LynxIT Big Data Platform architecture on AWS Ecosystem

|  |  |
| --- | --- |
| Created Date:  Updated Date:  Author:  Version Number:  Status: | 20 July 2015  14 January 2016  Bart Ochirbat, Richard Long  0.3  Draft |

**Disclaimer:** This paper and its content is the intellectual property of LynxIT, A Dialog Group Company. 2015. All Rights Reserved.

All materials contained in this paper may not be redistributed, transmitted or displayed without its express prior permission of LynxIT, A Dialog Group Company.

Contents

[Part A - LynxIT Big Data Platform 3](#_Toc440532841)

[A.1 Synopsis 3](#_Toc440532842)

[A.2 LynxIT Big Data Platform on AWS ecosystem 3](#_Toc440532843)

[A.2.1 LynxIT Big Data ETL framework by using Spark – (For heavy transformation on-the-fly) 4](#_Toc440532844)

[A.2.2 LynxIT Big data ETL framework components: 4](#_Toc440532845)

[A.2.3 LynxIT Big data ETL architecture 4](#_Toc440532846)

[A.2.4 The below is a CLI example. 6](#_Toc440532847)

[A.3 LynxIT Big Data Platform on an isolated private cloud 10](#_Toc440532848)

[A.3.1 Mesos 10](#_Toc440532849)

[A.3.2 TACHYON 10](#_Toc440532850)

[A.4 Building predictive analytics in Spark ML 11](#_Toc440532851)

[A.5 Building Real time analytics framework on AWS 16](#_Toc440532852)

[A.5.1 Real world use cases 16](#_Toc440532853)

[A.5.2 LynxIT Real-time analytics architecture 20](#_Toc440532854)

[A.5.3 Capturing real-time streaming by using Apache NiFi 20](#_Toc440532855)

[A.6 Experience and Work with extremely large data sets at massive scale 21](#_Toc440532856)

[A.6.1 Launch EMR cluster with Spark 22](#_Toc440532857)

[A.6.2 Analysing Web Archive (WARC) File 23](#_Toc440532858)

[A.6.3 Actual Implementation in Spark 26](#_Toc440532859)

[Part B - Building ETL Server using AWS Native Services 28](#_Toc440532860)

[B.1 Our Design Target 28](#_Toc440532861)

[B.2 Three Editions 28](#_Toc440532862)

[B.3 Three Deployment Options: 29](#_Toc440532863)

[B.4 How to choose from 3 Editions and 3 Deployment Options 30](#_Toc440532864)

[B.5 Main Functions we have realised in our prototype: 31](#_Toc440532865)

[Part C - Appendix A 32](#_Toc440532866)

[C.1.1 How to configure a production-ready Mesosphere Cluster on Ubuntu 32](#_Toc440532867)

[C.1.2 Glossary of Terms 32](#_Toc440532868)

[Part D - References 33](#_Toc440532869)

[Part E - Big Data Project Team 33](#_Toc440532870)

# LynxIT Big Data Platform

## Synopsis

This paper describes the LynxIT Big data platform which includes running big data frameworks on an isolated private cloud using Mesos and a public cloud on AWS infrastructure.

Mesos is an open source platform for fine-grained resource sharing in an isolated cluster. Potentially you can turn your entire data centre into one huge computer.

There is a clear demand for deploying and running big data frameworks such as Hadoop and Spark on “on-premises” infrastructure due to various reasons. For instance, in the finance industry, financial institutions are reluctant to use a public cloud due to strict compliance and regulatory requirements, data confidentiality and security concerns.

On the other hand, a public cloud model has many benefits such as ultimate scalability, cost effectiveness and reliability.

Therefore, the LynxIT Big Data platform offers running big data frameworks on both “on-premises” isolated private cloud and a public cloud on AWS.

## LynxIT Big Data Platform on AWS ecosystem

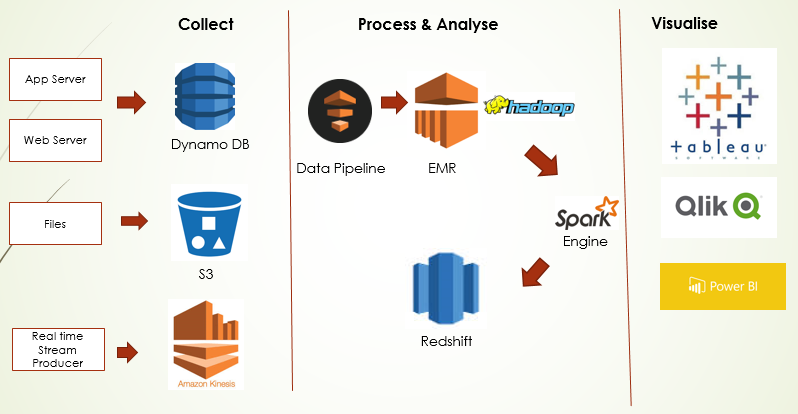


Figure 2 – LynxIT Big Data Platform on AWS ecosystem

### LynxIT Big Data ETL framework by using Spark – (For heavy transformation on-the-fly)

#### Overview

LynxIT Big Data ETL framework is a viable and end-to-end solution because it incorporates every aspect of ETL development life cycle including storage techniques for the large data set, resource and job scheduling, code development environment and compiling into executable distribute application (JAR file).

This ETL framework is built-on with Spark. Spark is a general purpose in-memory framework.

Spark is more suitable for heavy ETL transformation. Spark in general provides a broader set of capabilities than Redshift because it has APIs in general-purpose languages (Java, Scala, Python) and libraries.

### LynxIT Big data ETL framework components:

* **AWS Data Pipeline** - Job and resource scheduling
* **AWS Elastic Map Reduce** - Managed Hadoop cluster on AWS
* **AWS S3** - Storing input data
* **Apache Parquet** – a column storage format
* **Apache Spark** – General purpose in-memory data processing framework
* **Scala** – Functional programming language that natively supports Spark framework
* **Scala IDE for Eclipse** – Code Development
* **Apache Maven** for compiling ETL application into executable JAR files (Java )
* **AWS Redshift** – Petabyte-scale data warehouse

### LynxIT Big data ETL architecture

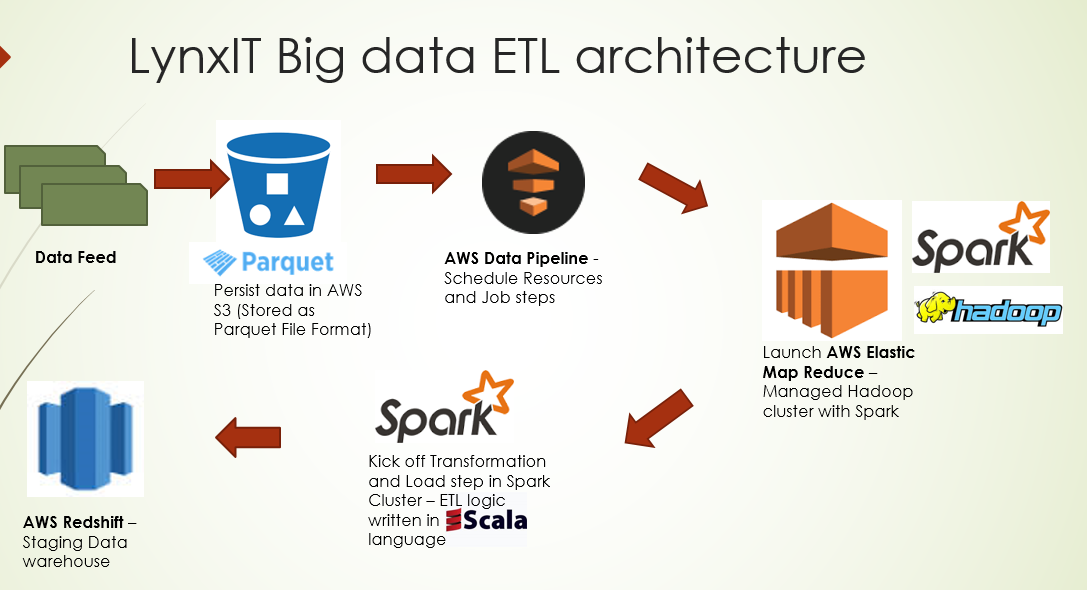


Figure 3 – Big Data ETL architecture on AWS Ecosystems

Amazon EMR is a web service that makes it easy and, fast and cost-effective to process vast amounts of data. Amazon EMR can be used as an Extract and Transformation tool.

The advantage of using Amazon EMR is that, it allows to define and allocate AWS resources such as spot instances and a number of required nodes etc. As soon as ETL process has been completed, we can terminate EMR instances.

#### The Data set storage

Amazon S3 can be used as a Data Lake or Landing zone. During our prototype design, we have utilised a parquet file format for data storage.

Apache Parquet is a columnar storage format and can be used in any project in the Hadoop ecosystem. A parquet file format is a binary, compressed, efficient format that provides better performance storage.



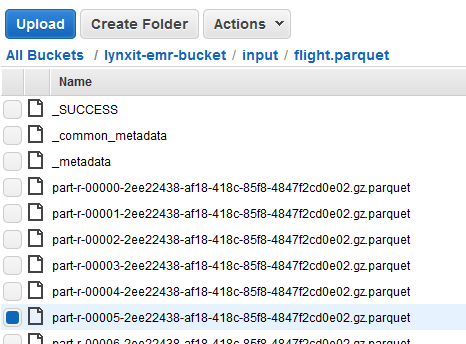


Figure 4 - Parquet files in S3

#### Job and Resource scheduling with AWS Data pipeline

AWS Data Pipeline is a web service that helps you orchestrate workflow, move, integrate and process data across AWS services.

With AWS Data Pipeline, you can define data-driven workflows, so that tasks can be dependent on the successful completion of previous tasks.

By using AWS Pipeline, you can do the following:

* Manages and orchestrates your work flow
* Automates the management of your AWS resources
* Manages dependencies and automated scheduling
* Setup notification and alerts using Amazon Simple Notifications Services (on Job Failure/Success)

You can build and deploy AWS Data Pipeline from a web console, CLI (command line interface) and API’s such as AWS Java SDK, AWS Python SDK and AWS Ruby SDK.

### The below is a CLI example.

##### Define

aws datapipeline create-pipeline –name MyETL –unique-id token

##### Import

aws datapipeline put-definition -pipeline-id-df-1gh2454d8d1f87r2s –pipeline-definition /home/repo/etl.json

##### Activate

aws datapipeline activate-pipeline –pipeline-id- df-1gh2454d8d1f87r2s

##### Quickly Diagnose Failure

Turn on logging:

“enableDebugging”, “logUri”, “emrLogUri”

Namespace your logs:

“s3://#{LOGS\_BUCKET}/#{@s3prefix}/#{START\_TIME}/SalesByDayEMRLogs”

Log into dev instances

Clean Up

“terminateAfter”: “6 hours”

#### LynxIT Big Data ETL Development Environment

We have setup the following for the development environment of the prototype

* **Scala** – Functional programming language that natively supports Spark framework. The below table shows Scala vs Java.

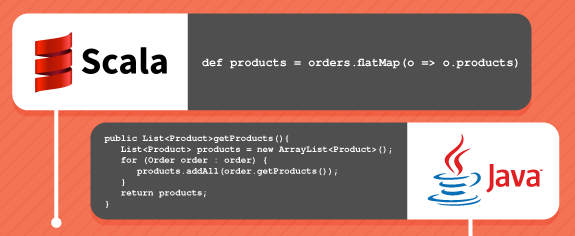


Figure 5 – Scala vs Java

* **Scala IDE for Eclipse** – Code Development
* **Apache Maven** for compiling ETL application into executable JAR files

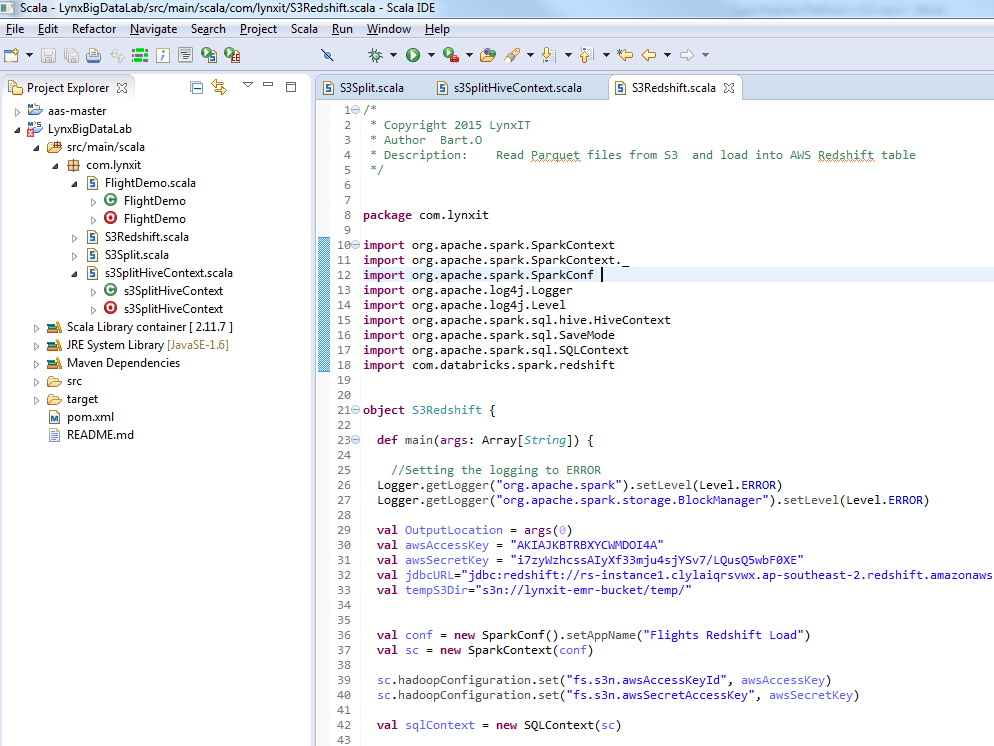


Figure 6 – Scala IDE

#### Source Control with GitHub

GitHub is a Web-based Git repository hosting service.

##### Clone Git Repository

git **clone** http:*//github.com/lynxit/BigData.git*

#### Compiling Scala application with Maven

Apache Maven is a software project management based on the concept of a project object model. You can manage all your application dependency with Maven project.

There was a limitation, compiling large application with Eclipse Windows. Our recommendation is to use AWS Linux instance to compile large applications with Maven.

##### **Install Maven with on AWS Linux instance**

$ sudo wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo

$ sudo sed -i s/\$releasever/6/g /etc/yum.repos.d/epel-apache-maven.repo

$ sudo yum install -y apache-maven

$ mvn --version

##### **Build Application with Maven**

$ mvn clean package appassembler:assemble

##### **Adding third party jar file to local Maven Repository**

mvn install:install-file -Dfile=C:\Users\ochirbatb\Documents\MyScalaApps\LynxBigDataLab\Jars\warcbase-0.1.0-SNAPSHOT-fatjar.jar -DgroupId=warcbase -DartifactId=warcbase -Dversion=0.1.0 -Dpackaging=jar

Also we will need to update POM.xml file.

<dependency>

<groupId>warcbase</groupId>

<artifactId>warcbase</artifactId>

<version>0.1.0</version>

</dependency>

#### Debugging Scala application with Zeppelin

Zeppelin is a web based notebook and an ideal debugging tool for Scala applications with Apache Spark.

* Automatic SparkContext and SQLContext injection
* Runtime jar dependency loading from local filesystem or maven repository.
* Canceling job and displaying its progress

##### Loading dependency JAR file in Zeppelin

%dep

z.load("/mnt/warcbase-0.1.0-SNAPSHOT-fatjar.jar")

#### LynxIT Big data ETL Engine

Apache Spark is used for this ETL framework. Apache Spark is a general purpose in-memory data processing framework.

* Spark is a successor of Map Reduce
* Up to 100 times faster than Hadoop Map Reduce. Please see the below table. Source : <https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html>

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Hadoop World Record** | **Spark 100 TB \*** | **Spark 1 PB** |
| **Data Size** | 102.5 TB | 100 TB | 1000 TB |
| **Elapsed Time** | 72 mins | 23 mins | 234 mins |
| **# Nodes** | 2100 | 206 | 190 |
| **# Cores** | 50400 | 6592 | 6080 |
| **# Reducers** | 10,000 | 29,000 | 250,000 |
| **Rate** | 1.42 TB/min | 4.27 TB/min | 4.27 TB/min |
| **Rate/node** | 0.67 GB/min | 20.7 GB/min | 22.5 GB/min |
| **Environment** | dedicated data center | EC2 (i2.8xlarge) | EC2 (i2.8xlarge) |

Figure 7 – Hadoop vs Spark benchmark

#### Summary of ETL Framework

* Up to 100 times faster than Hadoop Map Reduce
* In-memory, fault tolerant data processing
* Highly scalable ETL framework on AWS infrastructure
* More suitable for heavy ETL transformation
* Rich built-in libraries
* Incorporates every aspect of ETL development life cycle including large data set storage technique, resource and job scheduling, code development environment and compiling executable application.

## LynxIT Big Data Platform on an isolated private cloud

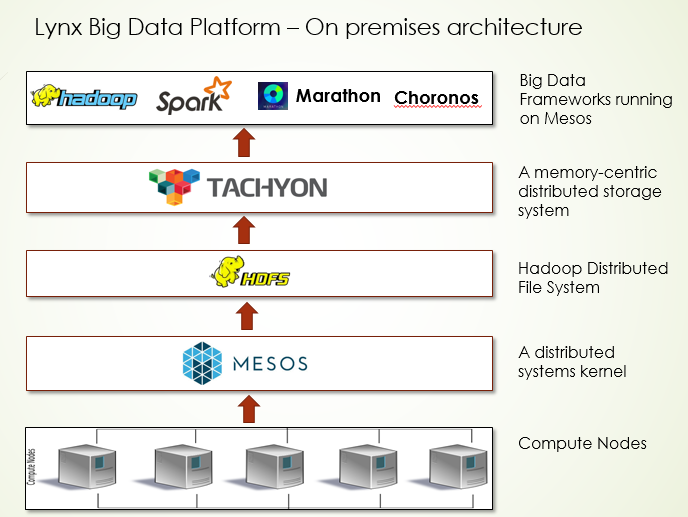


Figure 1 – LynxIT Big Data Platform – on premises architecture

### Mesos

Mesos is a distributed systems kernel and built using the same principles as the Linux kernel. Mesos abstracts CPU, memory, storage and other resources away from physical machines and enabling fault tolerant and elastic distributed systems.

Mesos allows to run multiple frameworks (e.g. Hadoop and Spark) on a cluster.

Also it is a possible to deploy Mesosphere’s Data Centre Operating system (DCOS) on AWS. Mesosphere’s DCOS is based on Mesos and managed by Mesosphere. Mesosphere’s DCOS and AWS are naturally well suited.

### TACHYON

Tachyon is a memory-centric distributed and reliable storage system that enables data sharing across clusters at memory speed.

One of the big advantages of using Tachyon is that if a job crashes, the data will still be accessible from Tachyon’s clusters.

Currently, Chinese internet giant Baidu utilises a production Tachyon cluster with 100 nodes and the data access speed has been increased by 30 times.

## Building predictive analytics in Spark ML

#### Predictive analytics in Spark

Spark is a great framework for predictive analytics. We’ve analysed the Flight Data and predicted airline delay. This model is based on supervised learning model.

In order to build predictive modelling, we’ve utilised 2007 flight data for build the model and 2008 flight data for test the model.

In this prototype, we’ve evaluated the following statistical models:

* Logistic Regression
* Decision trees
* Random Forest
* Naïve Bayes

On order to evaluate and choose the best model, we had to compute the below key metrics.

The below key metrics will tell you the accuracy of your model.

**Accuracy**

Example: 6 in 1000 flights is delayed flights (positive). A model which simply returns “negative”. E.g. not delayed. Therefore accuracy is 0.994 (99.4% correct). In general, accuracy is not a good metric. The reason is the data might be highly unbalanced. For this reason, metrics like precision and recall are typically used because they take into account the type of error.

**Precision**

Let’s assume, a model predicts 8 out of 1000 flights are delayed. If only 5 of those are truly delay, the precision is 5/8=0.625

**Recall**

The above example, we knew 6 out of 1000 flight are truly delay. The model correctly predicted 5 of them. It also predicted 3 incorrectly, the recall therefore 5/6=0.833

**F1 Score**

The F1 Score is the weighted average of precision model. The formula for F1 Score is 2\*(Recall \* Precision) / Recall + Precision)

##### Pre-processing and parsing data

import org.apache.spark.rdd.\_

import scala.collection.JavaConverters.\_

import au.com.bytecode.opencsv.CSVReader

import java.io.\_

import org.joda.time.\_

import org.joda.time.format.\_

import org.joda.time.format.DateTimeFormat

import org.joda.time.DateTime

import org.joda.time.Days

case class DelayRec(year: String,

month: String,

dayOfMonth: String,

dayOfWeek: String,

crsDepTime: String,

depDelay: String,

origin: String,

distance: String,

cancelled: String) {

val holidays = List("01/01/2007", "01/15/2007", "02/19/2007", "05/28/2007", "06/07/2007", "07/04/2007",

"09/03/2007", "10/08/2007" ,"11/11/2007", "11/22/2007", "12/25/2007",

"01/01/2008", "01/21/2008", "02/18/2008", "05/22/2008", "05/26/2008", "07/04/2008",

"09/01/2008", "10/13/2008" ,"11/11/2008", "11/27/2008", "12/25/2008")

def gen\_features: (String, Array[Double]) = {

val values = Array(

depDelay.toDouble,

month.toDouble,

dayOfMonth.toDouble,

dayOfWeek.toDouble,

get\_hour(crsDepTime).toDouble,

distance.toDouble,

days\_from\_nearest\_holiday(year.toInt, month.toInt, dayOfMonth.toInt)

)

new Tuple2(to\_date(year.toInt, month.toInt, dayOfMonth.toInt), values)

}

def get\_hour(depTime: String) : String = "%04d".format(depTime.toInt).take(2)

def to\_date(year: Int, month: Int, day: Int) = "%04d%02d%02d".format(year, month, day)

def days\_from\_nearest\_holiday(year:Int, month:Int, day:Int): Int = {

val sampleDate = new DateTime(year, month, day, 0, 0)

holidays.foldLeft(3000) { (r, c) =>

val holiday = DateTimeFormat.forPattern("MM/dd/yyyy").parseDateTime(c)

val distance = Math.abs(Days.daysBetween(holiday, sampleDate).getDays)

math.min(r, distance)

}

}

}

// function to do a preprocessing step for a given file

def prepFlightDelays(infile: String): RDD[DelayRec] = {

val data = sc.textFile(infile)

data.map { line =>

val reader = new CSVReader(new StringReader(line))

reader.readAll().asScala.toList.map(rec => DelayRec(rec(0),rec(1),rec(2),rec(3),rec(5),rec(15),rec(16),rec(18),rec(21)))

}.map(list => list(0))

.filter(rec => rec.year != "Year")

.filter(rec => rec.cancelled == "0")

.filter(rec => rec.origin == "ORD")

}

val data\_2007 = prepFlightDelays("s3://lynxit-emr-bucket/input/2007.csv").map(rec => rec.gen\_features.\_2)

val data\_2008 = prepFlightDelays("s3://lynxit-emr-bucket/input/2008.csv").map(rec => rec.gen\_features.\_2)

data\_2007.take(5).map(x => x mkString ",").foreach(println)

##### Parsing feature matrices into RDD

To use Spark ML-Lib's machine learning algorithms, first we parse our feature matrices into RDDs of LabeledPoint objects (for both the training and test datasets). LabeledPoint is ML-Lib's abstraction for a feature vector accompanied by a label. We consider flight delays of 15 minutes or more as "delays" and mark it with a label of 1.0, and under 15 minutes as "non-delay" and mark it with a label of 0.0.

import org.apache.spark.mllib.regression.LabeledPoint

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.mllib.feature.StandardScaler

def parseData(vals: Array[Double]): LabeledPoint = {

LabeledPoint(if (vals(0)>=15) 1.0 else 0.0, Vectors.dense(vals.drop(1)))

}

// Prepare training set

val parsedTrainData = data\_2007.map(parseData)

parsedTrainData.cache

val scaler = new StandardScaler(withMean = true, withStd = true).fit(parsedTrainData.map(x => x.features))

val scaledTrainData = parsedTrainData.map(x => LabeledPoint(x.label, scaler.transform(Vectors.dense(x.features.toArray))))

scaledTrainData.cache

// Prepare test/validation set

val parsedTestData = data\_2008.map(parseData)

parsedTestData.cache

val scaledTestData = parsedTestData.map(x => LabeledPoint(x.label, scaler.transform(Vectors.dense(x.features.toArray))))

scaledTestData.cache

scaledTrainData.take(3).map(x => (x.label, x.features)).foreach(println)

##### Function to computer metrics

// Function to compute evaluation metrics

def eval\_metrics(labelsAndPreds: RDD[(Double, Double)]) : Tuple2[Array[Double], Array[Double]] = {

val tp = labelsAndPreds.filter(r => r.\_1==1 && r.\_2==1).count.toDouble

val tn = labelsAndPreds.filter(r => r.\_1==0 && r.\_2==0).count.toDouble

val fp = labelsAndPreds.filter(r => r.\_1==1 && r.\_2==0).count.toDouble

val fn = labelsAndPreds.filter(r => r.\_1==0 && r.\_2==1).count.toDouble

val precision = tp / (tp+fp)

val recall = tp / (tp+fn)

val F\_measure = 2\*precision\*recall / (precision+recall)

val accuracy = (tp+tn) / (tp+tn+fp+fn)

new Tuple2(Array(tp, tn, fp, fn), Array(precision, recall, F\_measure, accuracy))

##### Logistic Regression model

import org.apache.spark.mllib.classification.LogisticRegressionWithSGD

// Build the Logistic Regression model

val model\_lr = LogisticRegressionWithSGD.train(scaledTrainData, numIterations=100)

// Predict

val labelsAndPreds\_lr = scaledTestData.map { point =>

val pred = model\_lr.predict(point.features)

(pred, point.label)

}

val m\_lr = eval\_metrics(labelsAndPreds\_lr).\_2

println("precision = %.2f, recall = %.2f, F1 = %.2f, accuracy = %.2f".format(m\_lr(0), m\_lr(1), m\_lr(2), m\_lr(3)))

**Result:**

Precision = 0.37, recall = 0.64, F1 = 0.47, accuracy = 0.59

##### Decision Tree model

import org.apache.spark.mllib.tree.DecisionTree

// Build the Decision Tree model

val numClasses = 2

val categoricalFeaturesInfo = Map[Int, Int]()

val impurity = "gini"

val maxDepth = 10

val maxBins = 100

val model\_dt = DecisionTree.trainClassifier(parsedTrainData, numClasses, categoricalFeaturesInfo, impurity, maxDepth, maxBins)

// Predict

val labelsAndPreds\_dt = parsedTestData.map { point =>

val pred = model\_dt.predict(point.features)

(pred, point.label)

}

val m\_dt = eval\_metrics(labelsAndPreds\_dt).\_2

println("precision = %.2f, recall = %.2f, F1 = %.2f, accuracy = %.2f".format(m\_dt(0), m\_dt(1), m\_dt(2), m\_dt(3)))

**Result:**

Precision = 0.41, recall = 0.25, F1 = 0.31, accuracy = 0.68

##### Random Forest model

import org.apache.spark.mllib.tree.RandomForest

import org.apache.spark.mllib.tree.configuration.Strategy

import org.apache.spark.mllib.util.MLUtils

val treeStrategy = Strategy.defaultStrategy("Classification")

val numTrees = 3

val featureSubsetStrategy = "auto"

val model\_rf = RandomForest.trainClassifier(parsedTrainData, treeStrategy, numTrees, featureSubsetStrategy, seed = 12345)

val labelsAndPreds\_rf = parsedTestData.map { point =>

val pred = model\_rf.predict(point.features)

(pred, point.label)

}

val m\_rf = eval\_metrics(labelsAndPreds\_rf).\_2

println("precision = %.2f, recall = %.2f, F1 = %.2f, accuracy = %.2f".format(m\_rf(0), m\_rf(1), m\_rf(2), m\_rf(3)))

**Result:**

Precision = 0.43, recall = 0.16, F1 = 0.23, accuracy = 0.70

##### NaiveBayes

import org.apache.spark.mllib.classification.{NaiveBayes, NaiveBayesModel}

import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics

val model\_nb = NaiveBayes.train(parsedTrainData, lambda = 1.0, modelType = "multinomial")

val labelsAndPreds\_nb = parsedTestData.map { point =>

val pred = model\_nb.predict(point.features)

(pred, point.label)

}

Precision = 0.45, recall = 0.19, F1 = 0.26, accuracy = 0.70

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | precision | recall | F1 | accuracy |
| Logistic Regression | 0.37 | 0.64 | 0.47 | 0.59 |
| Decision Tree | 0.41 | 0.25 | 0.31 | 0.68 |
| Random Forest | 0.43 | 0.16 | 0.23 | 0.70 |
| NaiveBayes | 0.45 | 0.19 | 0.26 | 0.70 |

##### Confusion Matrix based on the best model (NaiveBayes)

A confusion matrix is a specific table layout that allows visualization of the performance of an algorithm.

import org.apache.spark.mllib.evaluation.MulticlassMetrics

val metrics = new MulticlassMetrics(labelsAndPreds\_nb)

val metrics2 = new BinaryClassificationMetrics(labelsAndPreds\_nb)

println(metrics.confusionMatrix)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Prediction** | | |
| **Actual** |  | On-time | Delay |
| a=0 (on-time) | 218181.0 | 21713.0 |
| b=1 (delay) | 77547.0 | 17889.0 |

**High Cost, lower is better**

###### Summary

* Spark is an ideal solution to apply distributed computing methodologies to predictive analytics
* Spark has rich machine learning libraries
* The same model could be used for the prediction of public transport delay (e.g. train delay)

## Building Real time analytics framework on AWS

### Real world use cases

#### Fraud detection

Real-time analytics framework is an ideal solution to apply distributed processing methodologies to detect fraud or other nefarious activities in real time.

The below is high level fraud detection algorithm:

* To collect fraudulent transactions
* To build a statistical model that identifies the characteristics of a fraudulent transactions
* Apply that model to new transactions, which gives you a probability as to how likely it is that transaction is fraudulent
* Flag any transaction whose score exceed a statistically- determined threshold

#### Recommendation engine

By using real-time analytics framework, it is a possible to build highly scalable recommendation engine the same as Amazon’s product and Netflix’s movie. E.g. product recommendation

##### **Movie Recommendation prototype in Spark ML**

This movie recommendation prototype is based on Databrick’s training examples and modified to run on AWS EMR services.

This prototype uses collaborative filtering which is commonly used for recommender systems and to fill in the missing entries of a user-item association matrix, in our case, the user-movie rating matrix.

This prototype utilises the alternating least squares (ALS) algorithm to learn these latent factors. As the Netflix Prize competition has demonstrated, matrix factorization models are superior to classic nearest-neighbour techniques for producing product recommendations.

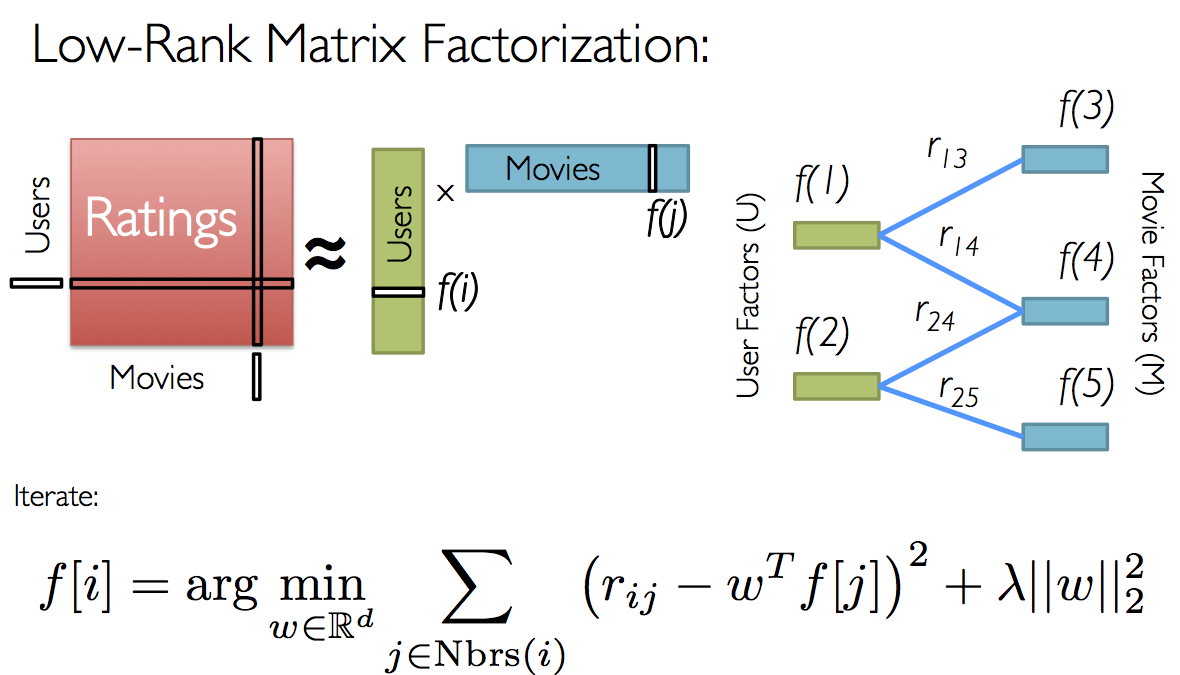


Figure 8 – Matrix Factorisation

The below code is written in Scala.

###### **Import libraries**

import java.io.File

import scala.io.Source

import org.apache.log4j.Logger

import org.apache.log4j.Level

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

import org.apache.spark.rdd.\_

import org.apache.spark.mllib.recommendation.{ALS, Rating, MatrixFactorizationModel}

###### **Load Data set to Spark Cluster**

val movieLensHomeDir = "s3://lynxit-emr-bucket/input/movielens/"

val movies = sc.textFile(movieLensHomeDir + "movies.dat").map { line =>

val fields = line.split("::")

// format: (movieId, movieName)

(fields(0).toInt, fields(1))

}.collect.toMap

val ratings = sc.textFile(movieLensHomeDir + "ratings.dat").map { line =>

val fields = line.split("::")

// format: (timestamp % 10, Rating(userId, movieId, rating))

(fields(3).toLong % 10, Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble))

}

###### **Verify and count the number of ratings**

val numRatings = ratings.count

val numUsers = ratings.map(\_.\_2.user).distinct.count

val numMovies = ratings.map(\_.\_2.product).distinct.count

println("Got " + numRatings + " ratings from " + numUsers + " users on " + numMovies + " movies.")

###### **Split dataset –Training (60%), validation (20%) and testing (20%)**

val training = ratings.filter(x => x.\_1 < 6)

.values

.cache()

val validation = ratings.filter(x => x.\_1 >= 6 && x.\_1 < 8)

.values

.cache()

val test = ratings.filter(x => x.\_1 >= 8).values.cache()

val numTraining = training.count()

val numValidation = validation.count()

val numTest = test.count()

println("Training: " + numTraining + ", validation: " + numValidation + ", test: " + numTest)

/\*\* Compute RMSE (Root Mean Squared Error). \*/

def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating], n: Long): Double = {

val predictions: RDD[Rating] = model.predict(data.map(x => (x.user, x.product)))

val predictionsAndRatings = predictions.map(x => ((x.user, x.product), x.rating))

.join(data.map(x => ((x.user, x.product), x.rating))).values

math.sqrt(predictionsAndRatings.map(x => (x.\_1 - x.\_2) \* (x.\_1 - x.\_2)).reduce(\_ + \_) / n)

}

###### **Training data set using ALS algoritm**

val ranks = List(8, 12)

val lambdas = List(0.1, 10.0)

val numIters = List(10, 20)

var bestModel: Option[MatrixFactorizationModel] = None

var bestValidationRmse = Double.MaxValue

var bestRank = 0

var bestLambda = -1.0

var bestNumIter = -1

for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {

val model = ALS.train(training, rank, numIter, lambda)

val validationRmse = computeRmse(model, validation, numValidation)

println("RMSE (validation) = " + validationRmse + " for the model trained with rank = "

+ rank + ", lambda = " + lambda + ", and numIter = " + numIter + ".")

if (validationRmse < bestValidationRmse) {

bestModel = Some(model)

bestValidationRmse = validationRmse

bestRank = rank

bestLambda = lambda

bestNumIter = numIter

}

}

###### **Evaluate the best model on the test set**

val testRmse = computeRmse(bestModel.get, test, numTest)

println("The best model was trained with rank = " + bestRank + " and lambda = " + bestLambda

+ ", and numIter = " + bestNumIter + ", and its RMSE on the test set is " + testRmse + ".")

###### **Create a naive baseline and compare it with the best model**

val meanRating = training.union(validation).map(\_.rating).mean

val baselineRmse =

math.sqrt(test.map(x => (meanRating - x.rating) \* (meanRating - x.rating)).mean)

val improvement = (baselineRmse - testRmse) / baselineRmse \* 100

println("The best model improves the baseline by " + "%1.2f".format(improvement) + "%.")

###### **Use the model to make personal recommendations**

val candidates = sc.parallelize(movies.keys.toSeq)

val recommendations = bestModel.get

.predict(candidates.map((1, \_)))

.collect()

.sortBy(- \_.rating)

.take(10)

var i = 1

println("Movies recommended for you:")

recommendations.foreach { r =>

println("%2d".format(i) + ": " + movies(r.product))

i += 1

}

### LynxIT Real-time analytics architecture

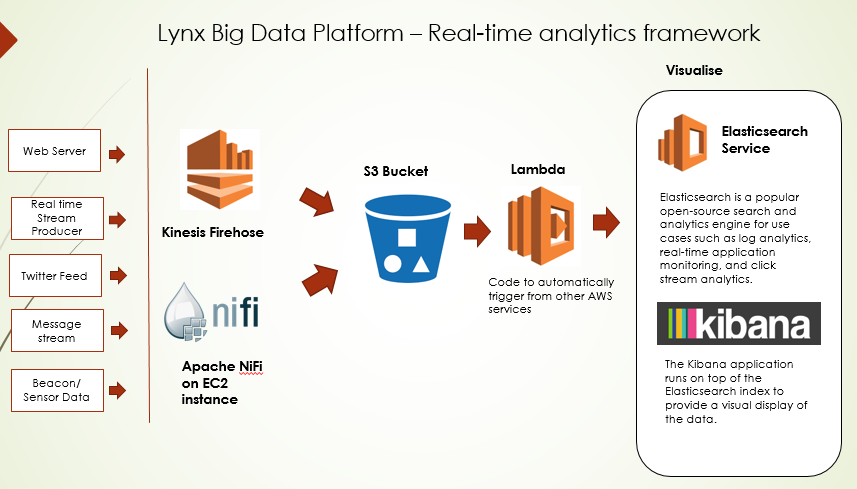


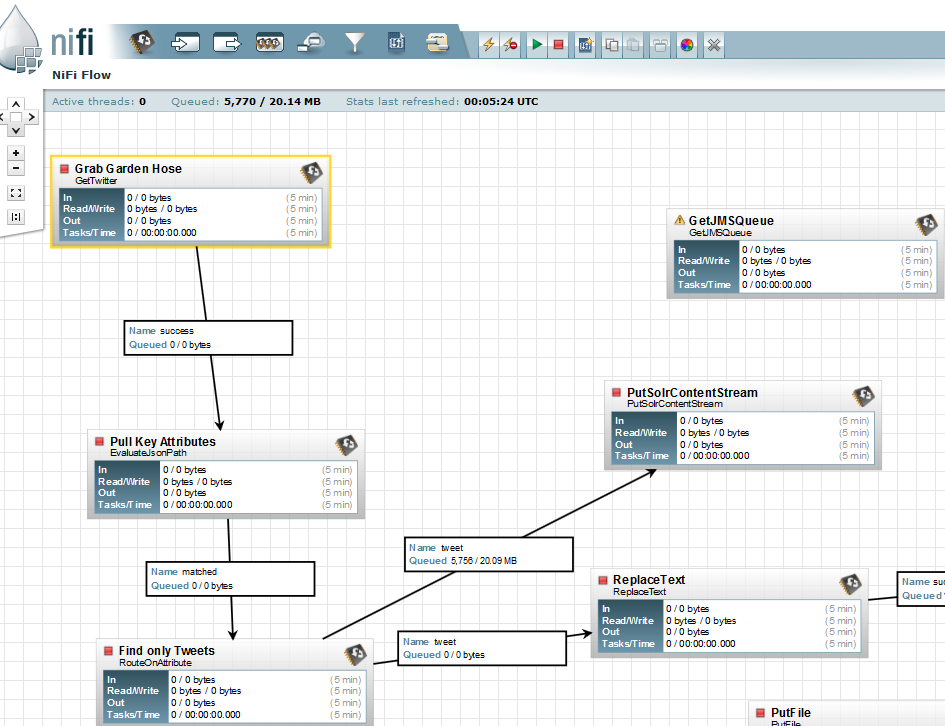
Figure 9 – Real Time analytics architecture

### Capturing real-time streaming by using Apache NiFi

Apache NiFi is automate the data flow between systems and purely based on GUI interface.

Apache NiFi Key features:

* Collect any and IoT data including sensors, clickstreams, files and social feeds
* Able to parse, filter and transform data
* Enables rapid development and effective testing



## Experience and Work with extremely large data sets at massive scale

In general, collection of extremely high volume data, like Google and Yahoo, is very costly process.

However, if you would like to work with extremely large data sets at a massive scale, you can use public data sets available on AWS. A corpus of web crawl data composed of over 5 billion web pages. This data set is freely available on Amazon S3 and is released under the Common Crawl Terms of Use.

Available at **s3://aws-publicdatasets/common-crawl/**

**Size**: 541 TB  
**Source**: Common Crawl Foundation - http://commoncrawl.org  
**Created On**: February 15, 2012  
**Last Updated**: November 25, 2015

In this prototype, I’ll show you how to process the Web ARChive data (WARC format) using Spark on AWS.

The WARC format is raw data from the crawl, providing a direct mapping to the crawl process. Not only does the format store the HTTP response from the websites it contacts (WARC-Type: response), it also stores information about how that information was requested (WARC-Type: request) and metadata on the crawl process itself (WARC-Type: metadata).

WARC file example

WARC/1.0

WARC-Type: response

WARC-Date: 2014-08-02T09:52:13Z

WARC-Record-ID:

Content-Length: 43428

Content-Type: application/http; msgtype=response

WARC-Warcinfo-ID:

WARC-Concurrent-To:

WARC-IP-Address: 212.58.244.61

WARC-Target-URI: http://news.bbc.co.uk/2/hi/africa/3414345.stm

WARC-Payload-Digest: sha1:M63W6MNGFDWXDSLTHF7GWUPCJUH4JK3J

WARC-Block-Digest: sha1:YHKQUSBOS4CLYFEKQDVGJ457OAPD6IJO

WARC-Truncated: length

HTTP/1.1 200 OK

Server: Apache

Vary: X-CDN

Cache-Control: max-age=0

Content-Type: text/html

Date: Sat, 02 Aug 2014 09:52:13 GMT

Expires: Sat, 02 Aug 2014 09:52:13 GMT

Connection: close

Set-Cookie: BBC-UID=...; expires=Sun, 02-Aug-15 09:52:13 GMT; path=/; domain=bbc.co.uk;

<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/REC-html40/loose.dtd">

<html>

<head>

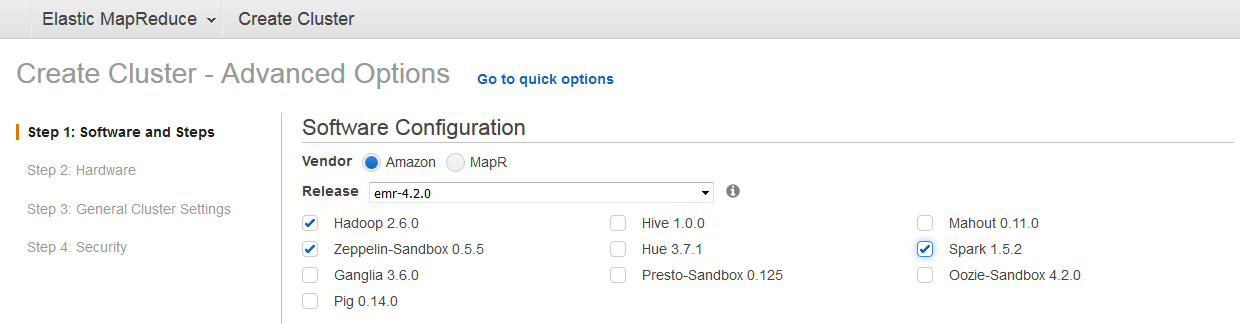
<title>

BBC NEWS | Africa | Namibia braces for Nujoma exit

### Launch EMR cluster with Spark

1. Click on Create cluster from EMR console
2. Click on Go to advanced options link
3. In the software configuration, enable Zeppelin-Sandbox 0.5.5 and Spark 1.5.2 as follows

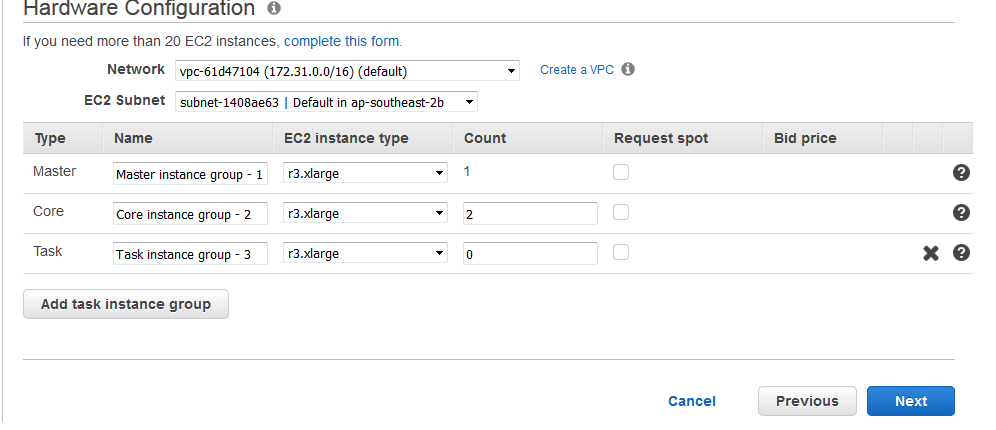
[{"classification":"spark-defaults", "properties":{"spark.serializer":"org.apache.spark.serializer.KryoSerializer", "spark.dynamicAllocation.enabled":"true"}, "configurations":[]}]



1. Enter configuration section, enable Spark to use dynamic allocation of executors. You can get the better performance.



1. In the hardware configuration, choose the memory optimised R3 instance type.



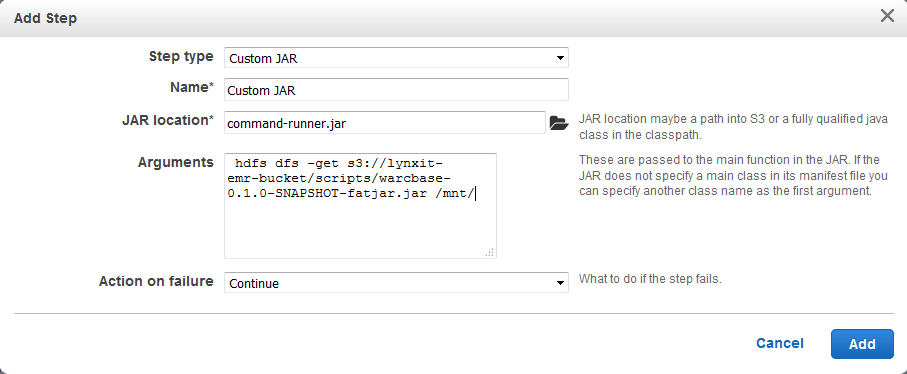
1. Enter a cluster name. e.g. MySparkCluster

### Analysing Web Archive (WARC) File

#### Copy dependency JAR files to master node

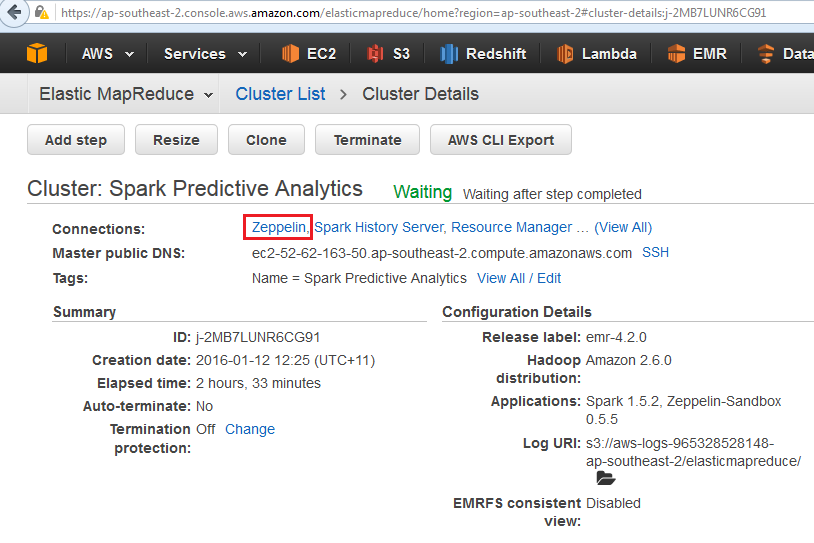
The below step will copy a dependency JAR file from an S3 location to the master node of your cluster.

hdfs dfs -get s3://lynxit-emr-bucket/scripts/warcbase-0.1.0-SNAPSHOT-fatjar.jar /mnt/



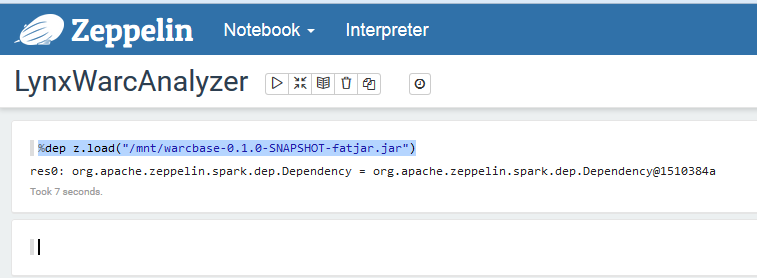
#### Opening the Zeppelin Notebook

Click on Zeppelin link.

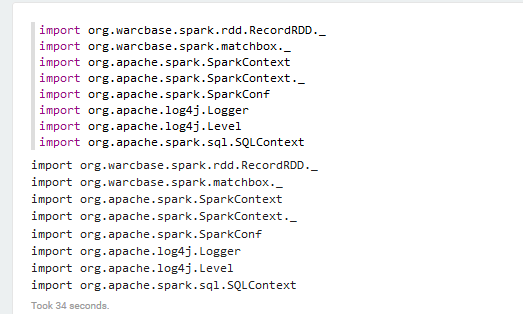


##### Load dependency

%dep z.load("/mnt/warcbase-0.1.0-SNAPSHOT-fatjar.jar")



##### Import libraries



#### Analyse of Site Link Structure

The below Spark code analyses the websites link structure grouped by crawl date.

val r = RecordLoader.loadWarc("s3://aws-publicdatasets/common-crawl/crawl-data/CC-MAIN-2015-48/segments/1448398444047.40/warc/CC-MAIN-20151124205404-00000-ip-10-71-132-137.ec2.internal.warc.gz",sc)

.keepValidPages()

.map(r => (r.getCrawldate, ExtractLinks(r.getUrl, r.getContentString)))

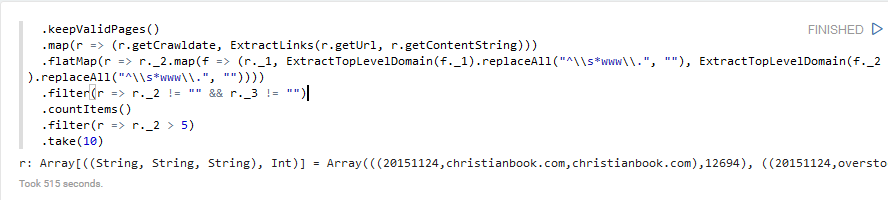
.flatMap(r => r.\_2.map(f => (r.\_1, ExtractTopLevelDomain(f.\_1).replaceAll("^\\s\*www\\.", ""), ExtractTopLevelDomain(f.\_2).replaceAll("^\\s\*www\\.", ""))))

.filter(r => r.\_2 != "" && r.\_3 != "")

.countItems()

.filter(r => r.\_2 > 5)

.take(10)



#### The output

((20151124,christianbook.com,christianbook.com),12694)

((20151124,overstock.com,overstock.com),9560)

((20151124,shop.nordstrom.com,shop.nordstrom.com),8860)

((20151124,medicinenet.com,medicinenet.com),8551)

I’ve used the following AWS instance type with 1 Master node and 2 compute nodes.

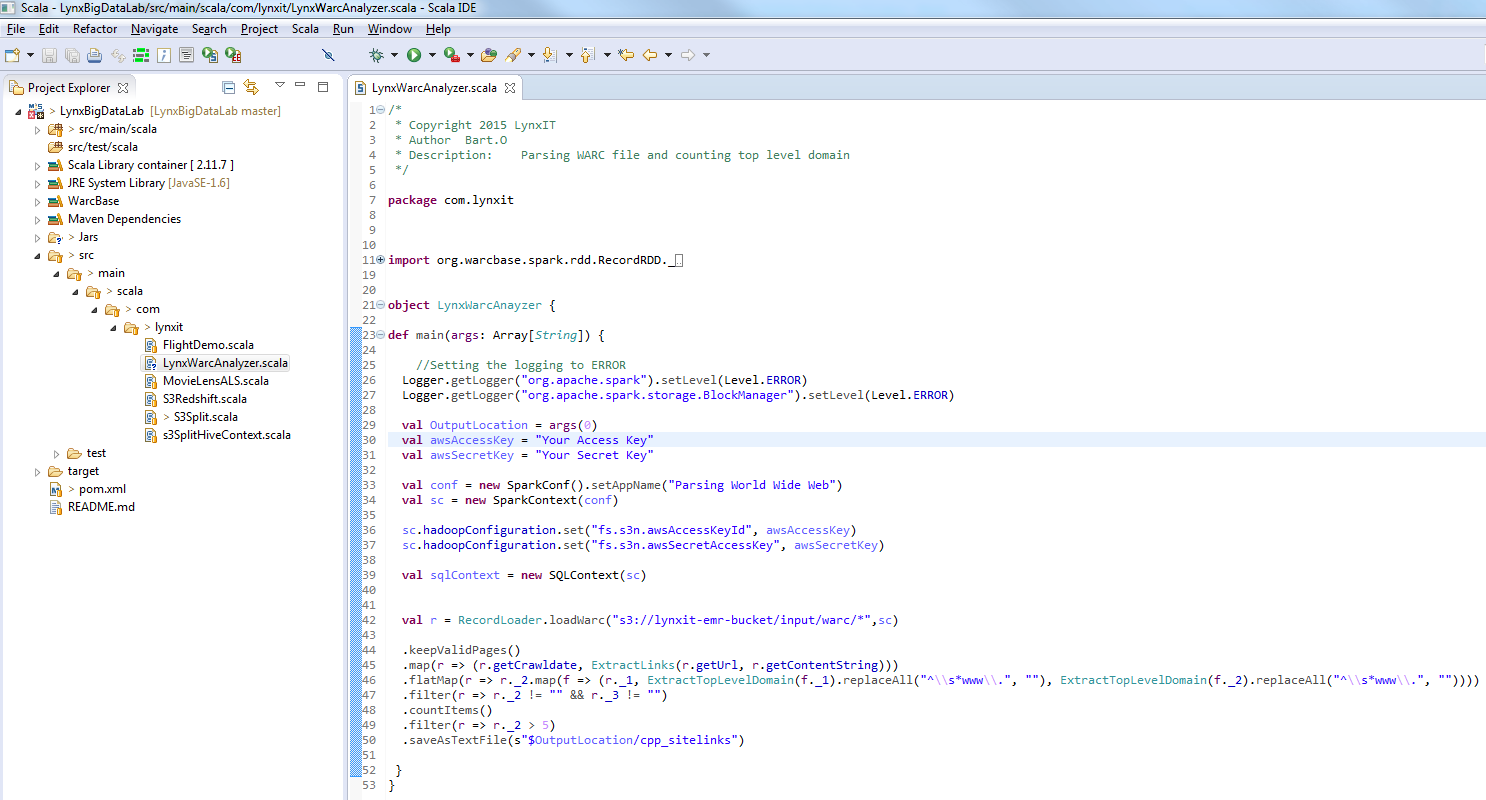
* Instance Type - r3.xlarge
* vCPU - 4
* Memory – 30Gib
* SSD Storage – 1 x 80

It took 7 minutes to analyse the single WARC file. Therefore, if you use 2 instances with the above configurations and process the 541TB data with all 5 billion pages, you will need 347 days.

If you want to process within 24 hours, then you will need approximately 400 compute nodes. It will cost you approximately USD2700, if you run your instances in N. Virginia region.

### Actual Implementation in Spark

If you want to run the whole solution at once, you will need to compile and create a java executable (JAR) file.



#### Submit Spark Jobs

##### Step1.

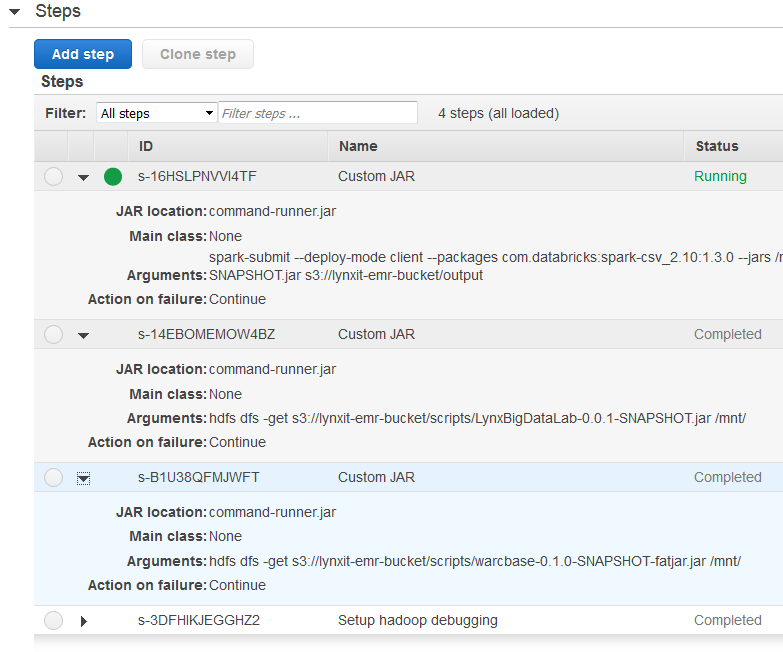
hdfs dfs -get s3://lynxit-emr-bucket/scripts/warcbase-0.1.0-SNAPSHOT-fatjar.jar /mnt/

##### Step2.

hdfs dfs -get s3://lynxit-emr-bucket/scripts/LynxBigDataLab-0.0.1-SNAPSHOT.jar /mnt/

##### Step3.

spark-submit --deploy-mode client --packages com.databricks:spark-csv\_2.10:1.3.0 --jars /mnt/warcbase-0.1.0-SNAPSHOT-fatjar.jar --class com.lynxit.LynxWarcAnayzer /mnt/LynxBigDataLab-0.0.1-SNAPSHOT.jar s3://lynxit-emr-bucket/output



# Building ETL Server using AWS Native Services

## Our Design Target

* **A complete Extract-Transform-Load solution** which
  + Is able to get data from **various data source** (Oracle, MS SQL, MySQL…)
  + **Extract** data and **transfer** files to **S3**
  + **Load** data into **Redshift** from **S3** with
  + Necessary, potentially complicate, **Transformation** and Merge processes
  + Using **minimum AWS service** to **reduce running cost** for customers with their
  + Familiar skill-set (Win, Linux, etc)

## Three Editions

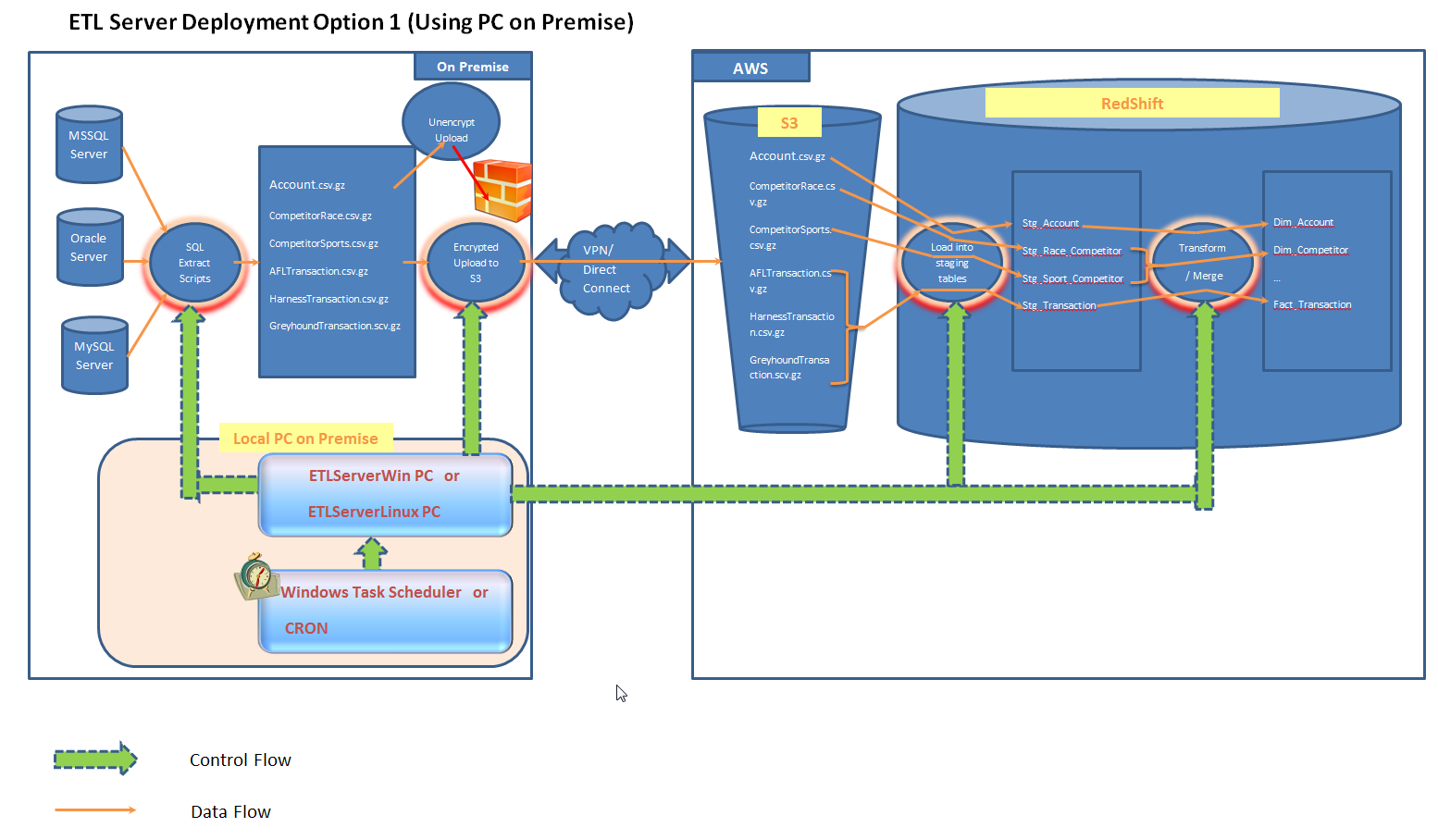
We have 3 editions for customer to choose from, to suit their tech skill-set

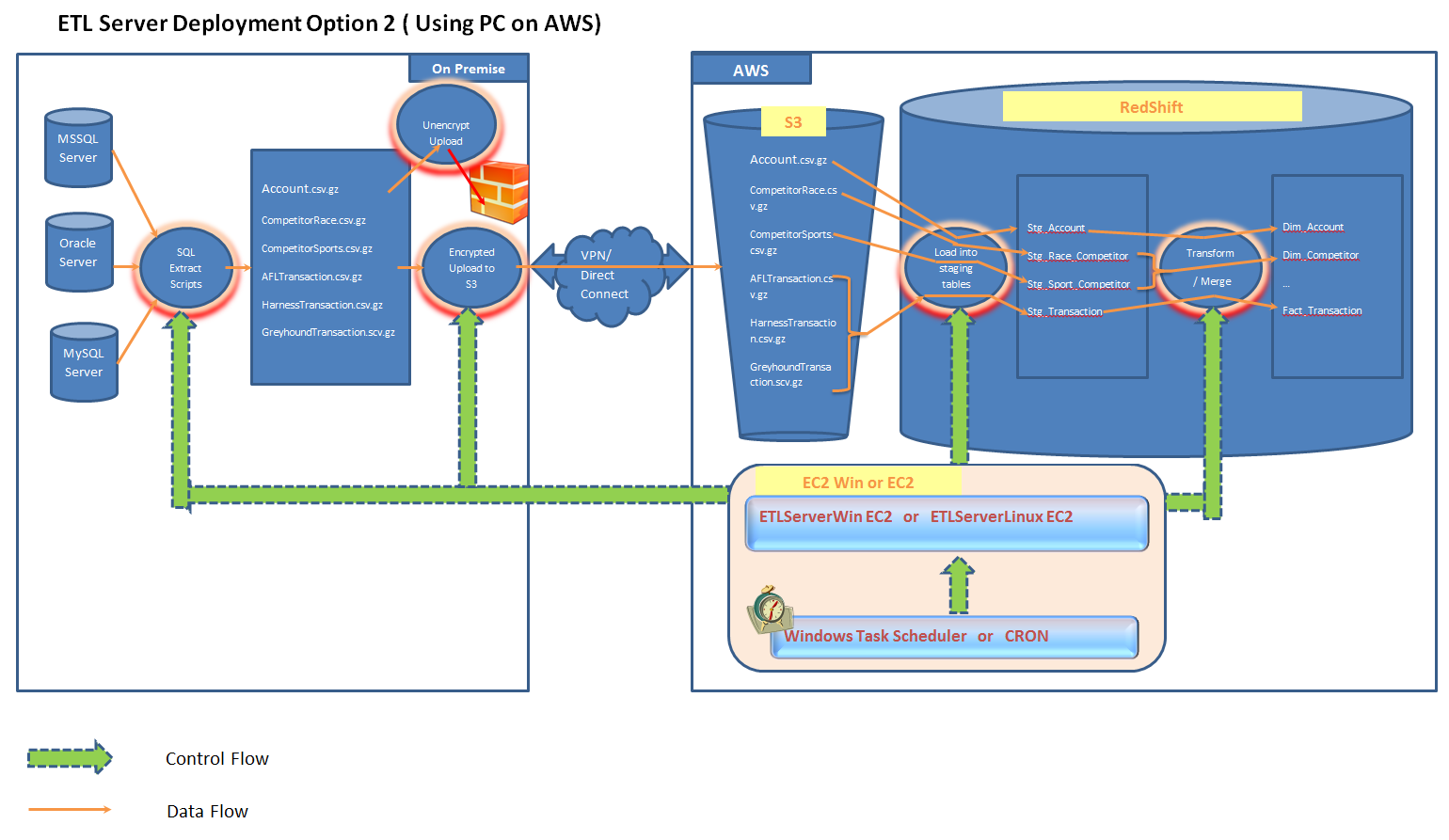
* ETLServerWin
* ETLServerLinux
* ETLServerDynamic

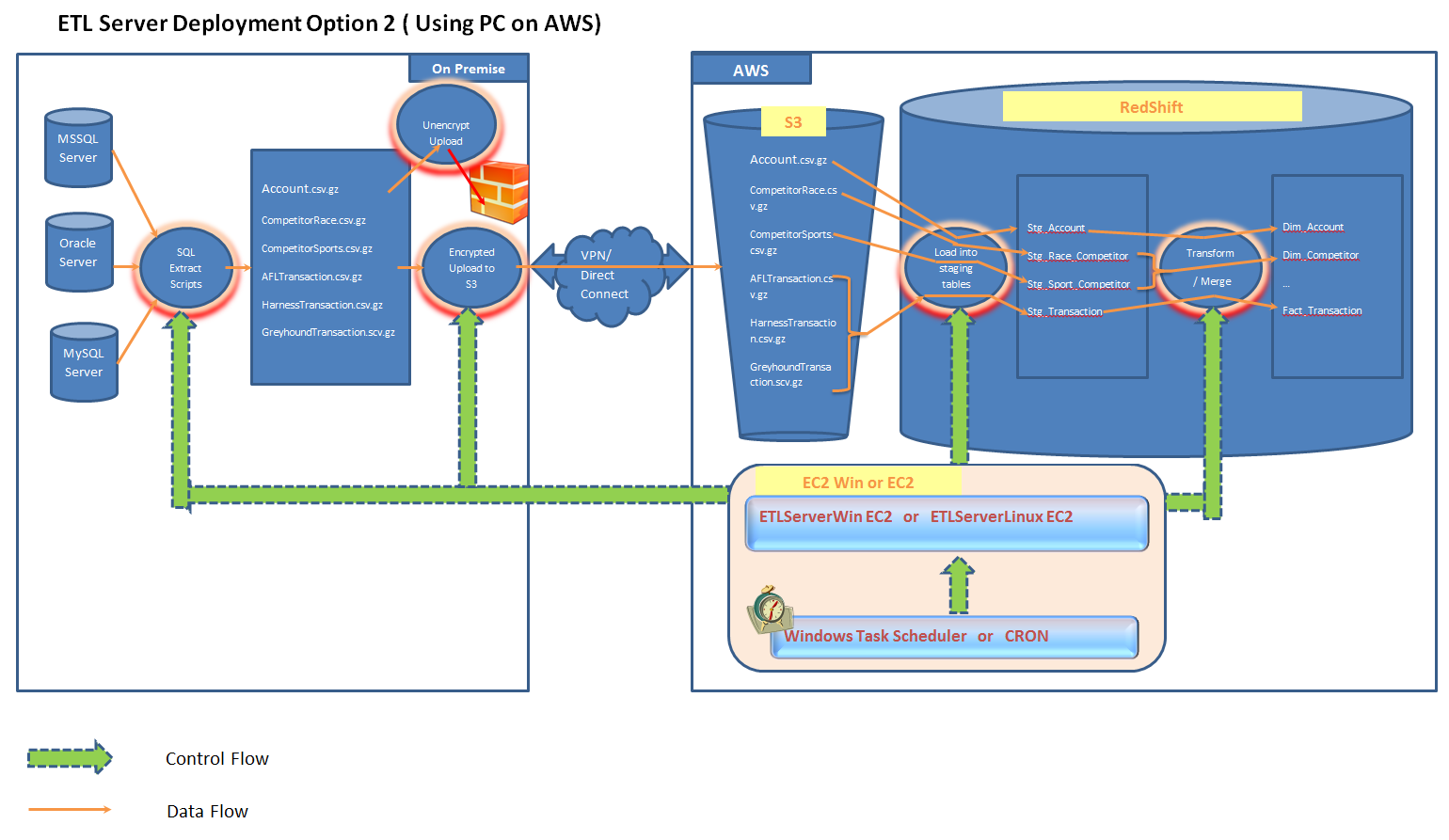
## Three Deployment Options:

We have 3 Deployment Options for customer to choose from, to fit into their existing infrastructure and to meet their ETL requirement

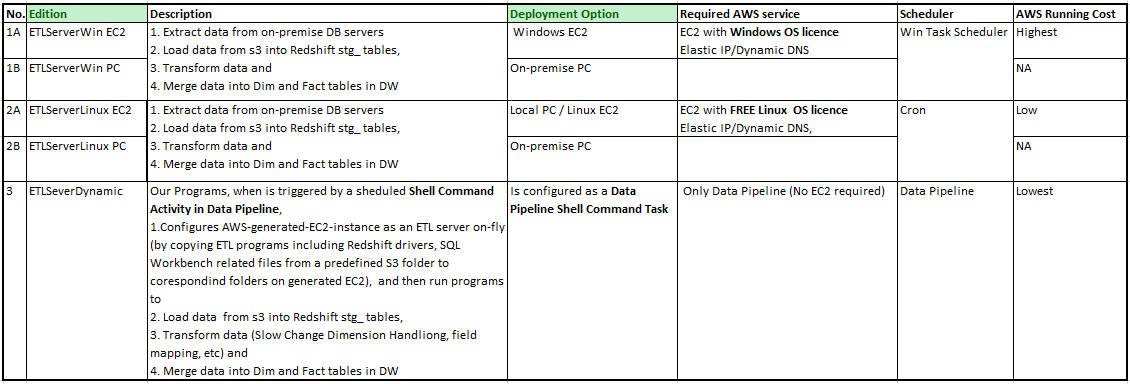
* On-Premise PC (ETLServerWin or ETLServerLinux)
* AWS EC2 (ETLServerWin or ETLServerLinux)
* AWS Data Pipeline with a single-shell-command activity (no EC2 cost, no Win license cost. LOWEST COST solution)







## How to choose from 3 Editions and 3 Deployment Options



## Main Functions we have realised in our prototype:

* Extract Data From MS SQL server
* Stop all un-encrypted files from uploading into AWS! (AWS S3 Server Side encryption)
* All files must be zipped before uploading
* All files in S3 are kept encrypted (By S3 configuration)
* Multiple files in one S3 folder can be merged into ONE staging table
* All records which already exist in DW tables are filtered out in merging process
* Multiple staging tables can be joined and merged into ONE DW table with data source flag inserted
* Multiple data sources can be connected in ONE ETL process from ETL Server (One SQL Workbench Command line can call multiple SQL Script Files, One SQL Script File can have multiple sections which may connect to different database sources/servers )
* Files in S3 are moved to Backup folder after they are imported into Redshift
* All files in S3 Backup folder are removed after x days (By Configuring AWS S3)

# Appendix A

### How to configure a production-ready Mesosphere Cluster on Ubuntu

The below is deployment instruction

* <https://www.digitalocean.com/community/tutorials/how-to-configure-a-production-ready-mesosphere-cluster-on-ubuntu-14-04>

### Glossary of Terms

**Avro.** Serialization and remote procedure capabilities for interacting with Hadoop, using the JSON data format. Offers a straightforward approach for portraying complex data structures within a Hadoop MapReduce job.

**Cluster.** Large-scale Hadoop environment commonly deployed on a collection of inexpensive, commodity servers. Clusters achieve high degrees of scalability merely by adding extra servers when needed, and frequently employ replication to increase resistance to failure.

**Hadoop.** A specific approach for implementing the MapReduce architecture, including a foundational platform and a related ecosystem.

**HBase.** A distributed – but non-relational – database that runs on top of the Hadoop File System.

**Hadoop File System (HDFS).** File system designed for portability, scalability, and large-scale distribution. Written in Java, HDFS employs replication to help increase reliability of its storage.

**Hive.** Data warehousing infrastructure constructed on top of Hadoop. Offers query, analysis, and data summarization capabilities.

**JavaScript Object Notation (JSON).** An open data format standard. Language independent, and human-readable, often used as a more efficient alternative to XML.

**Mahout.** A collection of algorithms for classification, collaborative filtering, and clustering that deliver machine learning capabilities. Commonly implemented on top of Hadoop.

**Maven.** A tool that standardizes and streamlines the process of building software, including managing dependencies among external libraries, components, and packages.

**NoSQL.** Refers to an array of independent technologies that are meant to go beyond standard SQL to provide new access methods, generally to work with unstructured or semi-structured data.

**Oozie.** A workflow engine that specializes in scheduling and managing Hadoop jobs.

**Pig.** Technology that simplifies the job of creating MapReduce applications running on Hadoop platforms. Uses a language known as ‘Pig Latin’.

**Spark.** General-purpose cluster computing system, intended to simplify the job of writing massively parallel processing jobs in higher-level languages such as Java, Scala, and Python. Also includes Shark, which is Apache Hive running on the Spark platform.

**Sqoop.** Tool meant to ease the job of moving data - in bulk – to and from Hadoop as well as structured information repositories such as relational databases.

**YARN.** New streamlined techniques for organizing and scheduling MapReduce jobs in a Hadoop environment.

# References

Robert D. Schneider, Everything you need to understand and get started with Big Data and Hadoop, 28-32

Databricks. (November 5, 2014). [Blog Post], Spark officially sets a new record in large-scale sorting. Retrieved from <https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html>

ACM digital library. **Matrix Factorization Techniques for Recommender Systems**. Retrieved from

<http://dl.acm.org/citation.cfm?id=1608614>

Databricks. Hands on Exercises. Retrieved from <https://databricks-training.s3.amazonaws.com/movie-recommendation-with-mllib.html#training-using-als>

# Big Data Project Team

The below is a proposed Big data project team. This can vary depending on an organization.

**Agile Project Team**

Project Manager

Solution Architect

Business Analyst

Data Scientist

Data Engineer

Application Developers

Test Lead

Testers

Visualisation

Business Sponsor

**Operations team**

Hadoop team lead

Network Engineer

Security Engineer

DBA’s

Hadoop Admin

Cluster Engineers

ETL Architects

ETL Data Cleansing Architect

ETL metadata Architects