Data Migration with Spark

Vineet Kumar

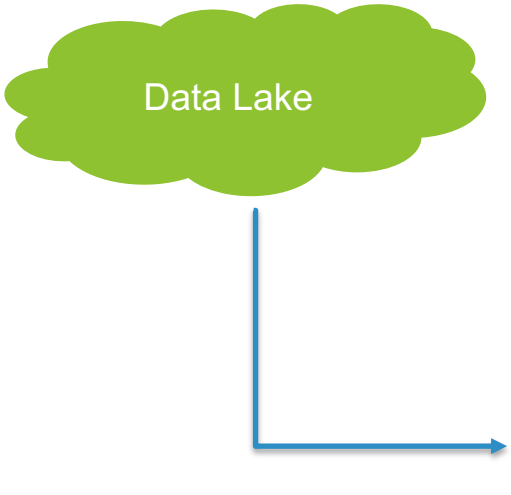
#UnifiedAnalytics #SparkAISummit

Why Data Migration?

- Business requirement - Migrate Historical data into data lake for analysis - This may require some light transformation.
- Data Science - This data can provide better business insight. More data -> better models -> better predictions
- Enterprise Data Lake - There may be thousands of datafiles sitting in multiple source systems.
- EDW - Data from EDW for archival purpose into the lake.
- Standards - Store data in with some standards- For example: partition strategy, storage formats- Parquet, Avro etc.

1000s of files
- Text Files
- XML
- JSON

- EDW
- Data Archive



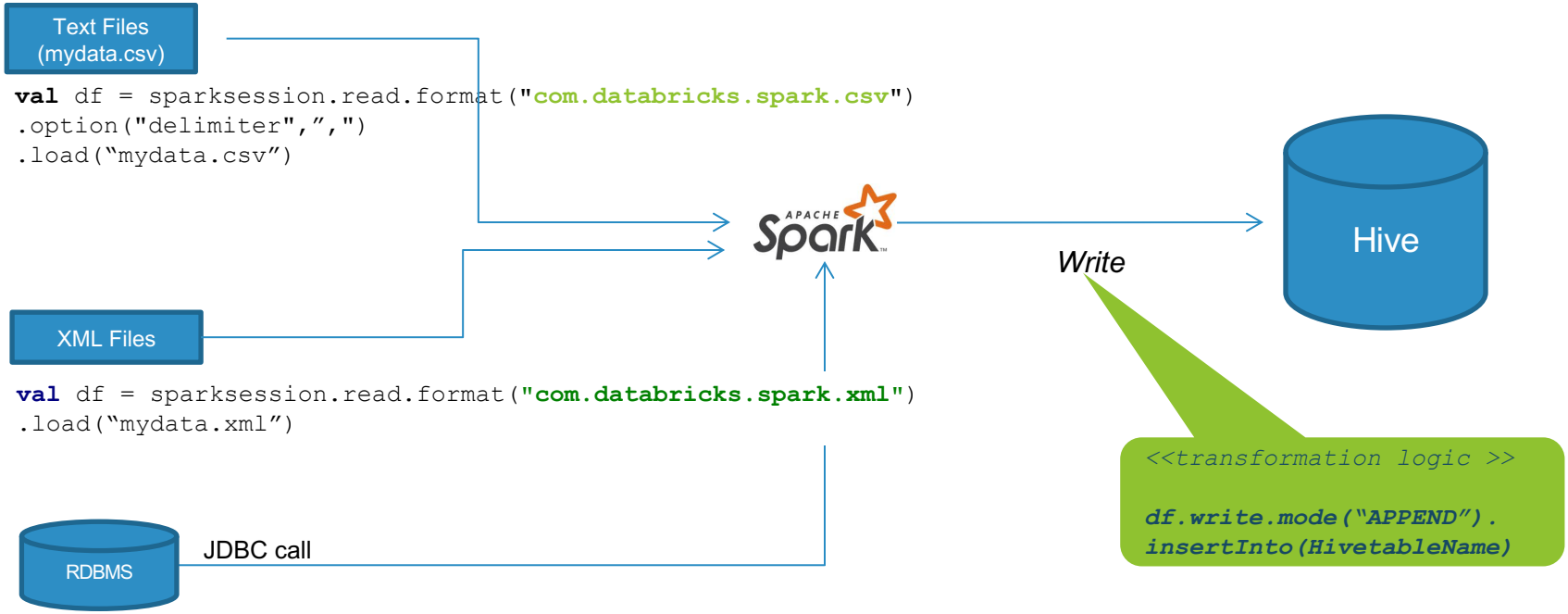
Migration Issues

- Schema for text files - header missing or first line as a header.
- Data in files is neither consistent an nor in target standard format.
  - Example – timestamp, Hive standard format is yyyy-MM-dd hh24:mi:ss
- Over the time file structure(s) might have changed: some new columns added or removed.
- There could be thousands of files with different structure- Separate ETL mapping/code for each file ???
- Target table can be partitioned or non-partitioned.
- Source data size can from few megabytes to terabytes, or from a few columns to thousands of columns.
- Partitioned column can be one of the existing columns or custom column based on the value passed as an argument.
  - partitionBy=partitionField, partitionValue=<<Value>>
  - Example : partitionBy='ingestion_date' partitionValue='20190401'

Data Migration Approach

Create Spark/Hive context

```
val sparksession = SparkSession
      .builder()
      .enableHiveSupport()
      .getOrCreate()
```



Wait… Schema ?

XML, JSON or RDBMS sources

- Schema present in the source.
- Spark can automatically infer schema from the source.

```
scala> df.columns
  res02: Array[String] = Array(id, name, address)
```

Text Files – how to get schema?

- File header is not present in file.
  - Can Infer schema, but what about column names?
- Data from last couple of years and the file format has been evolved.

```
scala> df.columns
      res01: Array[String] = Array(_c0, _c1, _c2)
```

Schema for Text files

- Option 1 : File header exists in first line
- Option 2: File header from external file – JSON
- Option 3: Create empty table corresponds to csv file structure
- Option 4: define schema - StructType or Case Class

Schema for CSV files : Option 3

- Create empty table corresponds to text file structure
  - Example – Text file : file1.csv

```
1|John|100 street1,NY|10/20/1974
```

  - Create Hive structure corresponds to the file : file1_structure

```
create table file1_raw(id:string, name:string, address:string, dob:timestamp)
```

  - Map Dataframe columns with the structure from previous step

```
val hiveSchema = sparksession.sql("select * from file1_raw where 1=0")
val dfColumns = hiveSchema.schema.fieldNames.toList
// You should check if column count is same before mapping it
val textFileWithSchemaDF = df.toDF(dfColumns: _*)
```

Before - Column Names :

```
scala> df.show()
+---+-----+--------------+----------+
|_c0|  _c1|           _c2|       _c3|
+---+-----+--------------+----------+
|  1| John|100 street1,NY|10/20/1975|
|  2|Chris|Main Street,KY|10/20/1975|
|  3|Marry|park Avenue,TN|10/20/1975|
+---+-----+--------------+----------+
```

After :

```
scala> df2.show()
+---+-----+--------------+----------+
|id|  name|           address|       dob|
+---+-----+--------------+----------+
|  1| John|100 street1,NY|10/20/1975|
|  2|Chris|Main Street,KY|10/20/1975|
|  3|Marry|park Avenue,TN|10/20/1975|
+---+-----+--------------+----------+
```

Dates/timestamp ?

- Historical files - Date format can change over time.

Files from year 2004:

```
1|John|100 street1,NY|10/20/1974
2|Chris|Main Street,KY|10/01/1975
3|Marry|park Avenue,TN|11/10/1972
```

…

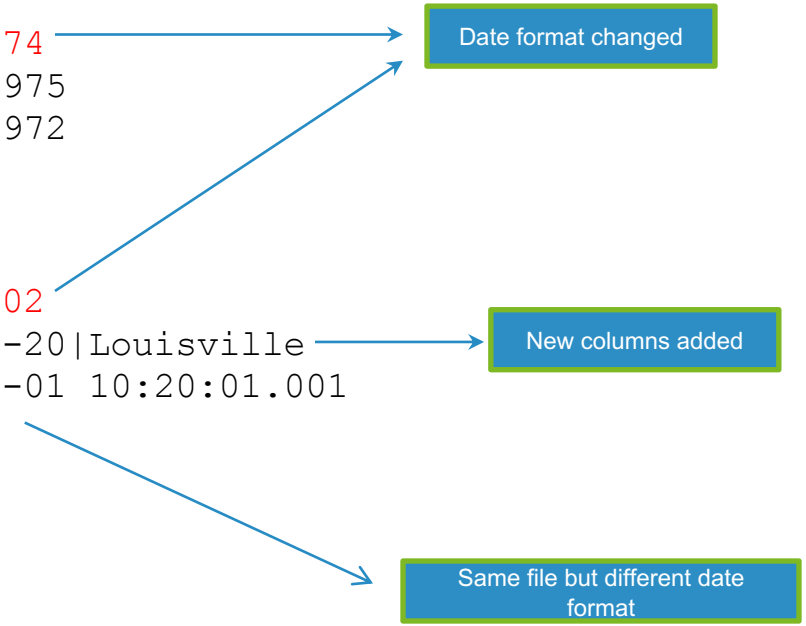
…

Files from year 2018 onwards:

```
1|John|100 street1,NY|1975-10-02
2|Chris|Main Street,KY|2010-11-20|Louisville
3|Marry|park Avenue,TN|2018-04-01 10:20:01.001
```

Files have different formats:

```
File1- dd/mm/yyyy
File2 – mm/dd/yyyy
File3 – yyyy/mm/dd:hh24:mi:ss
….
```



Timestamp columns

- Target Hive Table:

id: int,
name string,
dob timestamp,
address string,
move_in_date timestamp,
rent_due_date timestamp
PARTITION COLUMNS…

Timestamp Column – Can be at any location in target table. Find these first

```
for (i <- 0 to (hiveSchemaArray.length - 1)) {
      hiveschemaArray.toString.contains("Timestamp")
      }
```

Transformation for Timestamp data

- Hive timestamp format - yyyy-MM-dd HH:mm:ss
- Create UDF to return valid date format. Input can be in any formats

```
val getHiveDateFormatUDF=udf(getValidDateFormat)
```

UDF Logic for Date transformation

```
import java.time.LocalDateTime
import java.time.format.{DateTimeFormatter, DateTimeParseException)

val inputFormats=Array(“MM/dd/yyyy”, ”yyyyMMdd”, …….)
val validHiveFormat= DateTimeFormatter.ofPattern(“yyyy-MM-dd HH:mm:ss”)
val inputDate=“10/12/2019”

for (format <- inputFormats) {
try {
      val dateFormat = DateTimeFormatter.ofPattern(format)
       val newDate= LocalDateTime.parse(inputDate, dateFormat)
       newDate.format(validHiveFormat)
      }
catch e : DateTimeParseException =>null
}
```

This can build from a file as an argument

Parameter to a function

Date transformation – Hive format

- Find columns with timestamp data type and apply udf on those columns

```
for (i <- 0 to (hiveSchemaArray.length - 1)) {
  if (hiveschemaArray(i).toString.contains("Timestamp")) {
    val field=schemaArray(i).toString.replace("StructField(","").split(",")(0)
    val tempfield = field + "_tmp"
    val tempdf = df.withColumn(tempfield,getHiveDateFormatUDF(col(field)))
                 .drop(field).withColumnRenamed(tempfied, field)
    val newdf = tempdf
                                                        }
```

Before applying UDF :

```
scala> df2.show()
+---+-----+--------------+----------+
|id|  name|          address|       dob|
+---+-----+--------------+----------+
|  1| John|100 street1,NY|10/20/1975|
|  2|Chris|Main Street,KY|10/20/1975|
|  3|Marry|park Avenue,TN|10/20/1975|
+---+-----+--------------+----------+
```

After applying date UDF :

```
scala> newDF.show()
+---+-----+--------------+----------+
|id|  name|          address|       dob|
+---+-----+--------------+----------+
|  1| John|100 street1,NY|1975-10-20 00:00:00|
|  2|Chris|Main Street,KY|1975-10-20 00:00:00|
|  3|Marry|park Avenue,TN|1975-10-20 00:00:00|
+---+-----+--------------+----------+
```

Column/Data Element Position

- Spark Dataframe(df) format from text file: name at position 2, address at position 3

```
id| name| address| dob
------------------------------
1|John|100 street1,NY|10/20/1974
```

```
df.write.mode("APPEND").insertInto(HivetableName)
```

Or

```
val query = "INSERT OVERWRITE INTO hivetable SELECT * FROM textFileDF"
Sparksession.sql(query)
```

- Hive table format : address at position 2
  - name switched to address, and address to name

```
id| address| name| dob
------------------------------------------------------------
1,John, 100 stree1 NY, Null
```

Column/Data Element Position

- Relational world

```
INSERT INTO hivetable(id,name,address)
SELECT id,name,address from <<textFiledataFrame>>
```

- Read target hive table structure and column position.

//Get the Hive columns with position from target table

```
val hiveTableColumns= sparksession.sql("select * from hivetable where 1=0").columns
val columns = hiveTableColumns.map(x=> col(x))
```

// select columns source data frame and insert into target table.

```
dfnew.select(columns:_*).write.mode("APPEND").insertInto(tableName)
```

Partitioned Table.

- Target Hive tables can be partitioned or non-partitioned.

```
newDF.select(columns:_*).write.mode("APPEND").insertInto(tableName)
```

- Partitioned by daily, monthly or hourly.
- Partitioned with different columns.

  Example:

  Hivetable1 – Frequency daily – partition column – load_date
  Hivetable2 – Frequency monthly – partition column – load_month

- Add partitioned column as last column before inserting into hive

```
newDF2 = newDF.withColumn("load_date", lit(current_timestamp()))
```

Partitioned Table.

- Partition column is one of the existing field from the file.

```
val hiveTableColumns= sparksession.sql("select * from hivetable where1=0").columns
val columns = hiveTableColumns.map(x=> col(x))
newDF.select(columns:_*).write.mode("APPEND").insertInto(tableName)
```

- Partition is based on custom field and value passed as an argument from the command line.

```
spark2-submit arguments : partionBy="field1", partitionValue="1100"
```

- Add Partition column as a last column in Dataframe

```
newDF2 = newDF.withColumn(partitionBy.trim().toLowerCase(), lit(partitionValue))
```

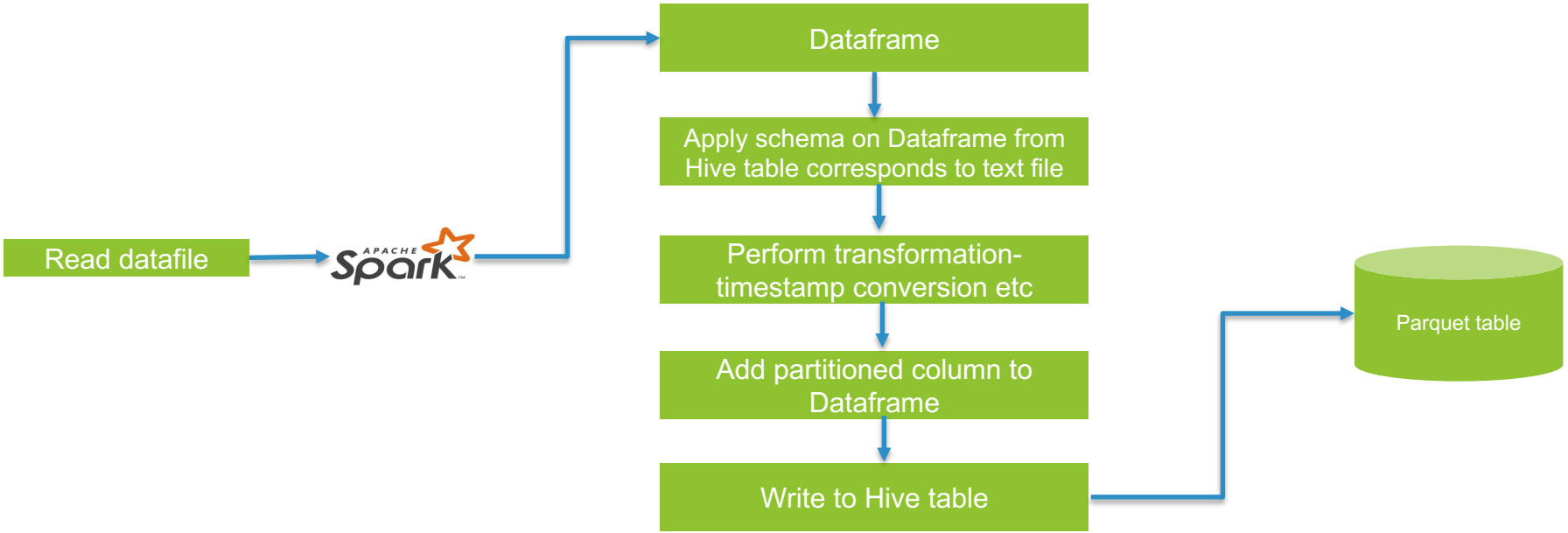
- Final step: before inserting into hive table

```
newDF2.select(columns:_*).write.mode("APPEND").insertInto(tableName)
```

Performance: Cluster Resources

- Migration runs in it's own Yarn Pool
- Large (Files Size >10GB, or with 1000+ Columns)
  # repartition size = min(fileSize(MB)/256,50)
  # executors
  # Executor-cores
  # executor-size
- Medium(File Size : 1 – 10GB, or with < 1000 Columns)
  # repartition size = min(fileSize(MB)/256,20)
  # executors
  # executor-cores
  # executor-size
- Small
  # executors
  # Executor-cores
  # executor-size

Data Pipeline



SPARK+AI SUMMIT 2019

DON'T FORGET TO RATE AND REVIEW THE SESSIONS

SEARCH SPARK + AI SUMMIT

Download on the App Store

GET IT ON Google Play