In-memory data pipeline and warehouse at scale using Spark, Spark SQL, Tachyon and Parquet

Ema Iancuta

iorhian@gmail.com

Radu Chilom

radu.chilom@gmail.com

Buzzwords Berlin - 2015

Atigeo™

- Big data analytics / machine learning
- 6+ years with Hadoop ecosystem
- 2 years with Spark
- http://atigeo.com/

BIG DATA
Research Group

- A research group that focuses on the technical problems that exist in the big data industry and provides open source solutions
- http://bigdataresearch.io/

Agenda

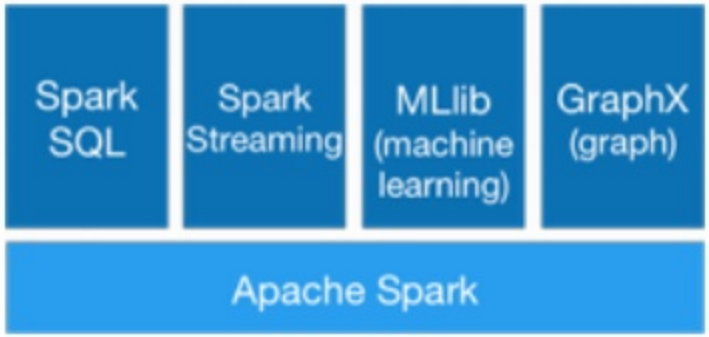
- Intro
- Use Case
- Data pipeline with Spark
- Spark Job Rest Service
- Spark SQL Rest Service (Jaws)
- Parquet
- Tachyon
- Demo

Use Case

- Build an in memory data pipeline for millions financial transactions used downstream by data scientists for detecting fraud
- Ingestion from S3 to our Tachyon/HDFS cluster
- Data transformation
- Data warehouse

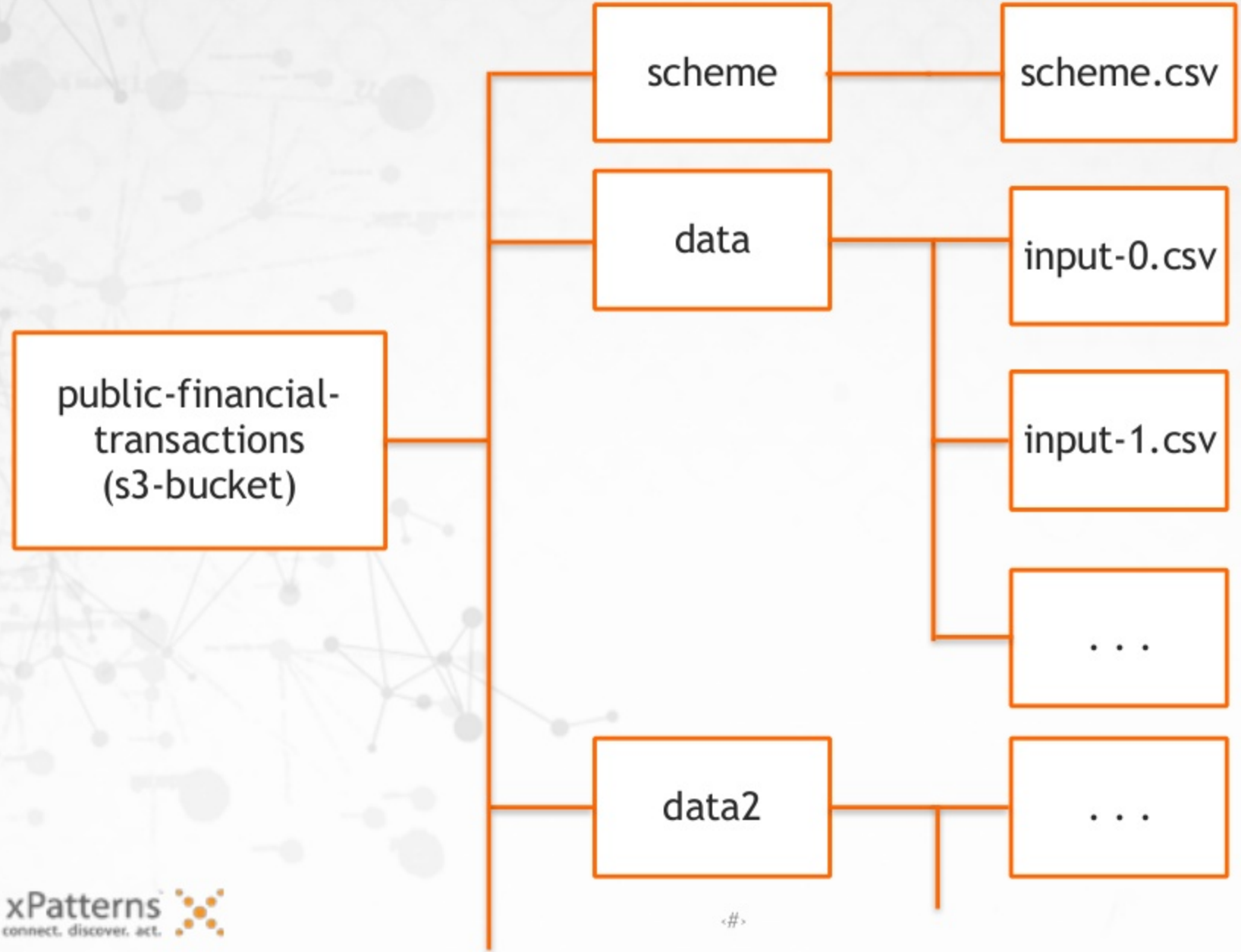
Apache Spark

- “fast and general engine for large-scale data processing”
- Built around the concept of RDD
- API for Java/Scala/Python (80 operators)



- powers a stack of high level tools including Spark SQL, MLlib, Spark Streaming.

Public S3 Bucket: public-financial-transactions



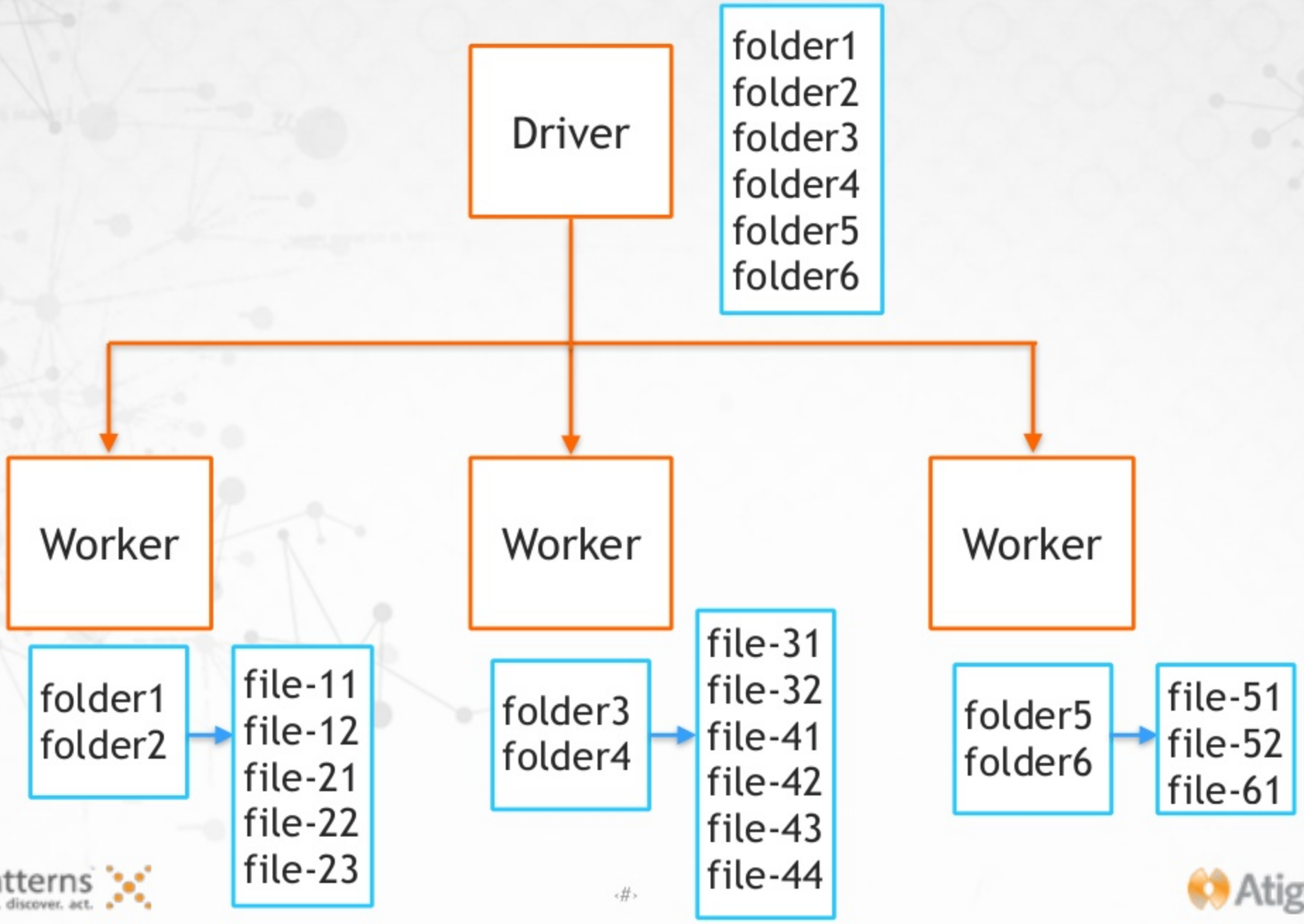
1. Ingestion

- Download from S3

```
sc.wholeTextFiles("s3a://public-financial-transactions/*/*")
```

- Resolving the wildcards means listing files metadata
  - Listing the metadata for a large number of files from external sources can take a long time

Listing the metadata (distributed)



Listing the metadata (distributed)

```
//Get folder list
val listObjectsRequest = new ListObjectsRequest()
  .withBucketName(bucketName).withPrefix("").withDelimiter("/")
val folderList = s3Client.listObjects(listObjectsRequest)
  .getCommonPrefixes

//Get files RDD
val folderRdd = sc.parallelize(folderList)
val filesRdd = folderRdd.flatMap{ folder =>
  getFilesFromFolder(bucketName, folder)
}
```

- For fine tuning, specify the number of partitions

```
val folderRdd = sc.parallelize(folderList, numPartitions)
```

Download Files

```
val results = files.map { file =>
  val s3Client = S3Utils.getS3Client()
    S3Utils.downloadFile(bucketName, file, outputFolder, s3Client)
}
```

- Unbalanced partitions

Unbalanced partitions

Partition 0	Partition 1
transactions.csv	input.csv data.csv values.csv buzzwords.csv buzzwords.txt

Balancing partitions

Partition 0

(0, transactions.csv)
(2, data.csv)
(4, buzzwords.csv)

Partition 1

(1, input.csv)
(3, values.csv)
(5, buzzwords.txt)

Balancing partitions

- Balancing partitions

```
var filesWithIndexRdd = filesRdd.zipWithIndex().map {
  case (value, index) => (index, value)
}
filesWithIndexRdd = filesWithIndexRdd.repartition(numPartitions)
```

Keep in mind that repartitioning your data is a fairly expensive operation.

2. Data Transformation

- Data cleaning is the first step in any data science project

- For this use-case:

- Remove lines that don't match the structure
- Remove “useless” columns
- Transform data to be in a consistent format

Find Country char code

Numeric Format	Alpha 2 Format	Name
276	DE	Germany

- Join

```
import org.apache.spark.SparkContext._
val finalRdd = normalizedRdd.join(countries).map   {
  case (k: String, (columns: ListBuffer[String], charCode: String)) => {
    columns(4) = charCode
    columns
  }
}
```

- Problem with skew in the key distribution

Metrics for Join

Summary Metrics for 20 Completed Tasks

Metric	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	8 ms
Duration	1 s	2 s	17 s
Time spent fetching task results	0 ms	0 ms	0 ms
Scheduler delay	42 ms	47 ms	57 ms
Shuffle Read (Remote)	321.7 KB	1778.5 KB	79.6 MB

Find Country char code

- Broadcast Country Codes Map

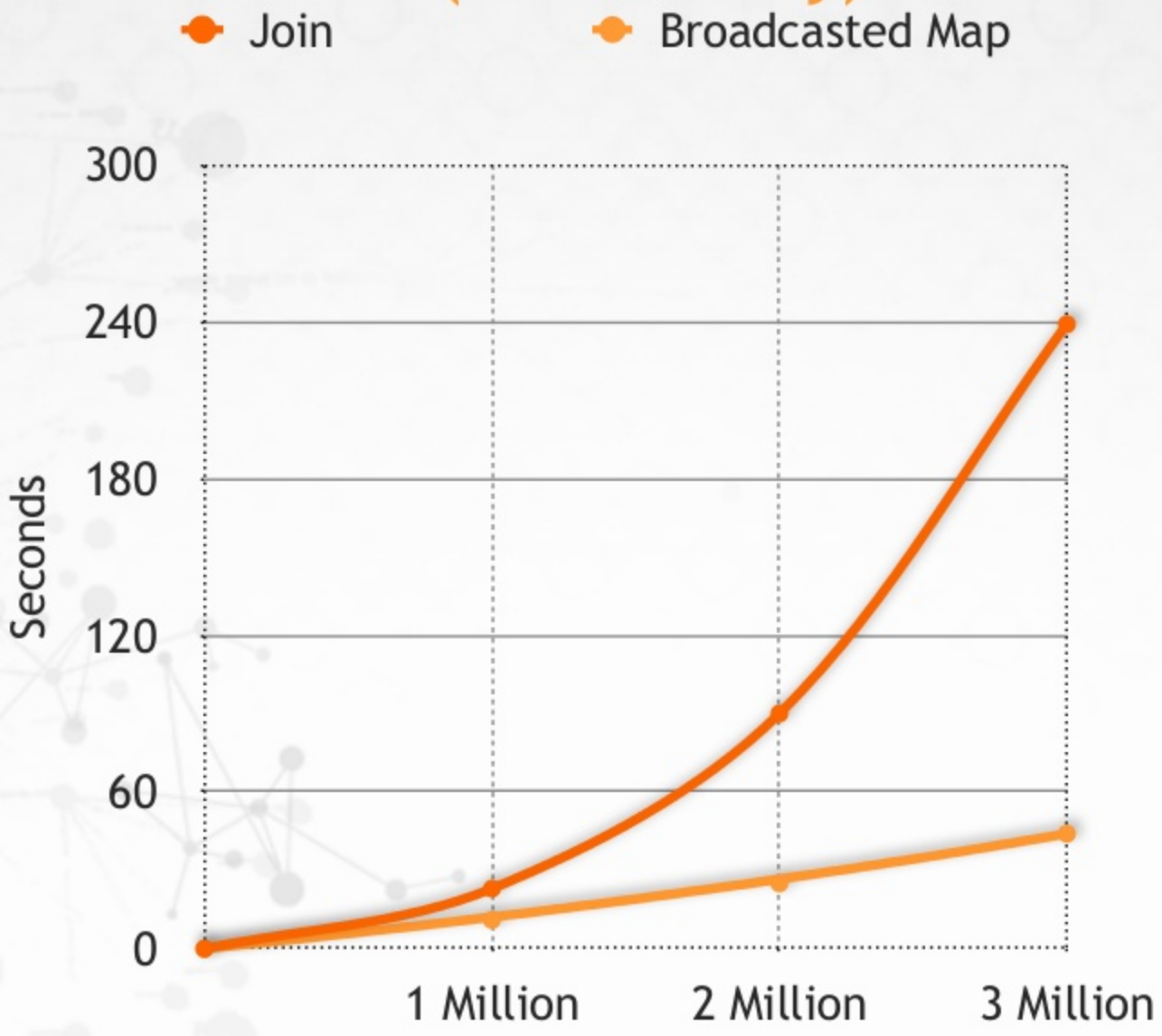
```
val countries: Map[String, String] = countriesRdd.collectAsMap()
val countriesBroadcast = sc.broadcast(countries)

val structuredRdd = normalizedRdd.flatMap { array =>
  countriesBroadcast.value.get(array(4)) match {
    case None => Nil
    case Some(value) => {
      array(4) = value
      List(array)
    }
  }
}
```

Metrics

Metric	Median	75th percentile	Max
Result serialization time	0 ms	1 ms	1 ms
Duration	7 s	7 s	8 s
Time spent fetching task results	0 ms	0 ms	0 ms
Scheduler delay	0.1 s	0.2 s	0.2 s
Input	36.6 MB	36.6 MB	36.6 MB

Transformation with Join vs Broadcasted Map (skewed key)



Spark-Job-Rest

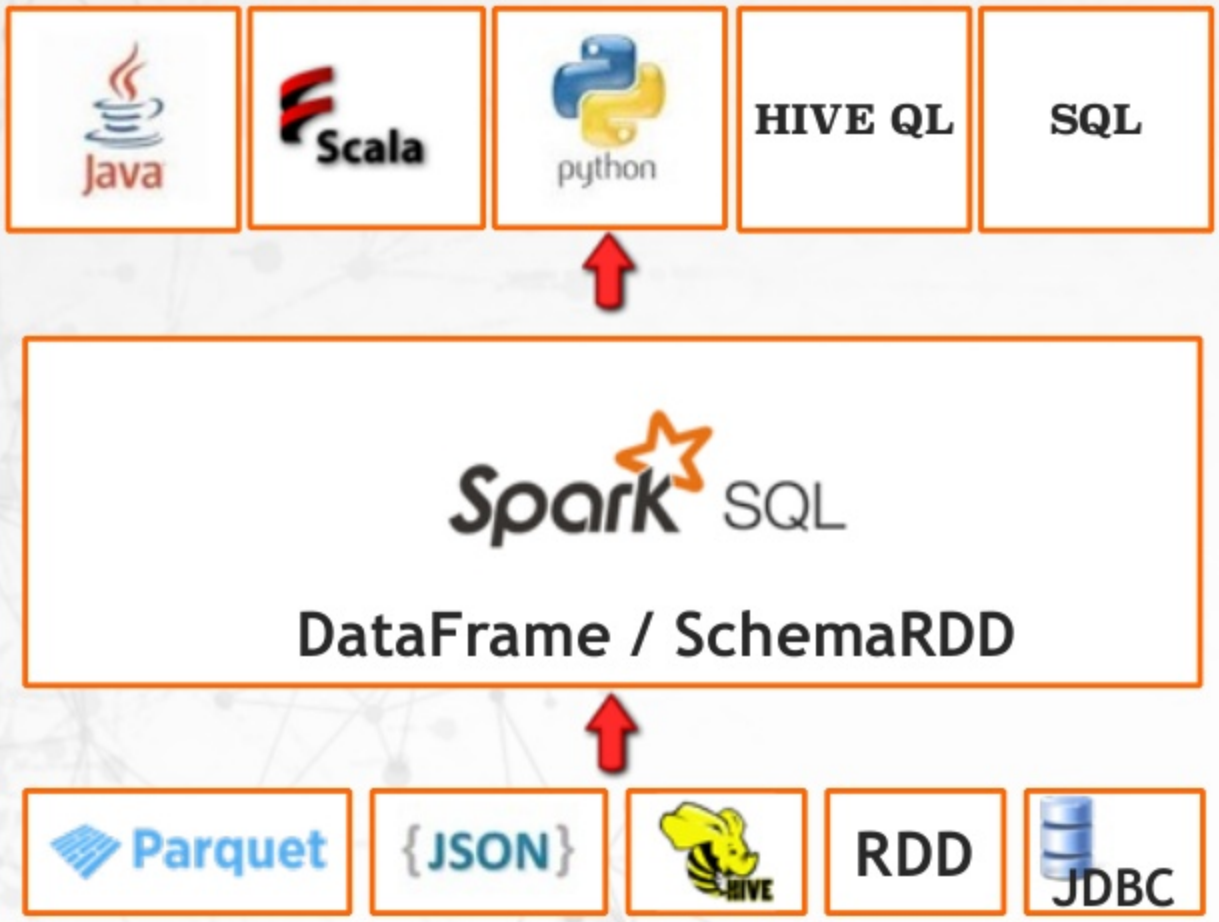
https://github.com/Atigeo/spark-job-rest

- Supports multiple contexts
- Launches a new process for each Spark context
- Inter-process communication with Akka actors
- Easy context creation & job runs
- Supports Java and Scala code
- Friendly UI

Build a data warehouse

- Hive
- Apache Pig
- Impala
- Presto
- Stinger (Hive on Tez)
- Spark SQL

Spark SQL



- Rich language interfaces
- RDD-aware optimizer
- Support for multiple input formats

Creating a data frame

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val schemeArray = sc textFile("/user/ubuntu/normalized/scheme.txt") collect
val schemeFields = schemeArray (0) split(",") map(fieldName =>
                        StructField(fieldName, StringType, true))
val scheme = StructType(schemeFields)

//read the data into an RDD[Array[String]] an then into a RowRDD
val transactionsRDD = sc textFile("/user/ubuntu/normalized/data") map (_.split(","))
val transactionsRowRdd = transactionsRDD map (Row.fromSeq(_))

val transactionsDataFrame = sqlContext.createDataFrame(transactionsRowRdd, scheme)
```

Explore data

Perform a simple query:

> Directly on the data frame

- select
- filter
- join
- groupBy
- agg
- join
- count
- sort
- where ..etc.

```
transactionsDataFrame groupBy("CUSTOMER") count() show()
```

> Registering a temporary table

```
transactionsDataFrame registerTempTable("transactionsTemp")
val cmd = "select CUSTOMER, count (*) from transactionsTemp group by CUSTOMER"
sqlContext.sql(cmd) show
```

Creating a data warehouse

```
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
hiveContext.sql("CREATE EXTERNAL TABLE transactions (ID String, SITE String,
                                                     ..............................
                                PRCCARD_IP String, PRCCARD_REG_DATETIME String)
                ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
                LOCATION '/user/ubuntu/normalized_even/data'")
```

```
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
val hiveTransactionsDF = hiveContext.createDataFrame(transactionsRowRdd, scheme)
hiveTransactionsDF.saveAsTable("myTransactions")
```

```
val tachyonOutput = "tachyon://masterip:19998/user/ema/outParquet"
transactionsDataFrame.saveAsParquetFile(tachyonOutput)
```

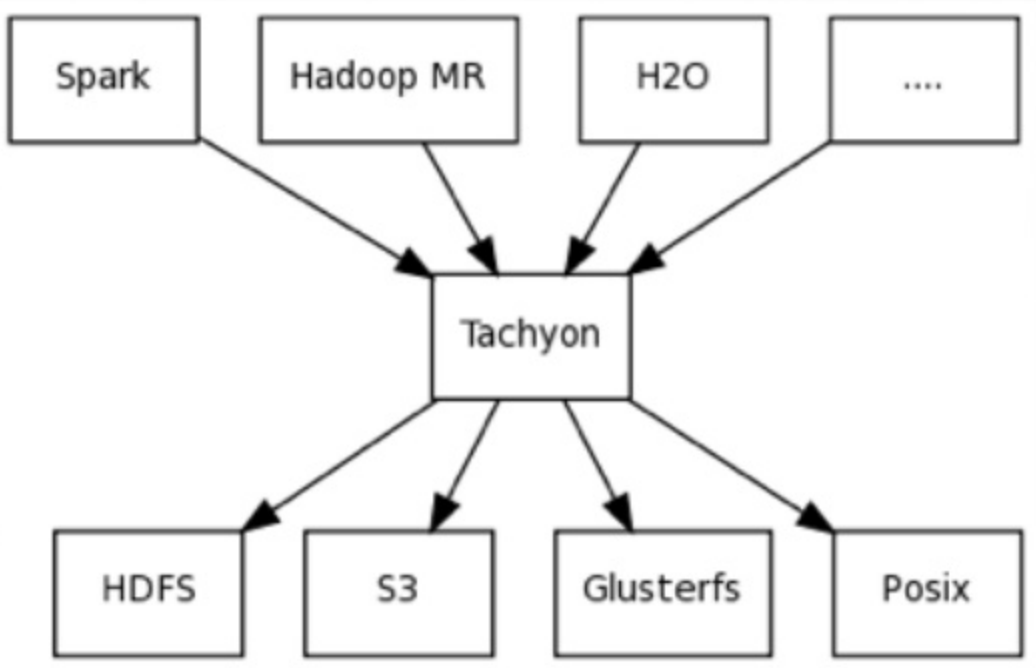
https://github.com/Atigeo/xpatterns-spark-parquet

File Formats

- TextFile
- SequenceFile
- RCFile (RowColumnar)
- ORCFile (OptimizedRowColumnar)
- Avro
- **Parquet**
  - > columnar format
  - > good for aggregation queries
  - > only the required columns are read from disk
  - > nested data structures
  - > schema with the data
  - > spark sql supports schema evolution
  - > efficient compression

Tachyon

- memory-centric distributed file system enabling reliable file sharing at memory-speed across cluster frameworks
- Pluggable underlayer file system: hdfs, S3,…



Caching in Spark SQL

```
hiveContext.cacheTable("transactions")
```

```
transactionsDataFrame.cache()
```

```
hiveContext.sql("CACHE TABLE transactions")
```

- Cache data in columnar format
- Automatically compression tune

Spark cache vs Tachyon

- spark context might crash
- GC kicks in
- share data between different applications

Jaws spark sql rest

- Highly scalable and resilient data warehouse

- Submit queries concurrently and asynchronously

- Restful alternative to Spark SQL JDBC having a interactive UI

- Since Spark 091 with Shark

- Support for Spark SQL and Hive - MR (and more to come)

https://github.com/Atigeo/jaws-spark-sql-rest

Jaws main features

- Akka actors to communicate through instances
- Support cancel queries
- Supports large results retrieval
- Parquet in memory warehouse
- returns persisted logs, results, query history
- provides a metadata browser
- configuration file to fine tune spark

Code available at

https://github.com/big-data-research/in-memory-data-pipeline

Q & A