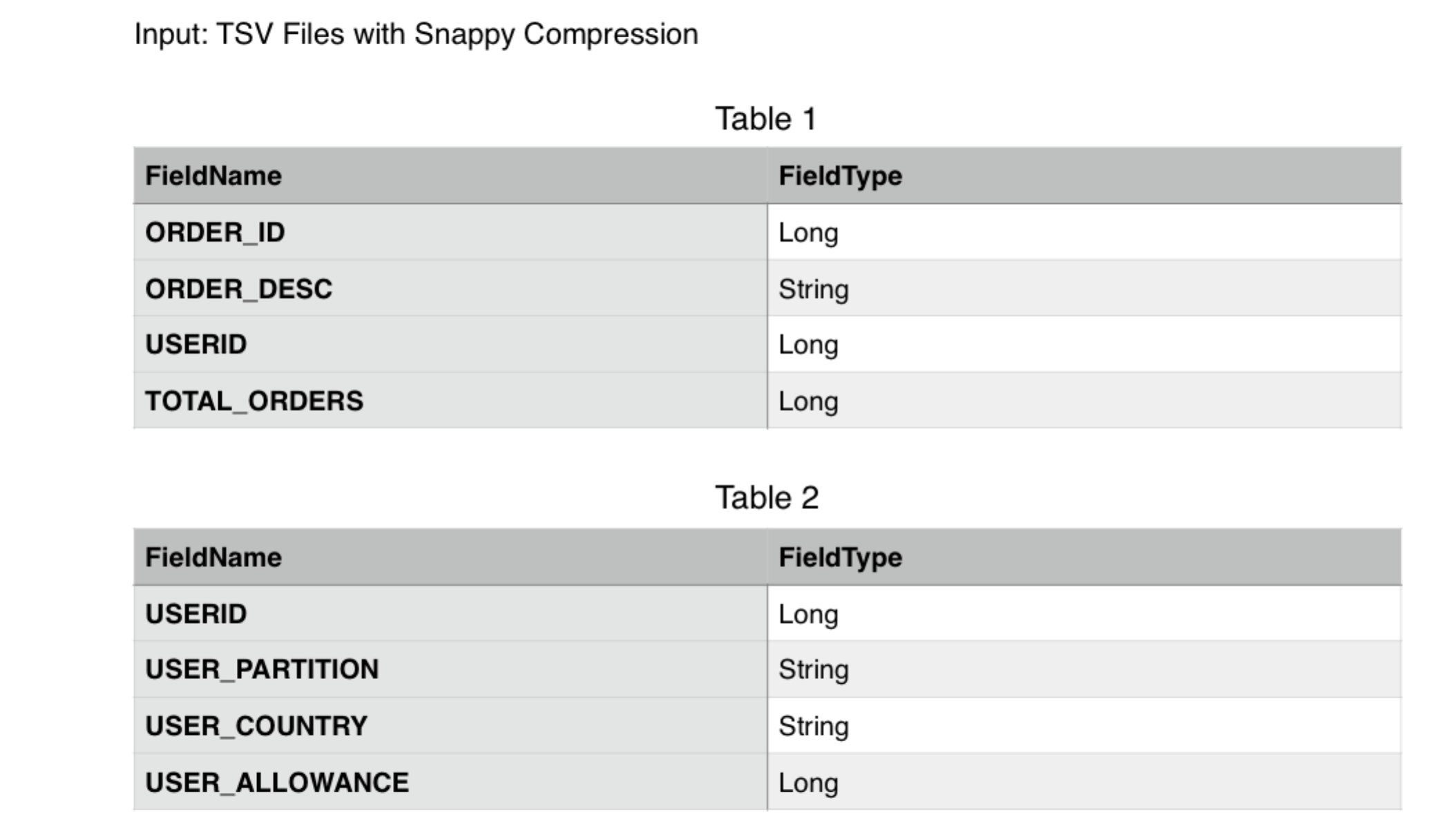
Input : Order TSV , User TSV

Order TSV Dataset Size ~ 1000 GB

User TSV Dataset Size ~ 10 GB



**Approach:**

Here we leverage the DataFrame to take benefits of Tungesten memory optimization and Catalyst query optimization.

We can always compare SQL Data Frame approach with RDD approach where we create > partitionedRDD = inputRDD.repartitionAndSortWithinPartitions(new KeyBasePartitioner(100)) and then use

> outputRDD = partitionedRDD.mapPartitions(..)

Time being, we are using DataFrame

The large tables need to be partitioned.

Enabling BroadcastHashJoin  optimizes joining a large and a small table

Spark automatically figures out the smaller table (saved by dataframe) which need to be used for BroadcastHashJoin

The advantage of clustering by key is that Spark will use SortMergeJoin

And will avoid **hashpartitionin** shuffles

The other params to be tuned appropriately :

spark.sql.autoBroadcastJoinThreshold: 1GB

spark.driver.maxResultSize

spark.driver.memory

spark.executor.memory

spark.memory.offHeap.enabled

spark.memory.offHeap.size

spark.default.parallelism

spark.dynamicAllocation.enabled

spark.dynamicAllocation.minExecutors

spark.akka.frameSize

spark.akka.threads

\*\* Note it’s subject to some RND – to figure out if we get better performance

>> by first DISTRIBUTING the user table by USER\_PARTITION or directly repartitioning by USER\_PARTITION and then caching the data and then JOINING

>> or by directly CLUSTERING the user table by USER\_ID (the current approach) and then follow other steps…

**Code:**

conf.set("spark.sql.shuffle.partitions”, 4)

conf.set("spark.hadoop.io.compression.codecs",  “snappy”)

SQLContext sqlContext = new SQLContext(sc);

DataFrame orderDF = sqlContext.read()

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

.option("inferSchema", "true")

.option("header", "true")

.option("delimiter","\t")

.load("orders.csv”);

orderDF.repartition(1000).registerTempTable("orders\_table")

DataFrame usersDF = sqlContext.read()

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

.option("inferSchema", "true")

.option("header", "true")

.option("delimiter","\t")

.load("users.csv”);

usersDF.write.saveAsTable("users\_table")

val usersDFDist = sqlContext.sql("SELECT \* FROM users\_table CLUSTER BY USERID ")

usersDFDist.registerTempTable("users\_dist")

sqlContext.sql("CACHE TABLE users\_dist”)

// do this only if partitions are under allowed size to avoid OOM

sqlContext.sql("SELECT \* FROM orders\_table CLUSTER BY USERID").registerTempTable("sorted\_order\_table")

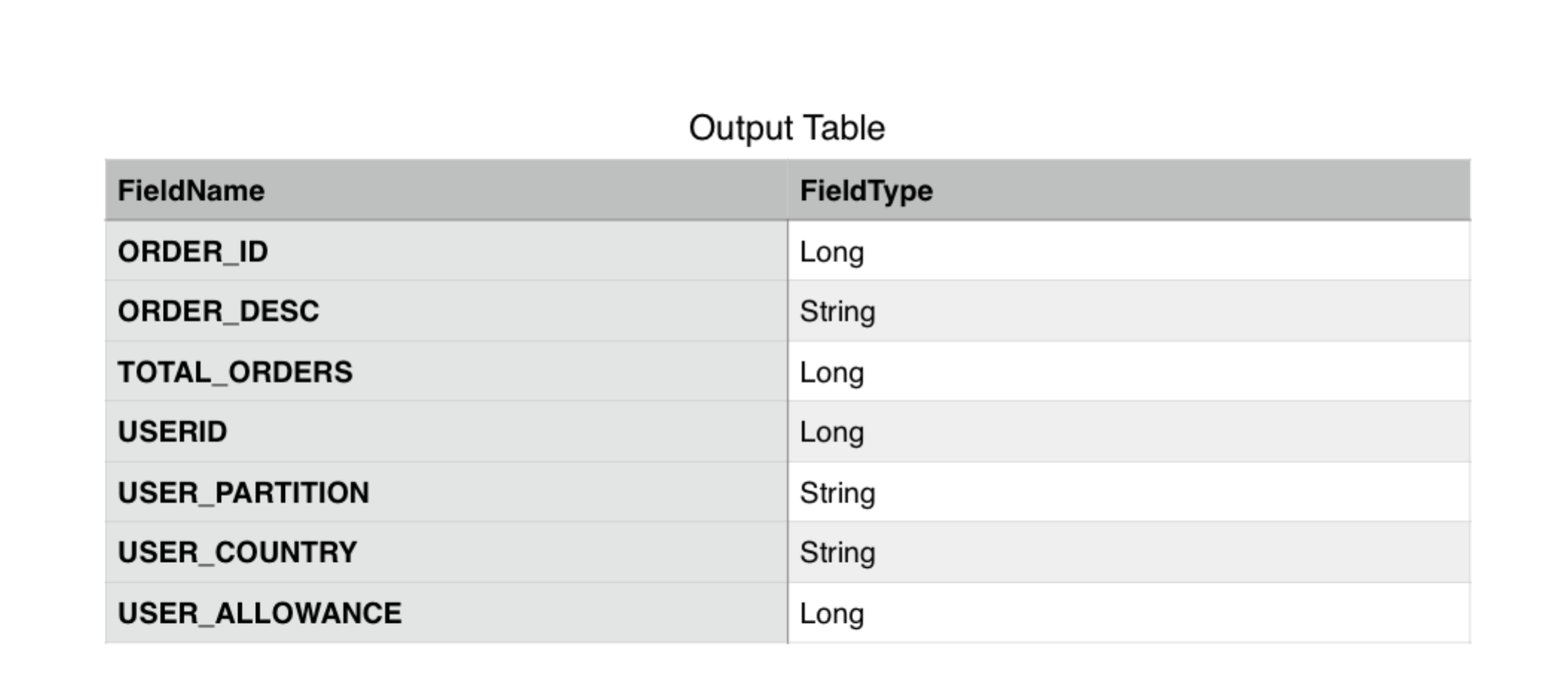
val joinedDF = sqlContext.sql("select ORDER\_ID, ORDER\_DESC, TOTAL\_ORDERS, users\_dist.USERID, USER\_PARTITION, USER\_COUNTRY, USER\_ALLOWANCE from sorted\_order\_table join users\_dist on orders\_table.userid = users\_dist.userid")

joinedDF.write.format("parquet").save("data.parquet")

// remove the tables

**%sql% DROP** **TABLE** **IF** **EXISTS** users\_table

**%sql% DROP** **TABLE** **IF** **EXISTS** sorted\_order\_table



**Other considerations**

\*\*\*\*

**Fault-Tolerant:**

Once Spark is deployed in YARN / MESOS , the above application automatically becomes Fault-Tolerant.

The parquet result should be saved in hdfs / s3 for fault-tolerance

\*\*\*\*\*

**Reliable:**

We should collect counters and other metrics about Job / Task execution

We should enable applicable logic to check data consistency (session context validation , data de-duplication)

\*\*\*\*\*\*

Scalable

If we set

spark.dynamicAllocation.enabled = true

spark.dynamicAllocation.minExecutors = N

then Spark will dynamically allocate executors.

The Spark Pipeline can be further auto-scaled and better managed if orchestrated via Apache Beam