On-Demand Risk Management

Value at Risk (VaR) is a risk assessment measure that is popularly used in risk management models. VaR is based on current market conditions, time and day of investment among other factors which are utilized to generate a confidence value to evaluate the risk involved for a given investment. The VaR can be calculated across days to generate comparison models for portfolios. Visualization of the results can also be integrated to enhance understandability and provide deeper analysis.

The reference application will be demonstrated to the financial clients and can be enhanced and customized to match the client requirement.

For Murali Adi Reddy, beneficiary the following work we had plans to perform:

Visualization
Results are currently persisted on the cluster. We are planning to enhance the solution by showing the results in Tableau dashboards. This would enable the users to view and analyze results across several days. This would require connecting Tableau software to the cluster and building dashboards to view the data.

Job Scheduler
Write an automatic job scheduler to schedule the and run the programs on daily basis. Currently, the jobs are run manually by executing the scripts as required. This should be automated using a job scheduler so that the jobs can be run automatically.

In-memory storage
The solution should be enhanced with in-memory storage solution. This will further enhance the computing times and provide the results faster. The in-memory storage can also be used to store recently used calculations and trials. We need to develop a caching strategy after analyzing and testing various caching strategies for the solution.

Source codes

The download-all script is run to download all the stocks and factors data from the mentioned url in the script. The stocks and factors are downloaded for the stock names mentioned as symbols in the symbols_all file.

```
#!/bin/bash

if [[ $(date -d "-1 days" +%u) -gt 5 ]] ; then
DATE=`date -d "-3 days" +%Y-%m-%d`
else
DATE=`date -d "-1 days" +%Y-%m-%d`
fi


if `hdfs dfs -test -d /projects/RefApps/OnDemandVaR/data/tmp/$DATE/`
then
echo "Directory exists for date: $DATE"
`hdfs dfs -rm -R /projects/RefApps/OnDemandVaR/data/tmp/$DATE/`
else
hdfs dfs -mkdir -p /projects/RefApps/OnDemandVaR/data/tmp/$DATE/stocks
hdfs dfs -mkdir -p /projects/RefApps/OnDemandVaR/data/tmp/$DATE/
factors
fi

if `hdfs dfs -test -d /projects/RefApps/OnDemandVaR/data/var/$DATE/`
then
echo "VAR Directory exists for date: $DATE"
`hdfs dfs -rm -R /projects/RefApps/OnDemandVaR/data/var/$DATE/`
fi

SYMBOLS=`hdfs dfs -cat $1`
while read -r SYMBOL; do
  data=`curl http://ichart.yahoo.com/table.csv?s=
$SYMBOL&a=0&b=1&c=2000&d=0&e=31&f=2013&g=d&ignore=.csv`
  if [[ "$data" != *"Yahoo"* ]]; then
  echo "$data" | hdfs dfs -appendToFile - /projects/RefApps/
OnDemandVaR/data/tmp/$DATE/stocks/$SYMBOL.csv
  fi
  sleep 1
done <<< "$SYMBOLS"

echo ' Completed downloading Stocks'


SNP=`curl http://ichart.yahoo.com/table.csv?
s=^GSPC&a=0&b=1&c=2000&d=0&e=31&f=2013&g=d&ignore=.csv`
if [[ "$SNP" != *"Yahoo"* ]]; then
echo "$SNP" | hdfs dfs -appendToFile - /projects/RefApps/OnDemandVaR/
data/tmp/$DATE/factors/SNP.csv
fi

NDX=`curl http://ichart.yahoo.com/table.csv?
s=^IXIC&a=0&b=1&c=2000&d=0&e=31&f=2013&g=d&ignore=.csv`
if [[ "$NDX" != *"Yahoo"* ]]; then
echo "$NDX" | hdfs dfs -appendToFile - /projects/RefApps/OnDemandVaR/
```

```
data/tmp/$DATE/factors/NDX.csv
fi

echo ' Completed downloading Factors'
```

The symbols_all text file contains a sample set of names of all the stocks whose data is to be downloaded by the download-all script. This data is used to calculate the VaR.

TFSC
TFSCR
TFSCU
TFSCW
PIH
FLWS
FCTY
FCCY
MLNX
MELR
MEMP
MRD
MENT
MTSL
OVBC
OHRP
ODFL
OLBK
ONB
OPOF
OSBC
OSBCP
ZEUS
OFLX
OMER
OABC
OMCL
OVTI
ONNN
OTIV
OGXI
ONCY
OMED
ONTX
ONTY
OHGI
ONFC
ONVI
OTEX
OPXA
OPHT
OPLK
OBAS
OCC
OPHC
OPB
ORMP
OSUR
ORBC
ORBT

ORBK
ORLY
OREX
SEED
ORIT
ORRF
OFIX
OSIS
OSIR
OSN
OTEL
OTIC
OTTR
OUTR
OVAS
OVRL
OSTK
OXBR
OXBRW
OXFD
OXLC
OXLCN
PULB
PCYO
QADA
QADB
QCCO
QCRH
QGEN
QIWI
QKLS
QLIK
QLGC
QLTI
QCOM
QLTY
QSII
QBAK
QLYS
QTWW
QRHC
QUIK
QDEL
QNST
QUMU
QUNR
QTNT
QTNTW
RRD
RADA

RDCM
ROIA
ROIAK
RSYS
RDUS
RDNT
RDWR
RMBS
RAND
RLOG
GOLD
RPTP
RAVN
ROLL
RICK
RCMT
RLOC
RDI
RDIB
RGSE
RNWK
RP
RCPT
DAX
QYLD
RCON
REPH
RRGB
RDHL
REDF
RGDO
REGN
RGLS
REIS
RELV
RLYP
MARK
REMY
RNST
REGI
RCII
RTK
RENT
RGEN
RPRX
RPRXW
RPRXZ
RJET
RBCAA
FRBK

REFR
RESN
REXI
RECN
RGDX
ROIC
SALE
RTRX
RVNC
RBIO
RVLT
RWLK
REXX
RFIL
RFMD
RGCO
RIBT
RIBTW
RELL
RIGL
NAME
RNET
RITT
RITTW
RIVR
RVBD
RVSB
RLJE
RMGN
ROBO
RCPI
FUEL
RMTI
RCKY
RMCF
RSTI
ROIQ
ROIQU
ROIQW
ROKA
ROSG
ROSE
ROST
ROVI
RBPAA
RGLD
ROYL
FUND
RPXC
RRST

RTIX
RBCN
RUSHA
RUSHB
RTGN
RUTH
Z
ZN
ZION
ZIONW
ZIONZ
ZIOP
ZIXI
ZGNX
ZSAN
ZSPH
ZU
ZUMZ
ZNGA

The CsvToParquetFile is run to convert the downloaded data by the download-all script from the CSV format to Parquet format as required for the next step.

```
package com.innovative

import java.text.SimpleDateFormat
import java.util.Calendar
import scala.reflect.runtime.universe
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext

object CsvToParquetFile {
  case class Stocks(fileName: String, content: String)
  case class Factors(fileName: String, content: String)
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf()
    val sc = new SparkContext(conf);
    val sqlContext = new SQLContext(sc);

    import sqlContext.implicits._

    val format = new SimpleDateFormat("y-MM-dd")
    val cal = Calendar.getInstance()
    cal.add(Calendar.DATE, -1)
    val formatter = new SimpleDateFormat("EEE")
    val day = formatter.format(cal.getTime())

    /*
     * Skip weekends
     */
    def getDayOfWeek(days: String): String = {
      if (days.equalsIgnoreCase("Sun")) {
        cal.add(Calendar.DATE, -2)
        return format.format(cal.getTime())
      } else {
        return format.format(cal.getTime())
      }
    }

    val date = getDayOfWeek(day)
    val stocksData = sc.wholeTextFiles(args(0) + "/" + date + "/
stocks")
    val factorsData = sc.wholeTextFiles(args(0) + "/" + date + "/
factors")

    /*
     * saving stocks data in parquet format in hdfs
     * */
    val dfStocks = stocksData.map {
      case (name, content) =>
        {
          val firstIndex = name.lastIndexOf("/") + 1;
```

```
          val lastIndex = name.indexOf(".", firstIndex);
          val fileName = name.substring(firstIndex, lastIndex);
          new Stocks(fileName, content)
        }
    }.toDF
    dfStocks.write.format("parquet").save(args(1) + "/" + date + "/
stocks/")

    /*
     * saving factors data in parquet format in hdfs
     * */
    val dfFactors = factorsData.map {
      case (name, content) =>
        {
          val first = name.lastIndexOf("/") + 1;
          val last = name.indexOf(".", first);
          val fileName = name.substring(first, last);
          new Factors(fileName, content)
        }
    }.toDF
    dfFactors.write.format("parquet").save(args(1) + "/" + date + "/
factors/")

  }
}
```

The OptimizedCsvToParquetFile is an optimized version that is run to convert the downloaded data by the download-all script from the CSV format to Parquet format as required for the next step.

```
package com.innovative

import java.text.SimpleDateFormat
import java.util.Calendar
import scala.reflect.runtime.universe
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.FileSystem
import org.apache.hadoop.fs.Path
import org.apache.spark.rdd.RDD
import org.apache.spark.storage.StorageLevel

object OptimizedCsvToParquetFile {

  case class VarData(Date: String, Open: Double, High: Double, Low:
Double, Close: Double, Volume: Long, Adj_Close: Double, ticker:
String);

  def main(args: Array[String]): Unit = {
    val conf = new SparkConf()
    val sc = new SparkContext(conf);
    val sqlContext = new SQLContext(sc);
    val configuration = new Configuration();
    val fs = FileSystem.get(configuration)

    import sqlContext.implicits._

    val format = new SimpleDateFormat("y-MM-dd")
    val cal = Calendar.getInstance()
    cal.add(Calendar.DATE, -1)
    val formatter = new SimpleDateFormat("EEE")
    val day = formatter.format(cal.getTime())

    /*
     * Skip weekends
     */

    def getDayOfWeek(days: String): String = {
      if (days.equalsIgnoreCase("Sun")) {
        cal.add(Calendar.DATE, -2)
        return format.format(cal.getTime())
      } else {
        return format.format(cal.getTime())
      }
    }

    val date = getDayOfWeek(day)
```

```
    val stocksList = fs.listStatus(new Path(args(0) + "/" + date + "/
stocks")).map(fileName => {

      val fileRDD =
sc.textFile(fileName.getPath.toString()).mapPartitionsWithIndex
{ (idx, iter) => if (idx == 0) iter.drop(1) else iter }

      val nameOfFile = fileName.getPath.toString()
      val firstIndex = nameOfFile.lastIndexOf("/") + 1
      val lastIndex = nameOfFile.indexOf(".", firstIndex)
      val ticker = nameOfFile.substring(firstIndex, lastIndex)

      import sqlContext.implicits;
      fileRDD.map(x =>
      {
        val str = x.split(",")
        new VarData(str(0), str(1).toDouble, str(2).toDouble,
str(3).toDouble, str(4).toDouble, str(5).toLong, str(6).toDouble,
ticker)
      })
      .toDF
      .write.parquet(args(1) + "/" + date + "/stocks/"+ticker)
    })


    val factorsList = fs.listStatus(new Path(args(0) + "/" + date + "/
factors")).map(fileName => {

      val fileRDD =
sc.textFile(fileName.getPath.toString()).mapPartitionsWithIndex
{ (idx, iter) => if (idx == 0) iter.drop(1) else iter }

      val nameOfFile = fileName.getPath.toString()
      val firstIndex = nameOfFile.lastIndexOf("/") + 1
      val lastIndex = nameOfFile.indexOf(".", firstIndex)
      val ticker = nameOfFile.substring(firstIndex, lastIndex)

      import sqlContext.implicits;
      fileRDD.map(x =>
      {
        val str = x.split(",")
        new VarData(str(0), str(1).toDouble, str(2).toDouble,
str(3).toDouble, str(4).toDouble, str(5).toLong, str(6).toDouble,
ticker)
      })
      .toDF
      .write.parquet(args(1) + "/" + date + "/factors/"+ticker)
    })

  }
```

```
}
```

The ToIgnite file is run to feed the output to Ignite which stores the data and retrieves the data faster than the direct method.

```
package com.innovative

import java.text.SimpleDateFormat
import java.util.Calendar
import org.apache.ignite.configuration.IgniteConfiguration
import org.apache.ignite.spark.IgniteContext
import org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext
import org.apache.ignite.Ignite
import org.apache.ignite.Ignition

object ToIgnite {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf()
    val sc = new SparkContext(conf);
    val sqlContext = new SQLContext(sc)

    import sqlContext.implicits._

    val format = new SimpleDateFormat("y-MM-dd")
    val cal = Calendar.getInstance()
    cal.add(Calendar.DATE, -1)
    val formatter = new SimpleDateFormat("EEE")
    val day = formatter.format(cal.getTime())

    /*
     * Skip weekends
     */
    def getDayOfWeek(days: String): String = {
      if (days.equalsIgnoreCase("Sun")) {
        cal.add(Calendar.DATE, -2)
        return format.format(cal.getTime())
      } else {
        return format.format(cal.getTime())
      }
    }

    val date = getDayOfWeek(day)

    //Ignition.start();
    //Ignition.setClientMode(true);

      val ic = new IgniteContext[String, String](sc,() => {
      val cfg = new IgniteConfiguration();
      val tc = new TcpDiscoverySpi();
      tc.setJoinTimeout(120000);
      cfg.setDiscoverySpi(tc);
      cfg})
```

```
    /*
     * saving the stocks data into ignite
     * */
    val stocksSharedRDD = ic.fromCache(date+"_stockdata")
    val stocks = sc.wholeTextFiles(args(0)+"/"+date+"/stocks").map {
      case (name, content) =>
        {
          val first = name.lastIndexOf("/") + 1;
          val last = name.indexOf(".", first);
          val fileName = name.substring(first, last);
          (fileName, content)
        }
    }

    stocksSharedRDD.savePairs(stocks)

    /*
     * save the factors data into ignite
     * */
    val factorsSharedRDD = ic.fromCache(date+"_factordata")
    val factors = sc.wholeTextFiles(args(0)+"/"+date+"/factors").map {
      case (name, content) =>
        {
          val first = name.lastIndexOf("/") + 1;
          val last = name.indexOf(".", first);
          val fileName = name.substring(first, last);
          (fileName, content)
        }
    }

    factorsSharedRDD.savePairs(factors)
    ic.close()

    System.exit(0)

  }
}
```

The OptimizedToIgnite file is an optimized version which is run to feed the output to Ignite which stores the data and retrieves the data faster than the direct method.

```
package com.innovative

import java.text.SimpleDateFormat
import java.util.Calendar
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.FileSystem
import org.apache.hadoop.fs.Path
import org.apache.ignite.configuration.IgniteConfiguration
import org.apache.ignite.spark.IgniteContext
import org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext

object OptimizedToIgnite {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf()
    val sc = new SparkContext(conf)
    val sqlContext = new SQLContext(sc)
    val configuration = new Configuration()
    val fs = FileSystem.get(configuration)
    import sqlContext.implicits._

    val format = new SimpleDateFormat("y-MM-dd")
    val cal = Calendar.getInstance()
    cal.add(Calendar.DATE, -1)
    val formatter = new SimpleDateFormat("EEE")
    val day = formatter.format(cal.getTime())

    /*
     * Skip weekends
     */
    def getDayOfWeek(days: String): String = {
      if (days.equalsIgnoreCase("Sun")) {
        cal.add(Calendar.DATE, -2)
        return format.format(cal.getTime())
      } else {
        return format.format(cal.getTime())
      }
    }

    val date = getDayOfWeek(day)

    val ic = new IgniteContext[String, String](sc, () => {
      val cfg = new IgniteConfiguration();
      val tc = new TcpDiscoverySpi();
      tc.setJoinTimeout(120000);
      cfg.setDiscoverySpi(tc);
      cfg
    })
```

```
    /*
     * saving the stocks data into ignite
     * */
    val stocksSharedRDD = ic.fromCache(date + "_stockdata")
    val stocksList = fs.listStatus(new Path(args(0) + "/" + date + "/
stocks")).map(fileName => {

      val fileRDD =
sc.textFile(fileName.getPath.toString()).mapPartitionsWithIndex
{ (idx, iter) => if (idx == 0) iter.drop(1) else iter }

      val nameOfFile = fileName.getPath.toString()
      val firstIndex = nameOfFile.lastIndexOf("/") + 1
      val lastIndex = nameOfFile.indexOf(".", firstIndex)
      val ticker = nameOfFile.substring(firstIndex, lastIndex)

      val stocks = fileRDD.map(line =>
        {
          (line + "," + ticker)
        })
      stocksSharedRDD.saveValues(stocks)
    })

    /*
     * save the factors data into ignite
     * */
    val factorsSharedRDD = ic.fromCache(date + "_factordata")
    val factorsList = fs.listStatus(new Path(args(0) + "/" + date + "/
factors")).map(fileName => {

      val fileRDD =
sc.textFile(fileName.getPath.toString()).mapPartitionsWithIndex
{ (idx, iter) => if (idx == 0) iter.drop(1) else iter }

      val nameOfFile = fileName.getPath.toString()
      val firstIndex = nameOfFile.lastIndexOf("/") + 1
      val lastIndex = nameOfFile.indexOf(".", firstIndex)
      val ticker = nameOfFile.substring(firstIndex, lastIndex)

      val factors = fileRDD.map(line =>
        {
          (line + "," + ticker)
        })
      factorsSharedRDD.saveValues(factors)
    })

    System.exit(0)

  }
```

```
}
```

The VarRunner is run to calculate the Var value with a confidence level and displays the results to the executor of the program.

```
package com.innovative.risk

import java.text.SimpleDateFormat

import com.github.nscala_time.time.Imports._
import
org.apache.commons.math3.distribution.MultivariateNormalDistribution
import org.apache.commons.math3.random.MersenneTwister
import org.apache.commons.math3.stat.correlation.Covariance
import
org.apache.commons.math3.stat.regression.OLSMultipleLinearRegression
import org.apache.ignite.Ignition
import org.apache.ignite.configuration._
import org.apache.ignite.spark._
import org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.rdd.RDD
import org.apache.spark.storage.StorageLevel
import org.joda.time.DateTime

import scala.collection.mutable.ArrayBuffer

object VarRunner {
  def main(args: Array[String]): Unit =
    {

      val sc = new SparkContext(new SparkConf().setAppName("VaR
Reference App"))

      val df = org.joda.time.format.DateTimeFormat.forPattern("yyyy-
MM-dd")
      val varDateStr = if (args.length < 1) new
DateTime().toString(df) else args(0)
      val varDate = DateTime.parse(varDateStr, df)
      println("Calculating VaR for the date: " + varDate)

      /*
       * Starting Reading Stocks and Factors data
       */
      val (stocksReturns, factorsReturns) =
readStocksAndFactors(varDate, sc)

      val numTrials = 10000000
      val parallelism = 1000
      val baseSeed = 1496L
      /*
       * Starting trials
       */

      val trials = computeTrialReturns(stocksReturns, factorsReturns,
```

```
sc, baseSeed, numTrials, parallelism)
      trials.cache()

      /*
       * Calculating VaR
       */
      val valueAtRisk = fivePercentVaR(trials)
      val varConfidenceInterval =
bootstrappedConfidenceInterval(trials, fivePercentVaR, 100, .05)
      println("VaR 5%: " + valueAtRisk)
      println("VaR confidence interval: " + varConfidenceInterval)
    }

  def computeTrialReturns(stocksReturns:
Seq[Array[Double]],factorsReturns: Seq[Array[Double]],sc:
SparkContext,baseSeed: Long, numTrials: Int,parallelism: Int):
  RDD[Double] =
  {

    val factorMat = factorMatrix(factorsReturns)
    val factorCov = new
Covariance(factorMat).getCovarianceMatrix().getData()
    val factorMeans = factorsReturns.map(factor => factor.sum /
factor.size).toArray
    val factorFeatures = factorMat.map(featurize)
    val factorWeights = computeFactorWeights(stocksReturns,
factorFeatures)
    val bInstruments = sc.broadcast(factorWeights)

    /*
     * Generate different seeds so that our simulations don't all end
up with the same results
     * */

    val seeds = (baseSeed until baseSeed + parallelism)
    val seedRdd = sc.parallelize(seeds, parallelism)

    /*
     * Main computation: run simulations and compute aggregate return
for each
     * */
    val returns = seedRdd.flatMap(
      trialReturns(_, numTrials / parallelism, bInstruments.value,
factorMeans, factorCov))

    returns
  }

  def trialReturns(seed: Long, numTrials: Int,instruments:
Seq[Array[Double]],factorMeans: Array[Double],factorCovariances:
```

```
Array[Array[Double]]): Seq[Double] = {

    /**
     *
     * Generate the random number using Mersenne Twister pseudorandom
number generator  algorithm
     *
     */
    val rand = new MersenneTwister(seed)

    val multivariateNormal = new MultivariateNormalDistribution(rand,
factorMeans, factorCovariances)

    val trialReturns = new Array[Double](numTrials);

    for (i <- 0 until numTrials) {

      /**
       *
       * Taking a sample of data using factors means and covariances
       *
       */

      val trialFactorReturns = multivariateNormal.sample()
      val trialFeatures = VarRunner.featurize(trialFactorReturns)
      trialReturns(i) = trialReturn(trialFeatures, instruments)
    }
    trialReturns
  }

  /**
   * Calculate the full return of the portfolio under particular trial
conditions.
   */
  def trialReturn(trial: Array[Double], instruments:
Seq[Array[Double]]): Double = {
    var totalReturn = 0.0
    for (instrument <- instruments) {
      totalReturn += instrumentTrialReturn(instrument, trial)
    }
    totalReturn
  }

  /**
   * Calculate the return of a particular instrument under particular
trial conditions.
   */
  def instrumentTrialReturn(instrument: Array[Double], trial:
Array[Double]): Double = {
    var instrumentTrialReturn = instrument(0)
```

```
    var i = 0
    while (i < trial.length) {
      instrumentTrialReturn += trial(i) * instrument(i + 1)
      i += 1
    }
    instrumentTrialReturn
  }

  def computeFactorWeights(stocksReturns:
Seq[Array[Double]],factorFeatures: Array[Array[Double]])
  : Array[Array[Double]] =
  {
    val models = stocksReturns.map(linearModel(_, factorFeatures))
    val factorWeights = Array.ofDim[Double](stocksReturns.length,
factorFeatures.head.length + 1)
    for (s <- 0 until stocksReturns.length) {
      factorWeights(s) = models(s).estimateRegressionParameters()
    }
    factorWeights
  }

  def featurize(factorReturns: Array[Double]): Array[Double] = {
    val squaredReturns = factorReturns.map(x => {math.signum(x) * x *
x})
    val squareRootedReturns = factorReturns.map(x => math.signum(x) *
math.sqrt(math.abs(x)))
    squaredReturns ++ squareRootedReturns ++ factorReturns
  }


  def twoWeekReturns(history: Array[(DateTime, Double)]):
Array[Double] = {
    history.sliding(10).map(window => window.last._2 -
window.head._2).toArray
  }

  def linearModel(instrument: Array[Double], factorMatrix:
Array[Array[Double]]): OLSMultipleLinearRegression = {
    val regression = new OLSMultipleLinearRegression()
    regression.newSampleData(instrument, factorMatrix)
    regression
  }

  def factorMatrix(histories: Seq[Array[Double]]):
Array[Array[Double]] = {
    val mat = new Array[Array[Double]](histories.head.length)
    for (i <- 0 until histories.head.length) {
      mat(i) = histories.map(_(i)).toArray
    }
```

```
    mat
  }

  def fivePercentVaR(trials: RDD[Double]): Double = {

    val topLosses = trials.takeOrdered(math.max(trials.count().toInt /
20, 1))
    topLosses.last
  }

  def fivePercentCVaR(trials: RDD[Double]): Double = {
    val topLosses = trials.takeOrdered(math.max(trials.count().toInt /
20, 1))
    topLosses.sum / topLosses.length
  }

  def bootstrappedConfidenceInterval(
    trials: RDD[Double],
    computeStatistic: RDD[Double] => Double,
    numResamples: Int,
    pValue: Double): (Double, Double) = {
    val stats = (0 until numResamples).map { i =>
      val resample = trials.sample(true, 1.0)
      computeStatistic(resample)
    }.sorted
    val lowerIndex = (numResamples * pValue / 2 - 1).toInt
    val upperIndex = math.ceil(numResamples * (1 - pValue / 2)).toInt
    (stats(lowerIndex), stats(upperIndex))
  }

  def readStocksAndFactors(varDate: DateTime, sc: SparkContext):
(Seq[Array[Double]], Seq[Array[Double]]) =
    {
      val start = varDate.minusYears(5)
      val end = varDate
      val format = new SimpleDateFormat("yyyy-MM-dd")

      val stockFile =
varDate.toString(DateTimeFormat.forPattern("yyyy-MM-dd")) +
"_stockdata"
      val factorFile =
varDate.toString(DateTimeFormat.forPattern("yyyy-MM-dd")) +
"_factordata"
      //val commFile =
varDate.toString(DateTimeFormat.forPattern("yyyy-MM-dd")) +
"_commdata"

      try {
        var parseStockFile: RDD[(String, Array[(DateTime, Double)])] =
null;
```

```
        var parseFactorFile: RDD[(String, Array[(DateTime, Double)])]
= null

        println("Reading stocks and factors for VarRunner")
        if (Ignition.allGrids() == null) {

          println("ignition.allgrids == null scenario = true")
          val igniteContext = new IgniteContext[String, String](sc, ()
=> {
            val cfg = new IgniteConfiguration();
            val tc = new TcpDiscoverySpi();
            tc.setJoinTimeout(120000);
            cfg.setDiscoverySpi(tc);
            cfg
          })
          val getStockFileRdd = igniteContext.fromCache(stockFile)
          val getFactorFileRdd = igniteContext.fromCache(factorFile)

          parseStockFile = getStockFileRdd.map(kv =>
(kv._1.substring(kv._1.lastIndexOf("/") + 1).stripSuffix(".csv"),
readStockHistory(kv._2)))
          parseFactorFile = getFactorFileRdd.map(kv =>
(kv._1.substring(kv._1.lastIndexOf("/") + 1).stripSuffix(".csv"),
readStockHistory(kv._2)))

          igniteContext.close()
        } else {

          println("ignition.allgrids == null scenario is false")
          val sqlContext = new org.apache.spark.sql.SQLContext(sc)

          val stockParquetDf = sqlContext.read.parquet("/projects/
RefApps/OnDemandVaR/data/var/" +
varDate.toString(DateTimeFormat.forPattern("yyyy-MM-dd")) + "/stocks")
          val factorParquetDf = sqlContext.read.parquet("/projects/
RefApps/OnDemandVaR/data/var/" +
varDate.toString(DateTimeFormat.forPattern("yyyy-MM-dd")) + "/
factors")

          println("Reading from parquet files is completed!!!!")

          if (stockParquetDf.rdd.count() == 0 ||
factorParquetDf.rdd.count() == 0) {
            println("Sorry, data is not available for this date")
            sys.exit(-1)
          }

          /**
           * After getting the stock data, pick date and open ( the
value of the stock at which it starts on a day)
```

```
           * So finally you will have all stocks data with their open
value
           */

          parseStockFile = stockParquetDf.rdd.map(row =>
(row.getString(0), row.getString(1)))
          .map(kv => (kv._1, readStockHistory(kv._2)))
          .persist(StorageLevel.MEMORY_ONLY)

          parseFactorFile = factorParquetDf.rdd.map(row =>
(row.getString(0), row.getString(1)))
            .map(kv => (kv._1, readStockHistory(kv._2)))
          //parseCommFile = getCommFileRddHdfs.map(kv => (kv._1,
readCommHistory(kv._2)))


        }

        /*
         * filter and trim data
         */

        /**
         *
         * Check if every ticker has full 5 Years data or not.
         */

        val rawStocks = parseStockFile.map(k => k._2).filter(kv =>
kv.size >= 260 * 5 + 10)
        val stocks = rawStocks.map(trimToRegion(_, start, end))
        .map(fillInHistory(_, start, end))
        .collect()

        val rawMktFactors = parseFactorFile.map(k =>
k._2).map(trimToRegion(_, start, end))
        .map(fillInHistory(_, start, end))
        .collect()

        //val rawCommFactors = parseCommFile.map(k =>
k._2).map(trimToRegion(_, start, end)).map(fillInHistory(_, start,
end)).collect()
        //val factors = (rawMktFactors ++ rawCommFactors)

        val factors = rawMktFactors
        val stockReturns = stocks.map(twoWeekReturns).toSeq

        val factorReturns = factors.map(twoWeekReturns).toSeq

        //val x = factorReturns.flatMap { x => x }
        //factorReturns.foreach { x => x.foreach(println)}
```

```
        (stockReturns, factorReturns)

      } catch {
        case e: Exception =>
          println("Errored loading files, StockFile:" + stockFile + ",
FactorFile:" + factorFile);
          println("Exception:" + e.getMessage);
          e.printStackTrace();
          sys.exit(-2)
      }

    }

  /**
   * Reads a file with stock history in the Yahoo format
   *
   */
  def readStockHistory(fileStr: String): Array[(DateTime, Double)] =
    {
      val format = new SimpleDateFormat("yyyy-MM-dd")
      val lines = fileStr.lines.toSeq
      // read all the lines except first line
      lines.tail.map(
         line =>
          {
            val cols = line.split(',')
            val date = new DateTime(format.parse(cols(0)))
            val value = cols(1).toDouble
            // Get first two columns. 1 - Get Date which is first
column 2 - Get Stock (Open) Value
            (date, value)
          }
      ).reverse.toArray
    }

  def trimToRegion(history: Array[(DateTime, Double)], start:
DateTime, end: DateTime): Array[(DateTime, Double)] =
    {
      var trimmed = history.dropWhile(_._1 < start).takeWhile(_._1 <=
end)
      if (trimmed.head._1 != start) {
        trimmed = Array((start, trimmed.head._2)) ++ trimmed
      }
      if (trimmed.last._1 != end) {
        trimmed = trimmed ++ Array((end, trimmed.last._2))
      }
      trimmed
    }
```

```
  /**
   * Given a timeseries of values of an instruments, returns a
timeseries between the given
   * start and end dates with all missing weekdays imputed. Values are
imputed as the value on the
   * most recent previous given day.
   */
  def fillInHistory(history: Array[(DateTime, Double)], start:
DateTime, end: DateTime): Array[(DateTime, Double)] =
    {
      var cur = history

      val filled = new ArrayBuffer[(DateTime, Double)]()
      var curDate = start
      while (curDate < end) {

        if (cur.tail.nonEmpty && cur.tail.head._1 == curDate) {

          cur = cur.tail
        }

        filled += ((curDate, cur.head._2))

        curDate += 1.days
        // Skip weekends
        if (curDate.dayOfWeek().get > 5) {curDate += 2.days}
      }
      filled.toArray
    }

  def readCommHistory(fileStr: String): Array[(DateTime, Double)] =
    {
      val format = new SimpleDateFormat("MMM d, yyyy")
      val lines = fileStr.lines.toSeq
      lines.map(line => {
        val cols = line.split('\t')
        val date = new DateTime(format.parse(cols(0)))
        val value = cols(1).toDouble
        (date, value)
      }).reverse.toArray
    }

}
```

The OptimizedVarRunner is an optimized version that is run to calculate the Var value with a confidence level and save the results to the database.

```
package com.innovative.risk

import java.text.SimpleDateFormat

import com.github.nscala_time.time.Imports.{DateTimeFormat, LocalDate,
richAbstractPartial, richInt, richLocalDate}
import
org.apache.commons.math3.distribution.MultivariateNormalDistribution
import org.apache.commons.math3.random.MersenneTwister
import
org.apache.commons.math3.stat.regression.OLSMultipleLinearRegression
import org.apache.ignite.Ignition
import org.apache.ignite.configuration.IgniteConfiguration
import org.apache.ignite.spark.IgniteContext
import org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi
import org.apache.log4j.Logger
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.distributed.RowMatrix
import org.apache.spark.rdd.RDD
import org.apache.spark.rdd.RDD.rddToOrderedRDDFunctions
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
import org.apache.spark.sql.hive.HiveContext
import org.apache.spark.sql.types._
import org.apache.spark.sql.{DataFrame, Row, functions => func}
import org.apache.spark.storage.StorageLevel
import scopt.OptionParser

import scala.collection.mutable.ArrayBuffer

case class VarMetrics(varDate: String,
                      NumTrials: Int,
                      Parallelism: Int,
                      VarPercentile: Int,
                      CalculateConfidence: String,
                      VarValue: Double,
                      ConfidenceInterval: (Double, Double),
                      DataFrom: String,
                      IsIgniteOn: String,
                      JobRunTime: Long,
                      Portfoilio: String,
                      ListOfStocks: Array[String])

object OptimizedVarRunner {
  object Config {
    var varDate = LocalDate.now().minusDays(1).toString()
    var NumTrials: Int = 1000000
    var Parallelism: Int = 1000
    var VarPercentile: Int = 5
```

```
    var CalculateConfidence: String = "no"
    var path: String = "/projects/RefApps/OnDemandVaR/data/var/"
    var portfolio: String = "all_stocks"
    var stocks: String = ""
  }
  def main(args: Array[String]): Unit =
    {

      val log = Logger.getLogger(OptimizedVarRunner.getClass)

      val parser = new OptionParser("scopt") {
        opt("varDate", "Date for calculating VaR", { v: String =>
Config.varDate = v })
        opt("numTrials", "Num of Trials", { v: String =>
Config.NumTrials = v.toInt })
        opt("parallelism", "Parallelism", { v: String =>
Config.Parallelism = v.toInt })
        opt("varPercentile", "Percentile to calculate VaR which is b/w
1 - 10", { v: String => Config.VarPercentile = v.toInt })
        opt("calculateConfidence", "Boolean to specify whether to
calculate confidence interval", { v: String =>
Config.CalculateConfidence = v })
         opt("path", "path to parquet files", { v: String =>
Config.path = v })
        opt("portfolio", "specify a portfolio to run the VaR", { v:
String => Config.portfolio = v })
        opt("stocks", "list the stocks that make up a portfolio", { v:
String => Config.stocks = v })
      }
      parser.parse(args, Config)

      val sc = new SparkContext(new SparkConf().setAppName("VaR
Reference App"))
      val startTime = System.currentTimeMillis()

      /*
       * Input parameters
       */

      val date = Config.varDate
      val numTrials = Config.NumTrials
      val parallelism = Config.Parallelism
      val percentile = Config.VarPercentile
      var confidence = Config.CalculateConfidence
      val baseSeed = 1496L

      if (numTrials < 1000000 || (percentile > 10 && percentile < 1)
|| (Config.portfolio.equalsIgnoreCase("custom") &&
Config.stocks.equalsIgnoreCase(""))) {
        log.info("Sorry, we cannot calculate VaR if the num of Trials
```

```
is less than 10000000 or if the VaR calculation % value is not in
between 1 - 10. If the VaR is calculated against a custom portfolio,
then list the stocks with comma separated values. Please check the
values and try again.")
        System.exit(0)
      }

      val df = org.joda.time.format.DateTimeFormat.forPattern("yyyy-
MM-dd")
      var varDate = LocalDate.parse(date, df)
      log.info("Calculating VaR for the date: " + varDate)

      /*
       * Starting Reading Stocks and Factors data
       */
       val listOfStocks = Config.stocks.split(",")
      val (stocksReturns, factorsReturns, dataFrom, isIgniteOn) =
readStocksAndFactors(varDate, sc, log, Config.portfolio, listOfStocks)

      /*
       * Starting trials
       */
      val trialReturns = computeTrialReturns(stocksReturns,
factorsReturns, sc, baseSeed, numTrials, parallelism, log)
      //.persist(StorageLevel.MEMORY_AND_DISK)

      /*
       * Calculating VaR
       */
      val fivePercentValue = fivePercentVaR(trialReturns, percentile)
      log.info("VaR " + percentile + "%: " + fivePercentValue)

      /*
       * calculating VaR confidence interval
       */
      val varConfidenceInterval = getConfidence(confidence,
trialReturns, percentile)
      log.info("VaR confidence interval: " + varConfidenceInterval)

      /*
       * pushing results to hbase to persist
       */
      val endTime = System.currentTimeMillis()
      val VarMetricsObj = new VarMetrics(date, numTrials, parallelism,
percentile, confidence, fivePercentValue, varConfidenceInterval,
dataFrom, isIgniteOn, (endTime - startTime), Config.portfolio,
listOfStocks)
      val returnObj = new ResultsToHBase()
      returnObj.getHBaseMethod(VarMetricsObj)
```

```
    }

  def getConfidence(confidenceBoolean: String, trialReturns:
RDD[Double], percentile: Int): (Double, Double) = {
    if (confidenceBoolean.equalsIgnoreCase("Yes")) {
      return bootstrappedConfidenceInterval(trialReturns,
fivePercentVaR, 100, (percentile / 100), percentile)
    } else {
      return (0.0, 0.0)
    }
  }
  def fivePercentVaR(trials: RDD[Double], percentile: Int): Double = {
    val percentage = ((trials.count().toInt) * percentile) / 100
    val topLosses = trials.takeOrdered(math.max(percentage, 1))
    topLosses.last
  }

  def bootstrappedConfidenceInterval(
    trials: RDD[Double],
    computeStatistic: (RDD[Double], Int) => Double,
    numResamples: Int,
    pValue: Double,
    pencentile: Int): (Double, Double) = {
    val stats = (0 until numResamples).map { i =>
      val resample = trials.sample(true, 1.0)
      computeStatistic(resample, pencentile)
    }.sorted
    val lowerIndex = (numResamples * pValue / 2 - 1).toInt
    val upperIndex = math.ceil(numResamples * (1 - pValue / 2)).toInt
    (stats(lowerIndex), stats(upperIndex))
  }

  def readStocksAndFactors(varDate: LocalDate, sc: SparkContext, log:
Logger, portfolio: String, stocks: Array[String]):
(RDD[Array[Double]], RDD[Array[Double]], String, String) =
    {
      val start = varDate.minusYears(5)
      val end = varDate
      val format = new SimpleDateFormat("yyyy-MM-dd")
      val stockFile =
varDate.toString(DateTimeFormat.forPattern("yyyy-MM-dd")) +
"_stockdata"
      val factorFile =
varDate.toString(DateTimeFormat.forPattern("yyyy-MM-dd")) +
"_factordata"
      var parseStockFile: DataFrame = null
      var parseFactorFile: DataFrame = null
      var dataFrom = ""
      var isIgniteOn = "no"
      val sqlContext = new HiveContext(sc)
```

```
      import sqlContext.implicits._

      println("reading stocks and factors for Optimized Var Runner")
      if (Ignition.allGrids() == null) {
        log.info("\n Successfully connected to Ignite. \n")
        isIgniteOn = "yes"

        println("ignition.allgrids != null scenario is true")
        try {
          val igniteContext = new IgniteContext[String, String](sc, ()
=> {
            val cfg = new IgniteConfiguration();
            val tc = new TcpDiscoverySpi();
            tc.setJoinTimeout(30000);
            cfg.setDiscoverySpi(tc);
            cfg
          })

          val getStockFileRdd = igniteContext.fromCache(stockFile)
          val getFactorFileRdd = igniteContext.fromCache(factorFile)

          if (getStockFileRdd.count() != 0 &&
getFactorFileRdd.count() != 0) {
            dataFrom = "igniteCache"
            val schema = StructType(Seq(
              StructField("Date", StringType, nullable = true),
              StructField("Open", DoubleType, nullable = true),
              StructField("High", DoubleType, nullable = true),
              StructField("Low", DoubleType, nullable = true),
              StructField("Close", DoubleType, nullable = true),
              StructField("Volume", LongType, nullable = true),
              StructField("Adj_Close", DoubleType, nullable = true),
              StructField("ticker", StringType, nullable = true)))

            parseStockFile = createDF(getStockFileRdd.map(x => x._2),
schema, sqlContext)
            parseFactorFile = createDF(getFactorFileRdd.map(x =>
x._2), schema, sqlContext)
          }

        } catch {
          case e: Exception =>
            log.info("Errored loading files from ignite.");
            log.info("Exception:" + e.getMessage);
        }
      }
      if (parseStockFile == null || parseFactorFile == null ||
parseStockFile.rdd.count() == 0 || parseStockFile.rdd.count() == 0 ||
parseFactorFile.rdd.count() == 0) {
        log.info("\n History data is not available in cache for the
```

```
given date, reading from parquet files. \n")
        dataFrom = "hdfs"

        println("apparently data is not available in cache so its
reading from parquet files")
        println("the other case where parse stocks and factors are
null")

        val stockParquetDf = sqlContext.read.parquet("/projects/
RefApps/OnDemandVaR/data/var/" +
varDate.toString(DateTimeFormat.forPattern("yyyy-MM-dd")) + "/stocks/
*")
        val factorParquetDf = sqlContext.read.parquet("/projects/
RefApps/OnDemandVaR/data/var/" +
varDate.toString(DateTimeFormat.forPattern("yyyy-MM-dd")) + "/factors/
*")

        if (stockParquetDf.rdd.count() == 0 ||
factorParquetDf.rdd.count() == 0) {
          log.info("Sorry, data is not available for this date")
          log.debug("Data is not available for the date: " + varDate)
          sys.exit(-1)
        }

        parseStockFile =
stockParquetDf.persist(StorageLevel.MEMORY_AND_DISK)
        parseFactorFile =
factorParquetDf.persist(StorageLevel.MEMORY_AND_DISK)

      }

      /*
       * Get stocks data for the given portfolio
       */
      if (portfolio.equalsIgnoreCase("XLK") ||
portfolio.equalsIgnoreCase("XLF")) {
        log.info("calculating VaR for the portfolio: "+ portfolio)
        val list = sqlContext.sparkContext.textFile("/projects/
RefApps/OnDemandVaR/portfolio/" + portfolio + ".txt").collect()
        val usersToKeepBD = sc.broadcast(list.toSet)
        val checkUser = udf((id: String) =>
usersToKeepBD.value.contains(id))
        parseStockFile = parseStockFile.where(checkUser($"ticker"))
      } else if (portfolio.equalsIgnoreCase("custom")) {
        log.info("calculating VaR for the custom portfolio: "+ stocks)
        val usersToKeepBD = sc.broadcast(stocks.toSet)
        val checkUser = udf((id: String) =>
usersToKeepBD.value.contains(id))
        parseStockFile = parseStockFile.where(checkUser($"ticker"))
      }
```

```
      val parseStockFiles = parseStockFile.select("Date", "Open",
"ticker")
      val parseFactorFiles = parseFactorFile.select("Date", "Open",
"ticker")

      val stockReturns = trimData(parseStockFiles, sqlContext, start,
end)
      //.persist(StorageLevel.MEMORY_AND_DISK)
      val factorReturns = trimData(parseFactorFiles, sqlContext,
start, end)
      //.persist(StorageLevel.MEMORY_AND_DISK)

      (stockReturns, factorReturns, dataFrom, isIgniteOn)

    }

  def createDF(row: RDD[String], schema: StructType, sqlContext:
HiveContext): DataFrame = {
    val result = row.map(line => {
      val str = line.split(",")
      Row(str(0), str(1).toDouble, str(2).toDouble, str(3).toDouble,
str(4).toDouble, str(5).toLong, str(6).toDouble, str(7));
    })
    val resDF = sqlContext.createDataFrame(result, schema)
    resDF
  }

  def trimData(parseFile: DataFrame, sqlContext: HiveContext, start:
LocalDate, end: LocalDate): RDD[Array[Double]] = {
    import org.apache.spark.sql._
    import sqlContext.implicits._
    val windowSpec =
Window.partitionBy($"ticker").orderBy($"Date".asc)

    val df2 = parseFile.withColumn("cnt",
func.count("ticker").over(windowSpec))
      .where($"cnt" >= (260 * 5 + 10))
      .where($"Date" >= start.toString() && $"Date" <= end.toString())
      .drop($"cnt")
    // .persist(StorageLevel.MEMORY_AND_DISK)

    val groupData = df2.rdd.groupBy { x =>
x.getString(2) }.sortByKey(true).persist()

    val trimmedData = groupData.map {
      case (ticker, data) =>
        {
          var list = data.toList
          if (list.head.get(0) != start.toString()) {
```

```
            list = Row(start.toString(), list.head.get(1),
list.head.get(2)) +: list
          }
          if (list.last.get(0) != end.toString()) {
            list = list :+ Row(end.toString(), list.last.get(1),
list.last.get(2))
          }
          (ticker, list.toArray)
        }
    }
    //.persist(StorageLevel.MEMORY_AND_DISK)

    val completeData = trimmedData.map { case (x, y) =>
(fillInHistory(y, start, end)) }.persist(StorageLevel.MEMORY_AND_DISK)

    completeData.map(data2 =>
      {
        data2.sliding(10).map(window => window.last.getDouble(1) -
window.head.getDouble(1)).toArray
      })
    //.persist(StorageLevel.MEMORY_AND_DISK)
  }

  def fillInHistory(history: Array[Row], start: LocalDate, end:
LocalDate): Array[Row] =
    {
      var cur = history
      var filled = new ArrayBuffer[Row]()
      var curDate = start
      while (curDate < end) {
        if (cur.tail.nonEmpty && cur.tail.head.get(0) ==
curDate.toString()) {
          cur = cur.tail
        }
        filled = Row(curDate, cur.head.get(1), cur.head.get(2)) +:
filled
        curDate += 1.days

        if (curDate.dayOfWeek().get > 5) curDate += 2.days
      }
      filled.reverse.toArray
    }

  def computeTrialReturns(stocksReturns: RDD[Array[Double]],
factorsReturns: RDD[Array[Double]], sc: SparkContext, baseSeed: Long,
numTrials: Int, parallelism: Int, log: Logger): RDD[Double] = {

    val factorMat =
sc.parallelize(factorsReturns.collect.transpose.toSeq);
    val factorFeatures = factorMat.map(featurize) //deriving
```

```
additional factor features
    val vectors = factorMat.map(s => Vectors.dense(s))
    val mat = new RowMatrix(vectors).computeCovariance()
    val factorMeans = factorsReturns.map(factor => factor.sum /
factor.size).collect
    val factorWeights = computeFactorWeightsWithRDD(stocksReturns,
factorFeatures.collect())
    //each executor will have full copy of instrument data, which
helps to calculate total loss for each trial on single machine without
aggregation
    val bInstruments = sc.broadcast(factorWeights.collect)
    val matrix: Array[Array[Double]] = Array.ofDim[Double]
(mat.numRows, mat.numCols)

    for {
      i <- 0 until mat.numRows
      j <- 0 until mat.numCols
    } matrix(i)(j) = mat(i, j)

    /*
     * We use Patition-by-trials approach.
     * So we need different seed in each partition so that each
partition generates different trials
     */

    val seeds = (baseSeed until baseSeed + parallelism)
    val seedRdd = sc.parallelize(seeds, parallelism)
    val numTrails = (numTrials / parallelism)

    //each seed will generate set of trials. Each element in
distributions would be array of trials.
    val distributions = seedRdd.flatMap(
      x => {
        val rand = new MersenneTwister(x);
        val multivariateNormal = new
MultivariateNormalDistribution(rand, factorMeans, matrix);
        val trialReturns = new Array[Array[Double]](numTrails);
        for (i <- 0 until numTrails) {
          trialReturns(i) = featurize(multivariateNormal.sample());
        }
        trialReturns
      })
    //.cache

    val returns = distributions.map { x =>
      {
        val totalReturn = new ArrayBuffer[Double];
        //val v1 = Vectors.dense(x);
        for (instrument <- bInstruments.value) {
          //val v2 = Vectors.dense(instrument)
```

```
          //Calculate the full return of the portfolio under
particular trial conditions.
          totalReturn += instrumentTrialReturn(instrument, x)
        }
        totalReturn.toArray;
      }
    }
    //.cache

    val finalReturns = returns.map { x => x.sum }
    log.debug("Total count of trials: " + finalReturns.count())
    finalReturns.cache
  }

  /**
   * Calculate the return of a particular instrument under particular
trial conditions.
   */
  def instrumentTrialReturn(instrument: Array[Double], trial:
Array[Double]): Double = {
    var instrumentTrialReturn = instrument(0)
    var i = 0
    while (i < trial.length) {
      instrumentTrialReturn += trial(i) * instrument(i + 1)
      i += 1
    }
    instrumentTrialReturn
  }

  def factorMatrix(histories: RDD[Array[Double]]):
Array[Array[Double]] = {
    histories.collect().transpose
  }

  def featurize(factorReturns: Array[Double]): Array[Double] = {
    val squaredReturns = factorReturns.map(x => math.signum(x) * x *
x)
    val squareRootedReturns = factorReturns.map(x => math.signum(x) *
math.sqrt(math.abs(x)))
    squaredReturns ++ squareRootedReturns ++ factorReturns
  }

  def computeFactorWeightsWithRDD(stocksReturns: RDD[Array[Double]],
factorFeatures: Array[Array[Double]]): RDD[Array[Double]] =
    {
      val models = stocksReturns.map(linearModel(_, factorFeatures))
      val factorWeights = Array.ofDim[Double]
(stocksReturns.count().toInt, factorFeatures.head.length + 1)
      models.map { x => x.estimateRegressionParameters() }
    }
```

```
  def linearModel(instrument: Array[Double], factorMatrix:
Array[Array[Double]]): OLSMultipleLinearRegression = {
    val regression = new OLSMultipleLinearRegression()
    regression.newSampleData(instrument, factorMatrix)
    regression
  }

}
```

The ResultsToHBase files defines the schema and stores the data to the HBase after calculation by OptimizedVarRunner.

```
package com.innovative.risk

import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.HTable
import org.apache.hadoop.hbase.client.Put
import org.apache.hadoop.hbase.util.Bytes
import org.apache.hadoop.io.ArrayWritable
import org.apache.hadoop.io.WritableUtils
import org.apache.hadoop.io.Writable

class ResultsToHBase {

  def getHBaseMethod(p: VarMetrics) = {

    val hbaseConf = HBaseConfiguration.create()
    hbaseConf.set("hbase.zookeeper.quorum", "10.21.10.202")
    val table = new HTable(hbaseConf, "VarResults");

    val put = new Put(Bytes.toBytes(p.varDate + "_" +
p.VarPercentile.toString()))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("VarValue"), Bytes.toBytes(p.VarValue.toString()))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("ConfidenceInterval"),
Bytes.toBytes(p.ConfidenceInterval.toString()))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("DataFrom"), Bytes.toBytes(p.DataFrom))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("IsIgniteOn"), Bytes.toBytes(p.IsIgniteOn))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("JobRunTime"), Bytes.toBytes(p.JobRunTime.toString() +
"ms"))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("VarDate"), Bytes.toBytes(p.varDate))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("NumTrials"), Bytes.toBytes(p.NumTrials.toString()))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("Parallelism"), Bytes.toBytes(p.Parallelism.toString()))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("VarPercentile"),
Bytes.toBytes(p.VarPercentile.toString()))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("CalculateConfidence"),
Bytes.toBytes(p.CalculateConfidence))
    put.addColumn(Bytes.toBytes("SummaryData"),
Bytes.toBytes("Portfolio"), Bytes.toBytes(p.Portfoilio))
    table.put(put);
  }
}
```

The downloadStocksBuild is a sbt file used to construct the downloadStocks part of the project along with all its dependencies.

```
name := “OnDemandVar"
version := "1.0"
scalaVersion := "2.10.4"

jarName in assembly:="download_fatjar.jar"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.0" %
"provided"
libraryDependencies += "org.apache.spark" %% "spark-sql" % "1.6.0" %
"provided"
libraryDependencies += "org.apache.spark" %% "spark-hive" % "1.6.0" %
"provided"
libraryDependencies += "com.databricks" %% "spark-csv" % "1.4.0"
libraryDependencies += "org.apache.ignite" %% "ignite-core" % "1.4.0"
cross CrossVersion.Disabled
libraryDependencies += "org.apache.ignite" %% "ignite-log4j" % "1.4.0"
cross CrossVersion.Disabled
libraryDependencies += "org.apache.ignite" % "ignite-spring" % "1.4.0"
libraryDependencies += "org.apache.ignite" %% "ignite-scalar" %
"1.4.0"
libraryDependencies += "org.apache.ignite" %% "ignite-spark" % "1.4.0"
excludeAll(
ExclusionRule(organization = "org.json4s"),
ExclusionRule(organization = "org.apache.spark")
)
mergeStrategy in assembly := {
  case m if m.toLowerCase.endsWith("manifest.mf")          =>
MergeStrategy.discard
  case m if m.toLowerCase.matches("meta-inf.*\\.sf$")      =>
MergeStrategy.discard
  case "log4j.properties"                                  =>
MergeStrategy.discard
  case m if m.toLowerCase.startsWith("meta-inf/services/") =>
MergeStrategy.filterDistinctLines
  case "reference.conf"                                    =>
MergeStrategy.concat
  case _                                                  =>
MergeStrategy.first
}
```

The VarRunnerBuild is a sbt file used to construct the VarRunner part of the project along with all its dependencies.

```
name := "OnDemandVaR"
version := "1.0"
scalaVersion := "2.10.4"
jarName in assembly := "OnDemandVaROptimized.jar"

libraryDependencies  += "org.apache.spark" % "spark-core_2.10" %
"1.5.2" % "provided"
libraryDependencies  += "org.apache.spark" % "spark-sql_2.10" %
"1.5.2" % "provided"
libraryDependencies  += "org.apache.spark" %% "spark-hive" % "1.5.2" %
"provided"
libraryDependencies += "org.apache.spark" %% "spark-mllib" % "1.5.2" %
"provided"
libraryDependencies += "com.github.scopt" % "scopt_2.10" % "2.1.0"
libraryDependencies += "org.apache.hbase" % "hbase-client" % "1.2.0" %
"provided" exclude("com.google.guava", "guava")
libraryDependencies += "org.apache.hbase" % "hbase-common" % "1.2.0" %
"provided" exclude("com.google.guava", "guava")
libraryDependencies += "org.apache.hbase" % "hbase-server" % "1.2.0" %
"provided" exclude("com.google.guava", "guava")
libraryDependencies += "org.apache.hbase" % "hbase-hadoop-compat" %
"1.2.0" % "provided" exclude("com.google.guava", "guava")
libraryDependencies += "org.apache.hbase" % "hbase-protocol" % "1.2.0"
% "provided" exclude("com.google.guava", "guava")
libraryDependencies += "org.apache.hbase" % "hbase-hadoop2-compat" %
"1.2.0" % "provided" exclude("com.google.guava", "guava")
libraryDependencies  += "org.apache.ignite" %% "ignite-spring" %
"1.4.0" cross CrossVersion.Disabled
libraryDependencies  += "org.apache.ignite" %% "ignite-core" % "1.4.0"
cross CrossVersion.Disabled
libraryDependencies += "org.apache.ignite" %% "ignite-log4j" % "1.4.0"
cross CrossVersion.Disabled
libraryDependencies += "org.apache.ignite" %% "ignite-spark" % "1.4.0"
excludeAll(
ExclusionRule(organization = "org.json4s"),
ExclusionRule(organization = "org.apache.spark")
)
libraryDependencies  += "org.scalanlp" %% "breeze-viz" % "0.11.2"
libraryDependencies  += "com.github.nscala-time" %% "nscala-time" %
"2.2.0"
libraryDependencies  += "org.apache.commons" % "commons-math3" % "3.5"
% "provided"



mergeStrategy in assembly := {
  case m if m.toLowerCase.endsWith("manifest.mf")          =>
MergeStrategy.discard
  case m if m.toLowerCase.matches("meta-inf.*\\.sf$")      =>
MergeStrategy.discard
```

```
  case "log4j.properties"                                     =>
MergeStrategy.discard
  case m if m.toLowerCase.startsWith("meta-inf/services/") =>
MergeStrategy.filterDistinctLines
  case "reference.conf"                                       =>
MergeStrategy.concat
  case _                                                     =>
MergeStrategy.first
}
```