**What this program is**

A Spark job that:

1. reads a Hive table default.employees,
2. audits missing data,
3. fills blanks (salary → average; name/department → most frequent),
4. writes a new Hive table default.employees\_clean.

**Imports**

from pyspark import SparkContext

from pyspark.sql import HiveContext

from pyspark.sql.functions import col, when, avg, desc, lit, sum as ssum

* **SparkContext**: boots the Spark driver & connects to the cluster.
* **HiveContext**: Spark SQL entry point *with Hive support* (metastore, saveAsTable, Hive SerDes). In Spark ≥2.0 this became SparkSession(enableHiveSupport()), but on 1.6 this is correct.
* **Functions**:
  + col(...): references a DataFrame column safely (no strings in expressions).
  + when(condition, value) ... otherwise(value): column-wise IF/ELSE.
  + avg: aggregator for the mean.
  + desc: sort direction helper.
  + lit: turn a Python literal into a column.
  + sum as ssum: we alias sum to ssum to avoid clashing with Python’s built-in sum().

**Start Spark + Hive**

sc = SparkContext(appName="Task10\_Part2")

hive = HiveContext(sc)

* Creates a **driver JVM** and executor slots, names the app in the UI.
* HiveContext loads hive-site.xml (so long as it’s on the classpath) to connect to the **same Hive metastore** your hive CLI uses. This is why you exported/pointed classpaths earlier.

**Select DB and read the table**

hive.sql("USE default")

df = hive.table("employees")

* USE default sets the current database in the session.
* hive.table("employees") builds a logical plan from the Hive metastore entry (schema + storage path). No data is read *yet* — Spark is lazy.

**Assumptions required to succeed**

* The table default.employees exists.
* Its columns match the script’s expectations:
  + emp\_id INT
  + name STRING
  + department STRING
  + salary INT (Spark will present it as numeric; after avg it becomes double)
  + join\_date STRING

**Peek at the data**

print("=== Original Data from Hive Table ===")

df.show()

* **Action**: show() triggers a small job to fetch the first 20 rows to the driver and pretty-print them.
* Useful for visual sanity checks (you saw the header-like row — more on that in “Notes & gotchas” at the end).

**Basic counts & schema**

print("Row count:", df.count())

df.printSchema()

* count() is a **full scan** (map-side) across all files; returns a single number.
* printSchema() shows inferred Spark types (post-Hive) — handy to verify that, e.g., salary is numeric.

**Quick stats**

df.describe("salary").show()

* describe computes count/mean/stddev/min/max for the listed columns (nulls ignored).
* **Action**: show() materializes the computation.

**Missing-value audit**

missing = df.select(

ssum(when((col("emp\_id").isNull()) | (col("emp\_id") == ""), 1).otherwise(0)).alias("emp\_id\_missing"),

ssum(when((col("name").isNull()) | (col("name") == "") | (col("name") == "N/A"), 1).otherwise(0)).alias("name\_missing"),

ssum(when((col("department").isNull()) | (col("department") == ""), 1).otherwise(0)).alias("dept\_missing"),

ssum(when((col("salary").isNull()) | (col("salary") == ""), 1).otherwise(0)).alias("salary\_missing"),

ssum(when((col("join\_date").isNull()) | (col("join\_date") == ""), 1).otherwise(0)).alias("join\_date\_missing")

)

print("=== Missing Value Counts ===")

missing.show()

* For each column, we create a **0/1 indicator** (1 if missing), then sum it to count missing rows.
* Checks:
  + isNull() catches actual SQL NULLs.
  + == "" catches empty strings — relevant for string columns loaded from CSV/Text.
  + For name, we also treat literal "N/A" as missing.
* select(...) builds a single-row DataFrame with five counts; show() evaluates it.

Note: Comparing numeric columns (like emp\_id, salary) to "" is harmless but redundant — those are usually NULL rather than "" when text couldn’t parse into INT.

**Imputation (filling in missing data)**

**1) Salary → average**

avg\_salary = df.select(avg("salary")).first()[0]

df\_clean = df.withColumn(

"salary",

when((col("salary").isNull()) | (col("salary") == ""), lit(avg\_salary))

.otherwise(col("salary"))

)

* avg("salary") ignores NULLs and returns a **double**. first()[0] extracts the Python float from a 1-row/1-col result.
* withColumn overwrites salary: if missing → average; else keep original.
* Result type becomes **double** (you saw 68375.0) because the branch inserts a double literal.

**Edge case:** if *all* salaries were NULL, avg\_salary would be None and this would still leave NULLs. Your data isn’t in that state, so you’re fine.

**2) Department → most frequent (mode)**

dept\_mode = df.groupBy("department").count().orderBy(desc("count")).first()[0]

df\_clean = df\_clean.withColumn(

"department",

when((col("department").isNull()) | (col("department") == ""), lit(dept\_mode))

.otherwise(col("department"))

)

* groupBy("department").count() computes frequencies.
* orderBy(desc("count")).first() picks the **mode**. first()[0] extracts the department value (column 0).
* Fill NULL/empty departments with that mode.

**Tie behavior:** If two departments tie for top frequency, Spark returns whichever appears first after the sort (ties are effectively arbitrary unless you add a tiebreaker).

**Important subtlety:** This counts "" as a category; if "" happened to be most frequent, it would become the “mode” and you’d fill blanks with blank — not useful. See “Improvements” below for a safer mode calculation.

**3) Name → most frequent (mode)**

name\_mode = df.groupBy("name").count().orderBy(desc("count")).first()[0]

df\_clean = df\_clean.withColumn(

"name",

when((col("name").isNull()) | (col("name") == "") | (col("name") == "N/A"), lit(name\_mode))

.otherwise(col("name"))

)

* Same pattern as department, but treats N/A as missing too.

**Inspect the cleaned result**

print("=== Cleaned Data Preview ===")

df\_clean.show()

* Triggers a small job; you saw that previously.

**Write back to Hive as a managed table**

hive.sql("DROP TABLE IF EXISTS employees\_clean")

df\_clean.write.mode("overwrite").saveAsTable("employees\_clean")

print("Cleaned data saved to table: default.employees\_clean")

* DROP TABLE IF EXISTS ensures a clean create (defensive; mode("overwrite").saveAsTable in Spark 1.6 would also replace, but explicit drop avoids old metadata quirks).
* saveAsTable("employees\_clean"):
  + **Catalog**: registers the table in the Hive metastore under default.employees\_clean.
  + **Storage**: writes data files (by default **Parquet**) under the warehouse dir (e.g., /user/hive/warehouse/employees\_clean).
  + **Schema**: persisted with the table so future Spark/Hive sessions read it consistently.

**Shutdown**

sc.stop()

* Tidies up executors and the Spark UI process.

**What you observed in Hive afterwards**

Your SELECT \* FROM employees\_clean returned rows with filled values (e.g., 68375.0 for salary). That confirms:

* Spark could **read** from the original Hive table,
* perform transforms,
* and **write** a new Hive-managed table that Hive CLI can query — i.e., your Spark ↔︎ Hive integration is correct.

**Note (why that “header-ish” row appears)**

* Because the original Hive table was created as TEXTFILE and you **loaded a CSV** likely *with a header line*, Hive stored that header as a real data row. That’s why you saw something like:
* NULL name department 68375.0 join\_date
  + emp\_id: couldn’t parse the word emp\_id into INT ⇒ becomes **NULL**.
  + name: literal text name.
  + department: literal text department.
  + salary: the string salary became NULL, then your imputation filled it with the **average** 68375.0.
  + join\_date: literal text join\_date.

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