.1 Drop any old table

DROP TABLE IF EXISTS employees;

This deletes the employees table if it already exists in the default database.  
We do this so that we always start fresh.

### 1.2 Create a new table

CREATE TABLE employees (

emp\_id INT,

name STRING,

department STRING,

salary INT,

join\_date STRING

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

STORED AS TEXTFILE;

This makes a new table called **employees** with 5 columns:

* emp\_id → number (employee ID)
* name → text
* department → text
* salary → number
* join\_date → text (we’ll keep it as string for simplicity)

The last two lines mean:

* The table is a **text file**.
* Columns are separated by a **comma** (CSV style).

### 1.3 Load the CSV file

LOAD DATA LOCAL INPATH '/home/cloudera/employee\_data.csv'

INTO TABLE employees;

This takes the CSV file that you already created earlier and copies its contents into the Hive table employees.

* LOCAL → means file is on the local Linux filesystem, not HDFS.
* Path /home/cloudera/employee\_data.csv → that’s where we stored it.
* INTO TABLE employees → loads into our table.

### 1.4 Insert rows with missing values

INSERT INTO TABLE employees VALUES

(109, '', 'IT', NULL, '2018-01-01'),

(110, 'N/A', '', NULL, '');

We are adding two extra rows:

* Row 109 → missing name and salary.
* Row 110 → name is "N/A", department is empty, salary and join\_date missing.

These will help us **test missing value handling**.

### 1.5 Exit Hive

exit;

Leaves the Hive shell, back to Linux terminal.

# Step 2 — PySpark Program

Now we prepare the Python program.

1. Open **Notepad**
2. Copy the code into it.
3. Save as task10\_part2.py.
4. Upload to /home/cloudera/ using WinSCP.

# Execution steps : Step-by-step (simplest path with your current script)

1. **Confirm the table is in Hive (default DB)**

hive -e "USE default; SHOW TABLES;"

You should see employees listed.

1. **Make Spark use the same Hive config**

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

This points Spark at the Quickstart VM’s Hive config. (Using /etc/hive/conf causes the “new empty metastore” problem you saw.)

1. **Run your script with Hive catalog + classpath to Hive config**

spark-submit \

--conf spark.sql.catalogImplementation=hive \

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

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

task10\_part2.py

This makes both the **driver** and **executors** load hive-site.xml, so Spark connects to the same metastore as Hive CLI.

1. **Verify the cleaned table**

hive -e "USE default; SHOW TABLES;"

hive -e "USE default; SELECT \* FROM employees\_clean LIMIT 20;"

You should now see and query employees\_clean.

# Explanation of the above execution commands : 1) Check the table in Hive (default DB)

**Command**

hive -e "USE default; SHOW TABLES;"

**What it does**

* hive starts the Hive CLI.
* -e "…" tells Hive to **execute the quoted SQL then exit** (no interactive shell).
* USE default; switches the session to Hive’s **default database**.
* SHOW TABLES; lists all tables in that database.

**Why we run it**

* To confirm the employees table **exists** in the metastore you expect.
* If you don’t see employees here, Spark won’t see it either (because Spark reads the same metastore when configured correctly).

**2) Point Spark at the *same* Hive configuration**

**Command**

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

**What it does**

* export VAR=VALUE sets an **environment variable** for this shell session.
* HIVE\_CONF\_DIR is where Spark/Hive look for **hive-site.xml** (and friends). That file tells clients **where the metastore is** and what warehouse path to use.

**Why we use this exact path**

* On the Cloudera Quickstart VM, /etc/hive/conf.dist contains the **real, working** Hive config.
* Pointing to the wrong directory (e.g., /etc/hive/conf) can make Spark **create a brand-new, empty metastore** (you will see logs like “Version information not found… Added admin/public role”), so your employees table “disappears.”
* Setting HIVE\_CONF\_DIR here ensures **Spark and Hive CLI read the same metastore**.

**3) Run your Spark job with Hive catalog + classpath to Hive config**

**Command**

spark-submit \

--conf spark.sql.catalogImplementation=hive \

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

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

task10\_part2.py

**What each piece does**

* spark-submit task10\_part2.py launches your Python app.
* --conf spark.sql.catalogImplementation=hive  
  Forces Spark SQL to use the **Hive catalog/metastore** (not the built-in/in-memory one). Your script already uses HiveContext, but this flag makes the intent explicit and avoids mis-detection.
* --conf spark.driver.extraClassPath=/etc/hive/conf.dist  
  Adds /etc/hive/conf.dist to the **driver’s classpath**, so the driver process can **load hive-site.xml**. That’s how it finds the same metastore Hive CLI uses.
* --conf spark.executor.extraClassPath=/etc/hive/conf.dist  
  Does the same for **executors** (the worker JVMs). This keeps driver and executors **in sync** about Hive configs—important when **reading** tables and especially when **writing** (saveAsTable) so everything lands in the same metastore/warehouse.

**What to expect**

* Your script:
  + Runs USE default and reads employees
  + Prints row count, schema, salary stats
  + Fills missing values (avg salary, mode department/name)
  + Writes a new managed Hive table **employees\_clean** (Spark 1.6 writes Parquet by default)
* In the Hive CLI later, you’ll see Parquet-related messages—that’s normal.

**4) Verify the cleaned table in Hive**

**Commands**

hive -e "USE default; SHOW TABLES;"

hive -e "USE default; SELECT \* FROM employees\_clean LIMIT 20;"

**What they do**

* First line: confirms employees\_clean **exists** in the default DB.
* Second line: **queries** the new table and shows up to 20 rows (so you don’t scan everything).

**What “success” looks like**

* employees\_clean appears in SHOW TABLES.
* SELECT \* returns rows where:
  + **salary** blanks/NULLs are replaced by the **average salary** (e.g., 68375.0 as you saw).
  + **department** blanks are replaced by the **most frequent department**.
  + **name** blanks/N/A are replaced by the **most frequent name**.

**Further explanation of “”**spark-submit \ --conf spark.sql.catalogImplementation=hive \ --conf spark.driver.extraClassPath=/etc/hive/conf.dist \ --conf spark.executor.extraClassPath=/etc/hive/conf.dist \ task10\_part2.py**”**

## 1. catalogImplementation

* Spark has to know **where to look for table definitions** (SHOW TABLES, SELECT \* FROM employees, etc.).
* Two options exist in Spark 1.x:
  1. **in-memory (“in-memory” / “in-memory” catalog)**
     + Spark just tracks tables/metadata inside itself.
     + If you stop Spark, those “tables” vanish.
     + No link to Hive.
  2. **Hive (“hive” catalog)**
     + Spark connects to Hive’s **metastore** (the central database where Hive stores schema info).
     + This lets Spark and Hive **share the same tables**.

When we write:

--conf spark.sql.catalogImplementation=hive

we are forcing Spark to use Hive’s catalog/metastore instead of its own temporary one.  
That’s why hive.table("employees") actually works.

## 2. driver

* The **driver** is the “master brain” process of your Spark job.
* It does:
  + Starts Spark.
  + Runs your Python script (task10\_part2.py).
  + Decides what computations should happen.
  + Schedules work to other processes (executors).
  + Collects results and prints them.

In our command:

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

* We are telling the **driver** process:  
  “Hey, also look in /etc/hive/conf.dist for config files (like hive-site.xml) when you run.”
* Without this, the driver wouldn’t know where the Hive metastore is.

## 3. executor

* The **executors** are the worker bees.
* They run on the cluster (in Cloudera Quickstart VM, they just run as local worker processes).
* The driver sends tasks to executors (e.g., “filter these rows”, “aggregate salaries”), and executors actually perform the heavy lifting.

In our command:

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

* We are telling **every executor**:  
  “Also load /etc/hive/conf.dist/hive-site.xml so you know where the Hive metastore and warehouse are.”
* This keeps executors in sync with the driver when reading/writing Hive tables.

### Simple analogy (driver vs executor)

Think of Spark as a **restaurant**:

* **Driver = the head chef** — reads the recipe (your Python code), decides what dishes to make, tells others what to do.
* **Executors = line cooks** — they actually chop, fry, and plate the food (process the data).
* **Catalog implementation** = the **recipe book location**. If you set it to hive, both the head chef and line cooks use the same **Hive recipe book** (the metastore). If you leave it as in-memory, they use a scratchpad that disappears when the kitchen closes.

So in short:

* **catalogImplementation = tells Spark which catalog to use (Hive’s vs temporary in-memory)**
* **driver = Spark’s brain/master process that runs your script**
* **executor = Spark’s worker processes that do the actual data crunching**