Detailed Explanation of “task10\_part1.py” program :  
  
===================================================

**Top imports (what and why)**

from pyspark import SparkContext

* **What:** Imports the core Spark driver handle for Spark 1.x.
* **Why:** You need a SparkContext to talk to the Spark engine (create RDDs, submit work).

from pyspark.sql import SQLContext, Row

* **SQLContext**: Adds DataFrame/SQL features on top of SparkContext (Spark 1.x style).
* **Row**: A small helper to create row objects with named fields (like a tiny record/struct).

from datetime import datetime, date

* **What:** Standard Python date utilities.
* **Why:** To parse join\_date and compute “years with company.”

from pyspark.sql.functions import col, avg, when

* **col("name")**: Refers to a DataFrame column in expressions.
* **avg("salary")**: Built-in aggregation function to compute average.
* **when(condition, value)**: SQL-style conditional builder for derived columns.

**Step 1 — Start Spark (Spark 1.6 style)**

sc = SparkContext(appName="Task10\_Part1")

* **Creates** the Spark driver called **Task10\_Part1**.
* This launches the internal services (scheduler, memory manager, etc.).
* In QuickStart, this typically runs in **local mode** on the VM.

sqlContext = SQLContext(sc)

* Wraps your SparkContext with **SQL/DataFrame** features.
* In Spark 2.x you’d use SparkSession; in **Spark 1.6**, SQLContext is correct.

**Step 2 — Read the CSV (from local disk, not HDFS)**

raw\_rdd = sc.textFile("file:///home/cloudera/employee\_data.csv")

* **Reads** the file as raw lines into an **RDD** (Resilient Distributed Dataset).
* file:/// tells Spark: **use local filesystem** (your CentOS disk), not HDFS.
* Result: each element of raw\_rdd is one text line (string).

header = raw\_rdd.first()

* **Action** (triggers a small job): returns the first line of the file.
* Here, the header is: emp\_id,name,department,salary,join\_date.

data\_rdd = raw\_rdd.filter(lambda line: line != header)

* **Transformation**: removes the header line from the data.
* lambda creates a tiny anonymous function; it keeps only lines that are not equal to the header.

Note: This is a **simple CSV reader** (good for our academic lab). For real CSVs (commas in names, quotes), you’d use a CSV library.

**Step 3 — Parse lines into typed rows**

def parse\_line(line):

parts = line.split(",")

return Row(

emp\_id=int(parts[0]),

name=parts[1],

department=parts[2],

salary=int(parts[3]),

join\_date=parts[4]

)

* **Splits** each line by comma into fields.
* **Casts** emp\_id and salary to int.
* Puts values into a **Row** with named fields (this gives a schema later).

rows\_rdd = data\_rdd.map(parse\_line)

* **Transformation**: applies parse\_line to every data line → RDD of Row objects.

**Step 4 — Create a DataFrame (table in memory)**

df = sqlContext.createDataFrame(rows\_rdd)

* Converts the RDD of Rows into a **DataFrame** (columns + schema).
* Spark infers schema from the Row field names and Python types.

print("=== Original Data ===")

df.show()

* print(...): plain Python message for clarity in console.
* df.show(): **Action** that triggers execution and prints the first 20 rows in a nice table.

Tip: df.printSchema() is handy if you want to see column types.

**Step 5 — Department average salary**

dept\_avg = df.groupBy("department").agg(avg("salary").alias("dept\_avg\_salary"))

* **groupBy("department")**: groups rows by department.
* **agg(avg("salary"))**: computes the average salary per group.
* **alias("dept\_avg\_salary")**: renames the result column.

print("=== Department Average Salary ===")

dept\_avg.show()

* Shows a tiny table like:

+----------+---------------+

|department|dept\_avg\_salary|

+----------+---------------+

|HR | 50000.0 |

|IT | 79333.33 |

|Finance | 79500.0 |

...

**Step 6 — Join the average back to each employee row**

df2 = df.join(dept\_avg, "department")

* **Join key**: "department".
* Default is an **inner join** (every row in df finds its dept match in dept\_avg).
* Now every row has its own columns + the department’s dept\_avg\_salary.

**Step 7 — “Salary Increase” (% vs dept avg)**

df2 = df2.withColumn(

"Salary\_Increase",

((col("salary") - col("dept\_avg\_salary")) / col("dept\_avg\_salary")) \* 100

)

* **withColumn(new\_name, expression)**: adds or replaces a column.
* Formula:

Salary\_Increase=salary−dept\_avgdept\_avg×100\text{Salary\\_Increase} = \frac{salary - dept\\_avg}{dept\\_avg} \times 100Salary\_Increase=dept\_avgsalary−dept\_avg​×100

* If salary is above the average, this is **positive**; below average → **negative**.
* col("...") references columns; math is vectorized (runs on entire column, distributed).

Note: In Spark 1.6 this yields a double/float automatically due to division by a double.

**Step 8 — “YearswithCompany” (custom UDF)**

def years\_since(dstr):

try:

dt = datetime.strptime(dstr, "%Y-%m-%d").date()

return float((date.today() - dt).days) / 365.25

except:

return None

* **Parses** join\_date with format YYYY-MM-DD.
* Computes today - join\_date in days, divides by **365.25** (accounts for leap years approx).
* Returns a float (years), or None if parsing fails.

from pyspark.sql.types import DoubleType

from pyspark.sql.functions import udf

years\_udf = udf(years\_since, DoubleType())

* **Registers** the Python function as a Spark **UDF** producing DoubleType.
* Spark will **ship** this function to executors and run it row-by-row.

df2 = df2.withColumn("YearswithCompany", years\_udf(col("join\_date")))

* Calls the UDF on each row’s join\_date and makes a new column.

Tip: Built-in date functions (when available) are **faster** than UDFs. We used a UDF here for Spark-1.6 compatibility and simplicity.

**Step 9 — “Salary\_Category” (Low / Medium / High)**

df2 = df2.withColumn(

"Salary\_Category",

when(col("salary") < 55000, "Low")

.when((col("salary") >= 55000) & (col("salary") < 80000), "Medium")

.otherwise("High")

)

* **when(condition, value)** chains work like IF ... ELSE IF ... ELSE.
* Thresholds are **simple, teacher-friendly** (you can change them):
  + < 55,000 → **Low**
  + 55,000–79,999 → **Medium**
  + ≥ 80,000 → **High**

**Step 10 — Show final result (screen only)**

print("=== Final Result ===")

df2.select("emp\_id", "name", "department", "salary", "dept\_avg\_salary",

"Salary\_Increase", "YearswithCompany", "Salary\_Category").show()

* select(...) reorders/chooses the columns to display.
* .show() triggers the computation and prints the table (first 20 rows by default).

**Step 11 — Save result as a local CSV (single file)**

(df2.select("emp\_id", "name", "department", "salary", "dept\_avg\_salary",

"Salary\_Increase", "YearswithCompany", "Salary\_Category")

.coalesce(1) # ensures one CSV file instead of many

.write

.mode("overwrite")

.format("com.databricks.spark.csv")

.option("header", "true")

.save("file:///home/cloudera/employee\_result\_out"))

* **Why “coalesce(1)”**: Spark normally writes one file **per partition**. coalesce(1) reduces to a single partition → a single CSV file.
  + Great for small lab data. Don’t do this for huge datasets.
* **.write.mode("overwrite")**: replace any existing output folder.
* **.format("com.databricks.spark.csv")**: The Spark-CSV data source (bundled in your QuickStart).
* **.option("header","true")**: write column names as the first row.
* **.save("file:///...")**: write to **local disk** (not HDFS) at /home/cloudera/employee\_result\_out.
  + Spark creates a **directory** with a file like part-00000-...csv.

If you prefer **no saving**, you can delete this whole Step 11 block.

**Clean shutdown**

sc.stop()

* Politely stops the SparkContext and frees resources. Prevents stray Java processes and port locks.