The Report Committee for Timothy Allan Mauldin
Certifies that this is the approved version of the following report:

Crunch the Market

A Big Data Approach to Trading System Optimization

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Adnan Aziz

Joydeep Ghosh

Crunch the Market

A Big Data Approach to Trading System Optimization

by

Timothy Allan Mauldin, B.S.C.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2013

Dedication

For Lauren.

Acknowledgements

I would like to thank Professor Aziz for new ways of looking at problems, and for the time and attention he gave to help me with this report. I'd also like to thank Professor Joydeep Ghosh for getting me interested in things I would not have found on my own. Lastly, thanks to Jason Lu for some great ideas.

Abstract

Crunch the Market

A Big Data Approach to Trading System Optimization

Timothy Allan Mauldin, MSE

The University of Texas at Austin, 2013

Supervisor: Adnan Aziz

Due to the size of data needed, running software to analyze and tuning intraday trading strategies can take large amounts of time away from analysts, who would like to be able to evaluate strategies and optimize strategy parameters very quickly, ideally in the blink of an eye. Fortunately, Big Data technologies are evolving rapidly and can be leveraged for these purposes. These technologies include software systems for distributed computing, parallel hardware, and on demand computing resources in the cloud. This report presents a distributed software system for trading strategy analysis. It also demonstrates the effectiveness of Machine Learning techniques in decreasing parameter optimization workload. The results from tests run on two different commercial cloud service providers show linear scalability when analyzing intraday trading strategies.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Some believers in the efficient market hypothesis claim that available information is already incorporated in the price of a security at any time, therefore there are no ways to outperform the market with publicly available information [1, 2, 3]. They would claim that any profitable traders are successful by chance.

Princeton/Newport Partners, under Ed Thorp is an example that seems to contradict the efficient market hypothesis. Out of 230 months in business, Princeton/Newport Partners had 227 winning months [4]. During their years in business, three month Treasury Bill rates reached a high in 1981 with an average of 14.03 percent. Assuming that the efficient market hypothesis is correct, and purposefully exaggerating the likelihood of a winning month to be that of 1981 through the life of the fund, consider a coin with a 14.03 percent bias. The probability of getting at least 227 heads out of 230 is less than 1 in 10^{40}.

There are several other examples of this kind of success. As President of Axiom, Elwyn Berlekamp provided returns of over 55% to investors after fees in 1990. After Berlekamp sold his stake in Axiom, the fund continued to yield roughly 30% throughout the 1990's [5]. Other consistently high performing funds include Renaissance Technologies, Paulson & CO, and Soros Fund Management [6].

These exceptional returns are encouraging for those looking to profit from algorithmic trading, while the efficient market hypothesis indicates that it is useless to try. Fama [1, 2] identified "transactions costs, information that is not freely available to all investors, and disagreement among investors about the implications of given information," [2] as three potential sources of market inefficiency. This report presents a set of efficient computational tools to search for market inefficiencies.

PROBLEM

Any time spent running software to back test a trading strategy takes valuable time away from analysts. The granularity of the data required and the necessity to evaluate each trade in the data set being considered makes analyzing intraday trading strategies is a time consuming process.

To use data from September 26, 2012 as an example, there were 25,504,290 trades made on U.S. exchanges. In the software system presented in this paper using the single machine setup discussed in Chapter 4, it takes an average of 32 microseconds to process each record, resulting in a run time greater than 13 minutes. Likewise, if an analyst wants to optimize parameters for a trading strategy, the combinatorial explosion that results can make the time to run a back test impractical.

Use Cases

Back Testing over Multiple Days

Say Sally the Analyst thinks she has discovered a way to implement an intraday trading strategy. Sally needs to test that strategy on historical data on a trade-by-trade basis before her firm is ready to commit real money. To test her strategy over a significant time period could take an intolerable amount of time.

Parameter Optimization

The strategy that Sally needs to test has parameters that she needs to optimize; this typically takes one simulation run per value combination, per trading day. For example, say Sally wants to tune the following parameters: number of standard deviations and the number of advancing periods to consider to determine whether to take an action, fixed position size to take, what time at the end of the day to liquidate, and trailing stop percent.

For simplicity, if she wants to consider 20 discrete values for each, the test will require 20^5 (3,200,000) runs per day of trade data.

If there is a bug in her code and the run has to be repeated, it could take quite some time to find out whether her hypothesis is valid. Hopefully her firm is patient.

VISION

Current and improving techniques in Big Data and Machine Learning can help to provide this analysis in a scalable way. The goals of this project are to be able to add computing power as the size of the workload grows, and to implement some cleverness so that less computing power is required to find answers to questions about the efficacy of a strategy and about ideal parameter combinations.

SOLUTION

Testing intraday trading strategies is a natural fit for parallelization. Each day of trading data can frequently be evaluated independently, and that evaluation can be distributed over many machines as needed. There are also many established machine learning techniques that handle parameter optimization. The combination of distributing the evaluation workload with the reduction of that workload is the focus of this report.

SUMMARY OF CONTRIBUTIONS

This report presents a software solution that extends QUARTS [7], a trading algorithm analysis system designed by Jinxiang Lu and originally implemented in Python.

The software system developed in this report provides:

- Distribution of the computational workload of QUARTS to enable back testing multiple days in parallel.
- A Java translation of QUARTS for improved performance.
- Performance measurements of different distribution techniques.
- A discussion of experiences with two commercial cloud service providers.
- An overview of MapReduce and higher-level abstractions.
- A basic distributed Machine Learning approach for reducing the amount of work needed for parameter optimization in trading strategies.

REPORT ORGANIZATION

The report is divided into six chapters. Chapter 2 will present detail about the technology stack used in the work. In Chapter 3, the design and implementation of the system and parameter optimization strategy will be discussed. Chapter 4 will provide findings and experimental results that were produced during testing and analysis. Finally, Chapter 5 will discuss the results shown in Chapter 4, and will present a summary of lessons learned and some suggestions for future work.

Chapter 2: Domain-Specific and Distributed Systems Technologies

DOMAIN-SPECIFIC TECHNOLOGIES

QUARTS

QUARTS, which stands for "A Quantitative Research and Trading System," [7] was developed by Jinxiang Lu at the University of Texas at Austin. It provides an execution framework for the analysis of intra-day trading strategies in a way that is flexible and easily extensible, as well as libraries for analysts to use when coding trading strategies. QUARTS architecture contains pluggable modules for entry strategy, exit strategy, position sizing, risk management, and execution.

Many parameters are quickly configurable in QUARTS, which allows them to be optimized easily. The configurable parameters in QUARTS include those around trailing stops (exit strategy), maximum daily loss (risk management), and number of shares to purchase (position sizing, if evaluating a model for a fixed number of shares). QUARTS also gracefully handles real issues that arise in trading, like partial order fulfillment, and incorporates a realistic delay in order fulfillment.

QUARTS contains the following main modules:

- Market Engine Data Simulator that simulates a market data feed and passes data to the strategy level to be evaluated.
- Entry, Exit, Risk Management, Position Sizing, and Execution Strategy modules that provide decision logic about buy, sell, short, and cover actions.
- An Account module that contains information about order history, positions, and cash flow.

- An Order Matching Engine Simulator, which simulates an exchange and facilitates orders.

QUARTS was implemented in Python with the goal of striking a balance between performance and analyst productivity. Each core strategy module provides an interface to make implementing a strategy work seamlessly with the system. Figure 1 shows the QUARTS data flow and Table 1 shows the trade data format used by QUARTS.

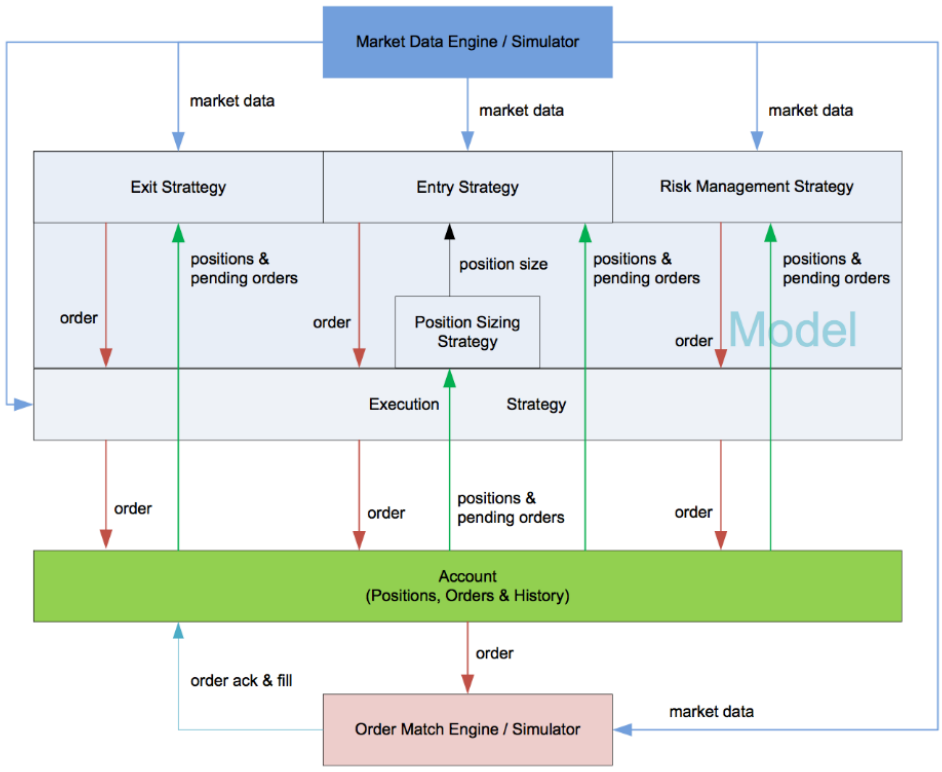

Figure 1: QUARTS Data Flow from [7, p. 25].

Field Name	Description
Date	Date of trade
Time	Time of day of trade
Symbol	Ticker symbol
Price	Price of trade
Volume	Number of shares traded
Participant Id	Code for the exchange on which the trade was made
SaleCondition	Any conditions on the trade, such as if it was an after-hours trade, or part of an aquisition
BestBidSize	Best bid size at the time of the trade
BestBidPrice	Best bid price at the time of the trade
BestAskPrice	Best ask price at the time of the trade
BestAskSize	Best ask size at the time of the trade

Table 1: Trade Data Format

TA-LIB

TA-LIB provides a technical analysis library that includes many common functions [8]. The RSI function, commonly used by technical analysis practitioners, is used in this project. RSI stands for Relative Strength Indicator. The idea behind it is that a sharp trend upwards is likely to result in a correction downward. Likewise, a sharp trend downward will be followed by an upward correction [9].

The RSI is calculated as: $RSI = 100 - (100 / (1 + (AVG_{up}/AVG_{down})))$ where:

- AVG_{up} = Sum of all changes for advancing periods divided by the total number of periods.
- AVG_{down} = Sum of all changes for declining periods divided by the total number of periods.

[9]

DISTRIBUTED COMPUTING

Hadoop

Hadoop has become the most widely recognizable name in distributed computing. Yahoo has been using Hadoop and contributing to the project for several years, beginning with work on their search then applying Hadoop to ad targeting as well as many other uses [10]. Facebook uses it for several purposes including Facebook Messages and Facebook Insights, their analytics tool [11]. Twitter uses Hadoop heavily for a large number of data analysis tasks [12]. Twitter also contributes a great deal of functionality back to Hadoop, and has spun off several projects based on the Hadoop ecosystem. Storm [13], "a distributed realtime computation system," is one example that has been gaining popularity.

Hadoop-based companies are gaining traction at a rapid pace. Now partnering with Teradata, Microsoft, and others, Hortonworks is bringing together big names in industry. In June 2013, Hortonworks announced that they had taken an additional $50M in venture capital [14]. DataTorrent, who provides a streaming solution based on Hadoop, raised $8M in Series A funding in June 2013 [15]. These are only a couple of examples to illustrate the buzz that has been building around Big Data, and Hadoop specifically.

An ecosystem of related open source projects has developed around Hadoop including but not limited to: HBase, a distributed database modeled after Google's

BigTable and currently storing all Facebook Messages data [9]; HIVE, a SQL-like interface for Hadoop; PIG, a framework for Hadoop development using a language called Pig Latin for data manipulation [16]; and Mahout, a machine learning library that is designed to run on Hadoop. PIG and HIVE will be discussed in more detail later in this chapter.

MapReduce

MapReduce is the programming model used to interface with Hadoop. Logically, MapReduce consists of functions specified as mappers and reducers that input and output key-value pairs. Mappers take a key-value pair, perform some operation, and output zero or more key-value pairs. Reducers receive a key and a collection of the values associated with that key, perform some operation, and output zero or more key-value pairs.

Between the map and reduce steps, there is a partitioner, a combiner and a shuffle and sort step. The partitioner determines which reducer receives which key and is optionally customizable. The default partitioner "involves computing the hash value of the key and then taking the mod of that value with the number of reducers. This assigns approximately the same number of keys to each reducer (dependent on the quality of the hash function)" [17]. A developer may want to specify a different partitioner if the values associated with each key are not evenly distributed. The combiner is an optional step where developers can specify pre-reduce operations and is typically used as an optimization. The shuffle and short step aggregates values by key to deliver to the reducer [17]. Analytics processing is a common use case for MapReduce, and Facebook Insights uses it for that purpose [11].

Hadoop Streaming

Hadoop Streaming allows reusability of components written in languages that do not get compiled to Java byte code. A developer can define map and reduce methods in a script or executable and run them on Hadoop. Hadoop Streaming was chosen to use for the initial round of QUARTS performance optimization because it required very little modification of QUARTS to be able to distribute the workload over a Hadoop cluster.

Cascading

The Cascading framework, in the spirit of HIVE and PIG, provides a high level abstraction for developers to use to create data processing and analytics jobs. While HIVE and PIG provide an interface for creating and running jobs, Cascading's focus is to provide a framework for enterprise grade data workflows. Cascading abstracts away the need to write MapReduce methods, and allows developers to focus on data flow.

The concept lends itself well to functional languages, which has resulted in two variants of Cascading, Cascalog and Scalding, which have been gaining popularity and are in use at a variety of companies including Twitter, where they originated. Cascalog is based on Clojure, a functional language that is a dialect of LISP, and Scalding is based on Scala, a language containing both object oriented and functional programming elements.

Framework Comparison

Examples of the ubiquitous WordCount program in MapReduce, HIVE, PIG, and Cascading are shown in Figures 2, 3, 4, and 5, respectively, to demonstrate the usage of each framework. WordCount takes text input and provides an occurrence count of each word as output and is frequently used as an example in Hadoop documentation.

HIVE, PIG, and Cascading all produce MapReduce jobs. The differences are in language and available libraries. HIVE provides a declarative SQL-like interface, while

PIG development is done in Pig Latin, a data manipulation language. Cascading, on the other hand, consists of a set of Java libraries with the intention that faults get caught during compilation rather than during job execution [16].

Cascading was chosen for this project in the hope that it would reduce development time, provide code that is more concise and readable, and thus be less prone to flaws. Cascading also has the advantage that development is done in Java, which eliminates the need to learn another programming language and thus reduces the learning curve.

```
// the mapper class takes a <LongWritable, Text> key-value pair as input
// and outputs a <Text, IntWritable> key-value pair
public static class Map extends MapReduceBase
                        implements
                        Mapper<LongWritable, Text, Text, IntWritable> {

     … // variable setup

     // the map function will be called for each line of text
     public void map(LongWritable key, Text value, OutputCollector<Text,
                     IntWritable> output,
                     Reporter reporter) throws IOException {
          // split the line into words
          StringTokenizer tokenizer = new StringTokenizer(line);

          // for each word, output the word (key), the number one (value)
          // and increment the counter
          while (tokenizer.hasMoreTokens()) {
               word.set(tokenizer.nextToken());
               output.collect(word, one);
               reporter.incrCounter(Counters.INPUT_WORDS, 1);
          }
     }
}

// the reducer takes a <Text, IntWritable> key-value pair as input
// and outputs another <Text, IntWritable> key-value pair
public static class Reduce extends MapReduceBase
                           implements
                           Reducer<Text, IntWritable, Text, IntWritable> {

     // input is each word from the mapper, and the corresponding
     // collection of one values
     public void reduce(Text key, Iterator<IntWritable> values,
                                  OutputCollector<Text, IntWritable> output,
                                  Reporter reporter)
                                  throws IOException {

          // sum up the values and output the word with its count
          int sum = 0;
          while (values.hasNext()) {
               sum += values.next().get();
          }
          output.collect(key, new IntWritable(sum));
     }
}
```

Figure 2: WordCount in MapReduce from [18].

```
-- create the logical table with one column of type STRING
CREATE TABLE input (line STRING);

-- load data from HDFS into table, overwriting the current table
LOAD DATA LOCAL INPATH 'input.tsv'
OVERWRITE INTO TABLE input;

-- from all lines split on delimiter ' ', group by word
-- select the word and count for each group
SELECT
     word, COUNT(*)
FROM input
     LATERAL VIEW explode(split(text, ' ')) lTable AS word
GROUP BY word
;
```

Figure 3: WordCount in HIVE from [19].

```
docPipe = LOAD '$docPath'
     USING PigStorage('\t', 'tagsource') AS (doc_id, text);
docPipe = FILTER docPipe BY doc_id != 'doc_id';

-- specify a regex to split "document" text lines into token stream
tokenPipe = FOREACH docPipe
     GENERATE doc_id, FLATTEN(TOKENIZE(text, ' [](),.')) AS token;
tokenPipe = FILTER tokenPipe BY token MATCHES '\\w.*';

-- determine the word counts
tokenGroups = GROUP tokenPipe BY token;
wcPipe = FOREACH tokenGroups
     GENERATE group AS token, COUNT(tokenPipe) AS count;

-- output
STORE wcPipe INTO '$wcPath'
     USING PigStorage('\t', 'tagsource');
EXPLAIN -out dot/wc_pig.dot -dot wcPipe;
```

Figure 4: WordCount in PIG from [19].

```
String docPath = args[ 0 ];
String wcPath = args[ 1 ];

Properties properties = new Properties();
AppProps.setApplicationJarClass( properties, Main.class );
HadoopFlowConnector flowConnector = new HadoopFlowConnector( properties );

// create source and sink taps
Tap docTap = new Hfs( new TextDelimited( true, "\t" ), docPath );
Tap wcTap = new Hfs( new TextDelimited( true, "\t" ), wcPath );

// specify a regex operation to split the "document" text lines into a token
// stream
Fields token = new Fields( "token" );
Fields text = new Fields( "text" );
RegexSplitGenerator splitter = new RegexSplitGenerator( token,
                                "[\\[\\]\\(\\),.]" );
// only returns "token"
Pipe docPipe = new Each( "token", text, splitter, Fields.RESULTS );

// determine the word counts
Pipe wcPipe = new Pipe( "wc", docPipe );
wcPipe = new GroupBy( wcPipe, token );
wcPipe = new Every( wcPipe, Fields.ALL, new Count(), Fields.ALL );

// connect the taps, pipes, etc., into a flow
FlowDef flowDef = FlowDef.flowDef()
     .setName( "wc" )
     .addSource( docPipe, docTap )
     .addTailSink( wcPipe, wcTap );

// write a DOT file and run the flow
Flow wcFlow = flowConnector.connect( flowDef );
wcFlow.writeDOT( "dot/wc.dot" );
wcFlow.complete();
```

Figure 5: WordCount in Cascading from [19].

Chapter 3: Design and Implementation

IMPROVING PERFORMANCE

The first goal was to enhance QUARTS so that it could evaluate multiple days of trade data more quickly. An incremental approach was used accomplish this. Opportunities for improvement were evaluated and ordered by estimated effort involved, then implemented and tested in an iterative fashion.

Trade Filtering

Frequently, whether analyzing trades made on a particular exchange [20], or after a filtering step [21], only a subset of symbols is to be considered for back testing. Since QUARTS loops through all trades made on a given day, if a large percent of trades could be eliminated because they were made on equities not being considered, QUARTS performance should be improved. Conveniently, filtering text is an excellent use case for Hadoop.

We used Cascading to filter the trades by a specified list of ticker symbols. This filtering is performed by the entire Hadoop cluster very efficiently. First, a join is performed on the trade data set and the list of ticker symbols. Then the null records are excluded from the result. Figure 6 shows the workflow for trade filtering performed by the generated MapReduce jobs.

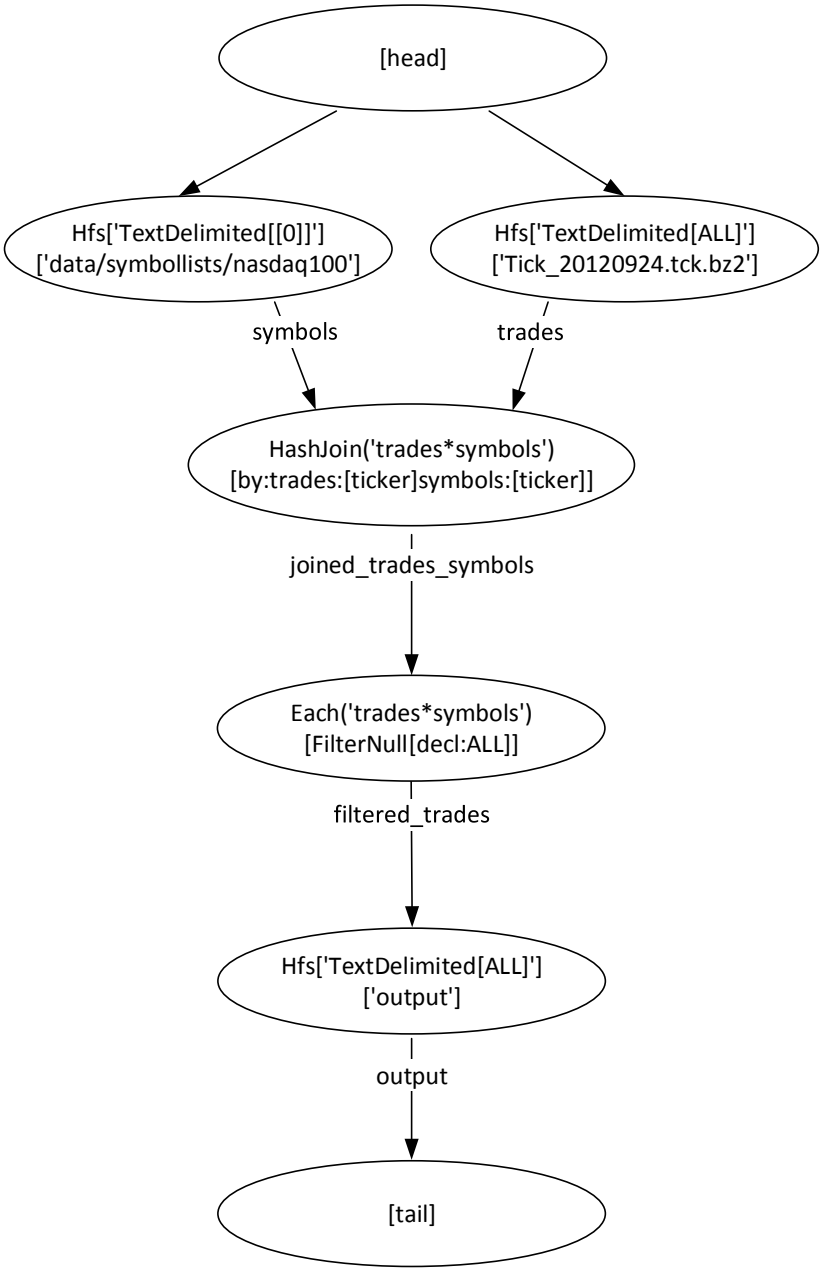

Figure 6: Trade Filtering Workflow.

The Cascading code needed to filter by symbol is very concise. A Memory Backed Join [17], which is named Hash Join in Cascading [16], was chosen because the set of symbols being considered will fit in memory and thus should perform better. The code is shown in Figure 7.

```
// ************ Filter trades flow setup ************
// ticker symbol at position 3
Fields ticker = new Fields(3);
List<Tap> inputs = new ArrayList<Tap>();
inputs.add(new Hfs( new TextDelimited( false, "|" ), inputPath + "/Tick_" +
                    date + ".tck.bz2" ));
MultiSourceTap inputTap = new MultiSourceTap(inputs.toArray(new
                    Tap[inputs.size()]));

// symbols is one column of ticker symbols
Fields symbols = new Fields(0);
Tap symbolsTap = new Hfs( new TextDelimited(symbols, false, "\t" ),
                    symbolsPath );

// lhs is tick data, rhs is symbols to include
// using a hash join [17] [12] because it is faster
// since the symbol list can fit in memory
Pipe tradesPipe = new Pipe("trades");
Pipe symbolsPipe = new Pipe("symbols");
Pipe filteredTradesPipe = new HashJoin(tradesPipe, ticker, symbolsPipe,
                              symbols, new LeftJoin());
filteredTradesPipe = new Each( filteredTradesPipe, new Fields(21), new
                              FilterNull());
```

Figure 7: Trade Filtering Cascading Code.

Distributing the QUARTS Workload

To address the distribution of back testing multiple days of trades, Cascading code was added to use Hadoop Streaming to run the Python application on multiple nodes, with each node processing one day of filtered trade data. A GroupBy was used to group on date and sort by time to prepare trades for each node. The only edits needed to QUARTS were replacing file reads and writes with inputs to stdin and outputs to stdout, respectively. Figure 8 shows the Cascading code used for grouping, sorting and generating Hadoop

Streaming jobs. The output is a list of buy, sell, short, and cover orders that were made during the back test.

```
// Group by date, each reducer ends up getting the data set for that date,
// only including rows for the symbol list specified
Pipe quartInputsPipe = new Pipe("quarts_input", filteredTradesPipe);
Fields date = new Fields(1);
Fields time = new Fields(2);
quartInputsPipe = new GroupBy(quartInputsPipe, date, time);

// set up subflow
FlowDef filterTradesFlowDef = FlowDef.flowDef()
               .setName("trades")
               .addSource(tradesPipe, inputTap)
               .addSource( symbolsPipe, symbolsTap )
               .addTailSink(quartInputsPipe, quartsInputTap);
Flow filterTradesFlow = flowConnector.connect(filterTradesFlowDef);

// set up the streaming job for quarts
String quartsOutputPath = args[1] + "-quarts-output";
JobConf streamConf = StreamJob.createJob(new String[]{
               "-input", quartsInputPath.toString(),
               "-output", quartsOutputPath.toString(),
               // python
               "-mapper", "python '" + pythonModulePath + "/identity.py'",
               "-reducer", "python '" + pythonModulePath + "/runner.py'"
});
```

Figure 8: QUARTS Cascading Hadoop Streaming Code.

Aggregation

Cascading code to aggregate the outcome in three different ways was added to be able to review back testing results. The added aggregation includes number of trades, number of trades per symbol, and total profit and loss (PNL). For the total count, we group by Fields.NONE, indicating no grouping, then count all of the orders made. Similarly, for the total PNL, we group by Fields.NONE, and sum all of the order prices, which include transaction costs. Finally, for count per symbol, we group by symbol, then perform the count. Cascading makes the code to create these MapReduce jobs very concise, as is shown

in Figure 9. Figure 10 shows the workflow demonstrating the process of aggregating the results.

```
Aggregator count = new Count(new Fields("count"));
// one branch counts the number of trades
Pipe cntPipe = new Pipe( "CT", quartsPipe );
cntPipe = new GroupBy(cntPipe, Fields.NONE);
cntPipe = new Every(cntPipe, count);

// one branch tallies the counts per ticker symbol
Fields symbol = new Fields(2);
Pipe cntPerSymbolPipe = new Pipe( "CTPA", quartsPipe);
cntPerSymbolPipe = new GroupBy(cntPerSymbolPipe, symbol);
cntPerSymbolPipe = new Every(cntPerSymbolPipe, count);

// one branch calculates total pnl
Aggregator sum = new Sum(new Fields("sum"));
Fields adjPrice = new Fields(5);
Pipe pnlPipe = new Pipe( "PNL", quartsPipe);
pnlPipe = new GroupBy(pnlPipe, Fields.NONE);
pnlPipe = new Every(pnlPipe, adjPrice, sum);
```

Figure 9: QUARTS Cascading Aggregation.

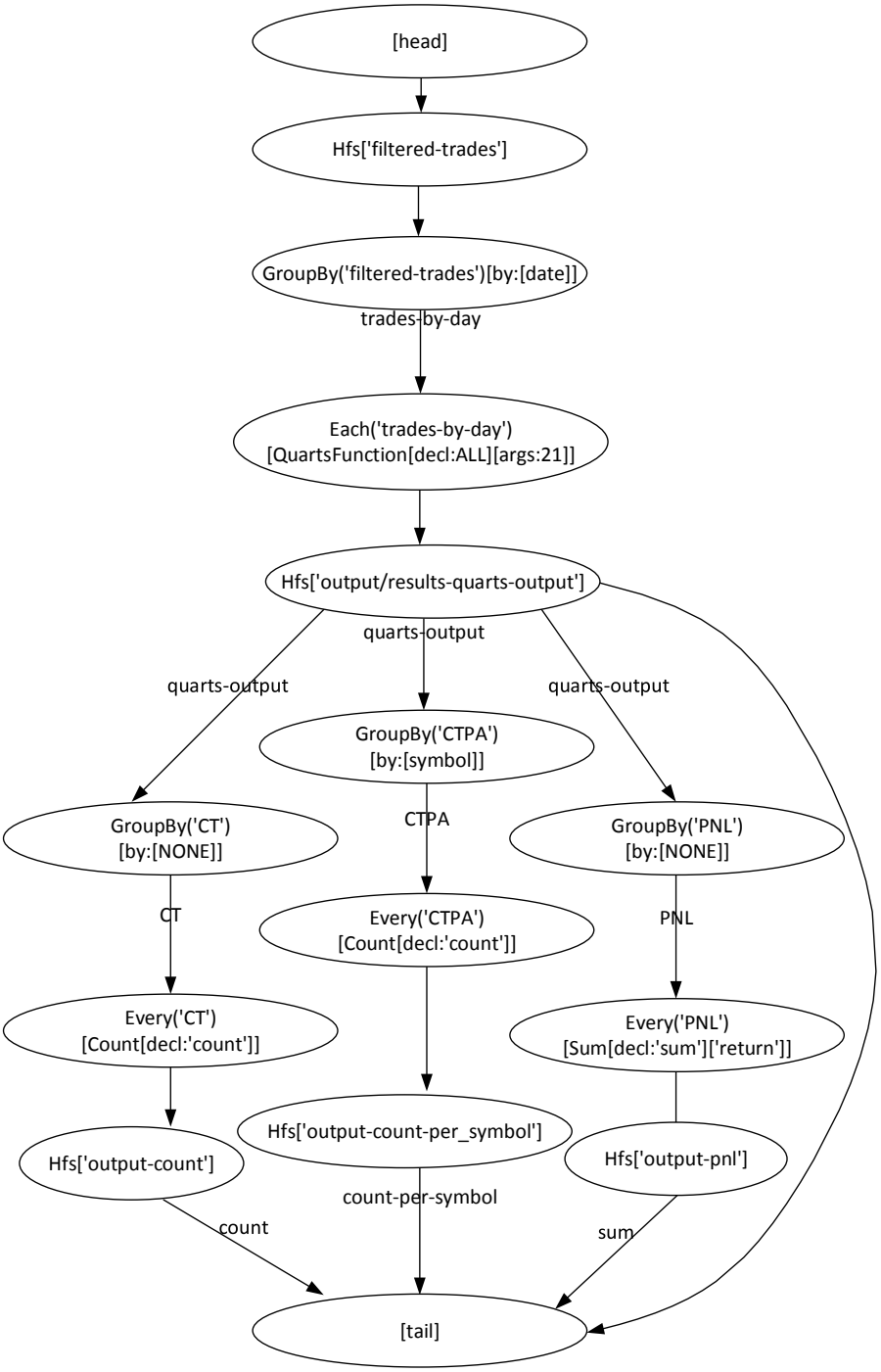

Figure 10: QUARTS Cascading Workflow.

Hadoop, Java versus Streaming

Because the jobs being run are data intensive, the performance of Java is expected to be significantly better than Hadoop Streaming [22]. QUARTS was completely translated from Python to Java with the intention of maximizing performance. Results of tests comparing the two implementations are presented in Chapter 4. This Java version will be referred to as QUARTS Java.

The same filtering and aggregation process used with Hadoop Streaming was used with QUARTS Java. The main piece that needed implementation to tie into Cascading was running the QUARTS Java translation itself.

A feature of Cascading is the ability to define a custom function to be run on each record after a GroupBy. Such a function was added and wired into the Cascading flow, which is shown in Figure 11. Logically, the filtered trades are grouped by date, sorted by time, and then each resulting record is passed to the Quarts Function for analysis. When running on Hadoop, the function is executed as a reduce step, with each reducer getting one day of data, presorted by time of day. The function code is shown in Figure 12.

```
… // instantiation of QUARTS objects

… // Cascading setup

… // pre-filtering

quartsPipe = new GroupBy(quartsPipe, date, time);

// call quarts function
quartsPipe = new Each( quartsPipe, Fields.ALL, new
QuartsFunction(Fields.ALL,
                              marketDataEngineSimulator,
                              orderMatchingEngineSimulator,
                              accounts, true, commissionPerShare, modelName,
                              options, params,
                              commissionFeeModelName, args), Fields.RESULTS);
```

Figure 11: QUARTS Java Cascading Function Call.

```
public void operate( FlowProcess flowProcess, FunctionCall functionCall ) {
     TupleEntry argument = functionCall.getArguments();
     String line = argument.getTuple().toString("|", false)
     …
     List<TradeWritable> trades =
marketDataEngineSimulator.processLine(line,
                              orderMatchingEngineSimulator, accounts,
                              isMarketOnlyHours);
     if (trades != null) {
          for (TradeWritable trade : trades) {
               BigDecimal tradePrice = trade.getPrice().multiply(
                                        new BigDecimal(trade.getShares()));
               BigDecimal tradeCommission = commissionPerShare.multiply(
                                        new BigDecimal(trade.getShares()));
               Tuple result = new Tuple();
               result.add(trade.getOrderType().toString());
               result.add(trade.getTradeTime().toString());
               result.add(trade.getSymbol());
               result.add(trade.getShares());
               result.add(trade.getPrice().toString());
               result.add(tradePrice.toString());
               result.add(tradePrice.subtract(tradeCommission).toString());
               result.add(tradeCommission.toString());
               functionCall.getOutputCollector().add( result );
          }
     }
}
```

Figure 12: QUARTS Java Cascading Function.

PARAMETER TUNING

The goal of parameter tuning, also referred to as parameter optimization, is to find the optimal parameters for a trading strategy. Examples of parameters that might be tuned are: number of standard deviations outside of past price movements to trigger a buy or sell action, and number of periods to look back on to consider for those past price fluctuations. Parameter tuning is, by nature, a computationally expensive process, since the trading strategy needs to be run through the entire trade data set for each parameter combination.

The RSI (Relative Strength Index) indicator was chosen for experimentation in tuning parameters because technical analysts commonly use it. As discussed in Chapter 2, RSI is used as an indicator to generate buy and sell signals. When RSI is less than some threshold, a security is considered oversold, which is interpreted as a signal to buy (or cover). When RSI is above a threshold, a security is considered overbought, which is interpreted as a signal to sell (or short).

For simplicity, we considered RSI as an entry strategy in a long-only account, which provided two parameters to optimize: number of advancing periods, and oversold threshold to use as a buy signal. This part of the RSI entry strategy is expressed in code in Figure 13.

```
Core core = new Core();
double[] rsiOut = new double[prices.size()];
MInteger rsiBegin = new MInteger();
MInteger rsiLength = new MInteger();

core.rsi(0, prices.size() - 1, pricesArray, periods, rsiBegin, rsiLength,
         rsiOut);
prices.poll();
int shares = model.getPositionSizingStrategy().evaluate(marketData,
             account).intValue();
if (shares == 0) {
     return false;
}

if (options.isLong() && rsiOut[rsiLength.value - 1] != 0.0 &&
               rsiOut[rsiLength.value - 1] < overSoldThreshold) {
     LOG.info(marketData.getTime() + ": rsi entry to buy triggered.");
     account.buy(marketData.getSymbol(), shares, marketData.getPrice(),
null);
     return true;
}
…
```

Figure 13: RSI Entry Strategy.

Brute Force

The most obvious starting point for tuning parameters in a trading strategy is brute force. While not an ideal solution because of the combinatorial explosion that occurs, it served to provide a reference for improvements.

The brute force implementation of parameter tuning uses the same workflow as previously described, except that after the filtering by symbol, jobs are launched in a loop that iterates over each combination of parameters. This creates a large number of Hadoop jobs and necessitated the addition of a step to limit the jobs launched at one time to prevent the name node from running out of memory. After some trial and error, we found that 100 combinations could be very reliably processed at a time, including evaluation and aggregation.

Hill Climbing and Genetic Algorithms

Hill Climbing algorithms are a class of algorithms that focus on optimizing a parameter combination. As the name indicates, they can be visualized as starting at some point on a surface and trying to get to the top in some number of steps with step distance delta. The simplest version is to take a step upward in the steepest direction, take another step upward in the steepest direction, and repeat until at the top-most point [23]. Gradient Descent is one flavor of Hill Climbing commonly used in regression algorithms to minimize a cost function.

One issue is that this simple sort of Hill Climbing does not take into account any local maxima that may occur on the surface. If there are several local maxima and delta is not chosen accurately, then it is likely that one of them will be the stopping point. Adding a mutation step makes it less likely to stop at a local maximum, since more of the surface is potentially traversed. Pseudocode for the algorithm used is shown in Figure 14. The number of candidates per generation is represented by *m*, and *n* is number of generations the algorithm will continue to run with no fitness improvement. In genetic algorithm terminology, our termination criterion is *n* generations passing with no improvement in fitness.

```
X0 <- randomly generated candidate
m <- number of candidates per generation per feature
n <- number of generations with no improvement to terminate after
countToTermination <- 0
while countToTermination <- n
     add X0 to collection of potential candidates
     for each feature
          for 0 to m
               Xq <- X0 with feature mutated
               add Xq to collection of potential candidates
          end for
     end for each
     X' <- candidate with maximum fitness in potential candidates
     if X0 = X'
          n++
     else
          n <- 0
          X0 <- Xq
     end if
end while
```

Figure 14: Hill Climbing with Mutations per Iteration.

We have the ability to get the fitness for each candidate in parallel. Also, finding candidate fitness is the most computationally expensive part of the process. Therefore we want to make *m* large enough to reduce the total number of generations required, but not too large. If the number of generations is 1, and *m* is the number of possible combinations, this is equivalent to the brute force method. For the tests presented in Chapter 4, *m* was set to 6.

Chapter 4: Experimental Results

DISTRIBUTING QUARTS WORKLOAD

Round One

The first round of experiments were focused on running QUARTS over multiple days of trade data, with the hope that the time to complete each run would be nearly linearly scalable by adding nodes as needed. These were run first on a MacBook Pro with a 2.6 GHz Intel i7 and 8 GB of RAM. The original QUARTS application was used as a baseline for comparison.

The QUARTS configuration that was run is as follows.

- Long Only: Only long positions are considered.
- Random Walk Entry Strategy: This entry strategy considers whether to make an entry randomly for each security. If there is already a position in a given security, no additional entry will be made. Entries will be made only between 10:00 am and 10:01 am.
- Profit, Trailing Stop, Timed Exit Strategy: If three percent profit is made, if a position is down one percent, or if the time is greater than or equal to 3:55 pm, exit.
- Maximum Daily Loss Risk Management Strategy: If the loss for the day hits $3000, exit all positions.
- NASDAQ 100: Only securities in the NASDAQ 100 are considered.

The results of running this configuration using the original QUARTS application and Hadoop Streaming, both on the MacBook Pro are shown in Table 2. The

improvements in time to complete a run on a single day of trade data shown are a result of filtering out trades made on securities not in the NASDAQ 100.

QUARTS	Hadoop Streaming
13 min., 45 sec.	7 min., 24 sec.

Table 2: Run Times over a Single Day on a Single Machine, QUARTS and QUARTS on Hadoop Streaming.

The next comparison is between Hadoop Streaming and QUARTS Java on Hadoop, using the same configuration and on the same machine. The improvement in time to complete the run is due to the speed of Python versus Java. Results are shown in Table 3.

Hadoop Streaming	QUARTS Java on Hadoop
7 min., 24 sec.	5 min., 56 sec.

Table 3: Run Times over a Single Day on a Single Machine, Hadoop Streaming and QUARTS Java on Hadoop.

Scalability was then tested on HDInsight, a managed Hadoop service offered by Microsoft on the Azure platform. Large virtual machine instances were used, which at the time of the tests were configured with 4 CPU cores and 7GB of RAM. Each test was run five times to verify that the process was repeatable.

Runs were then made on different sizes of Hadoop clusters running on HDInsight. First, runs were made on a four-node cluster of four days of trade data. The dates used were from 9/26/2012 to 10/1/2012, inclusive. After that, runs were made on an eight node cluster and 32 node cluster on a number of days corresponding to cluster size, all using the same start date of 9/26/2012. Results are shown in Table 4.

4 Days, 4 Nodes, HDInsight	8 Days, 8 Nodes, HDInsight	32 Days, 32 Nodes, HDInsight
13 min., 33 sec.	12 min., 49 sec.	13 min., 54 sec

Table 4: Run Times with Different Configurations, HDInsight.

The Hadoop Streaming job could not be run on HDInsight since Python is not installed on the cluster by default, and there was no discoverable documentation around Python installation. HDInsight is a new service and currently offered as a preview, so the documentation is not yet complete. The next section discusses results of QUARTS successfully run on Hadoop Streaming using a different commercial offering.

Round Two

The second round of tests was performed using Amazon's Elastic MapReduce (EMR). Amazon offers several size options for virtual machines, called Elastic Compute Cloud (EC2) instances, which make up EMR clusters. Tests were run on EC2 m1.xlarge instances, which fall into the AWS general-purpose category. This instance size, with 4 cores and 15 GB RAM, were the closest machines to the specifications of the MacBook Pro and HDInsight instances that AWS offers, while falling into their high network performance category.

With Elastic MapReduce, it was possible to test the Hadoop Streaming version. With EMR, software can be installed on cluster nodes by specifying a bootstrap shell script. Python 2.7 was loaded and the QUARTS python source was installed locally on each node using the script shown in Figure 15. Results of QUARTS running on Hadoop Streaming are shown in Table 5. The Hadoop Streaming implementation proved to be linearly scalable by adding nodes.

```
#!/bin/bash
wget http://python.org/ftp/python/2.7.2/Python-2.7.2.tar.bz2
tar jfx Python-2.7.2.tar.bz2
cd Python-2.7.2
./configure --with-threads --enable-shared
make
sudo make install
sudo ln -s /usr/local/lib/libpython2.7.so.1.0 /usr/lib/
sudo ln -s /usr/local/lib/libpython2.7.so /usr/
wget http://quartspy.s3.amazonaws.com/quarts_py.tar.gz
mkdir -p /home/hadoop/contents
tar -xzf quarts_py.tar.gz -C /home/hadoop/contents
```

Figure 15: EMR Hadoop Streaming Python 2.7 bootstrapper.

2 Days, 2 Nodes, EMR	4 Days, 4 Nodes, EMR	8 Days, 8 Nodes, EMR
14 min., 32 sec.	14 min., 56 sec.	14 min., 9 sec.

Table 5: Hadoop Streaming Run Times, EMR.

QUARTS Java performed roughly the same on EMR as HDInsight, averaging 13 minutes 3 and seconds on EMR compared to the 13 minutes and 6 seconds average on HDInsight.

PARAMETER OPTIMIZATION

Data Exploration

A brute force run was made on Elastic MapReduce to use as a baseline for comparison. The QUARTS setup was the same as above, the only difference being that instead of a random entry strategy, RSI was used to determine entry. Periods ranging from 10 to 20, and an oversold threshold from 20 to 40 were analyzed on trades from September 24, 2012. The run completed on a nine-node cluster in 3 hours and 50 minutes, successfully running 231 parameter combinations.

While the returns for these parameter combinations, shown in Figure 16, were dismal, the results were promising for using a Hill Climbing algorithm. The returns do include trading costs. For the given day and parameter ranges, the resulting surface was a fairly smooth plane. There were no significant local maxima.

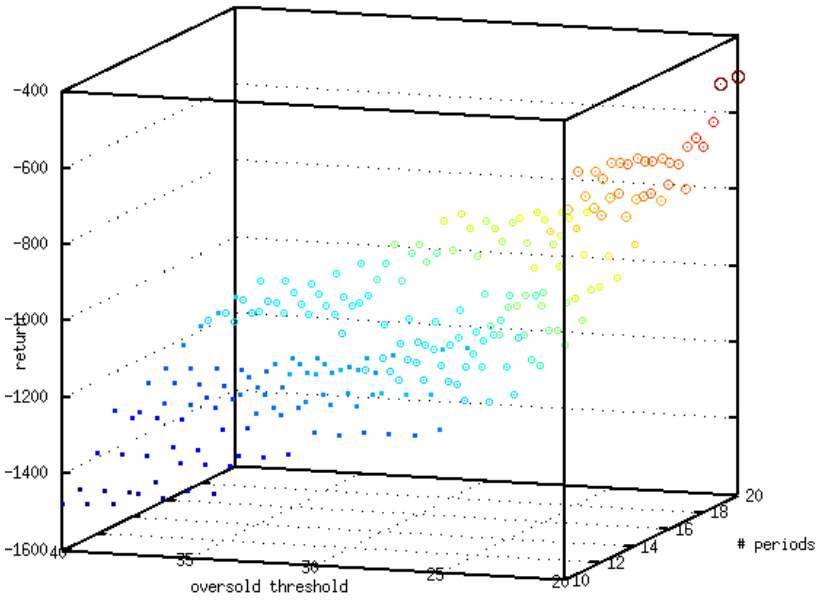

Figure 16: RSI Parameter Combination Results for September 24, 2012

Performance

The Hill Climbing Algorithm was then run to see how the complexity could be reduced from the brute force method. It should be noted that the focus with this run was on optimization of time taken to converge on a reasonable parameter combination. In practice, tuning on one day of trade data would likely result in overfitting the resulting model. The following section of this Chapter will provide a more realistic example of tuning and validation. The Hill Climbing method frequently found a slightly suboptimal parameter combination, which could potentially result in less overfitting, but tuning on a

larger time period and then testing that model on a significant number of days would be recommended.

Table 6 shows the results of running the Hill Climbing approach on the same day and the same parameter range. The number of candidates per generation was set to 6. With an average of 9 generations, this resulted in an average of 54 combinations being evaluated versus 231 using the brute force method.

	Run 1	Run 2	Run 3	Run 4	Run 5
Number of Generations	8	10	9	8	10
Best Candidate Period	19	20	19	20	20
Best Candidate Oversold Threshold	20	20	20	20	20
Best Candidate PNL	-485.90	-485.41	-485.90	-485.41	-485.41

Table 6: Hill Climbing Algorithm Results, 6 Candidates per Generation.

Strategy Testing

Finally, we ran the Hill Climbing Algorithm on the RSI Strategy as a real test of the strategy. We extended the parameter range used above since it seemed that for the data exploration run above, the optimal parameters may have not been included. Advancing periods from 20 to 50 and oversold thresholds from 5 to 40 were considered in the evaluation. We used a training set of 5 days of trade data for tuning, then ran the optimized parameters the following 5 days for validation. The dates used were from 9/27/2013 to 10/3/2013 for tuning then 10/4/2013 to 10/10/2013 for testing. The results of the tuning were promising, showing a positive sign and an identifiable slope, and are shown in Figure

17. The winning parameter combination was 21 advancing periods and an oversold threshold of 38, and was found in 10 generations. This combination had a return of $859.

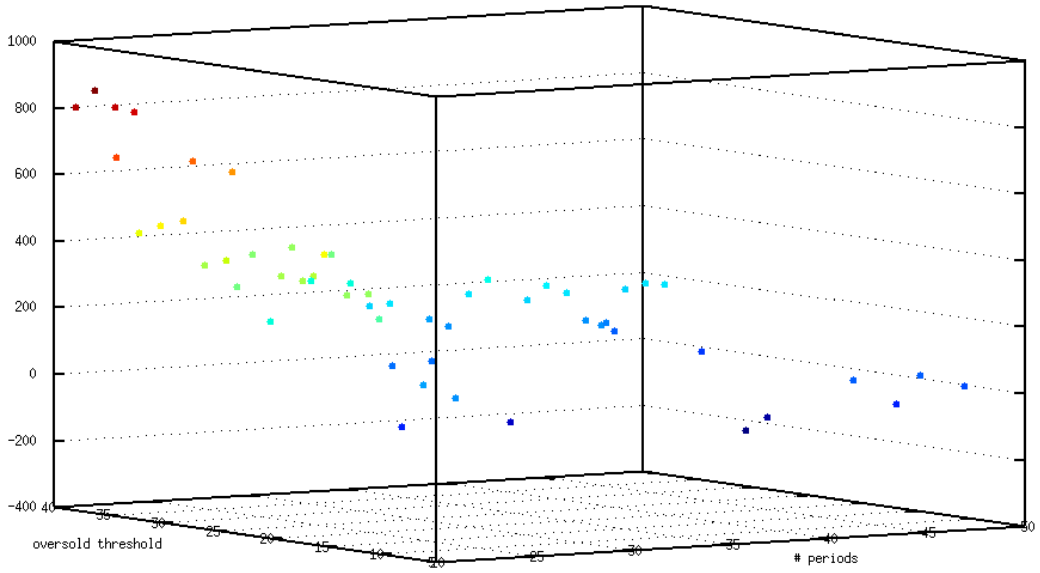

Figure 17: Hill Climbing Results, RSI Strategy, 9/27/2013 to 10/3/2013

The test run on the following week with the winning parameters told a different story, however. When the same strategy was run on the date range 10/4/2013 to 10/10/2013, the return was $-731.68. It seems that the strategy needs more work, but at least now we have tools to test that work.

Chapter 5: Conclusions

DISTRIBUTING QUARTS

The goal of distributing the back testing workload in a way that is linearly scalable was met. Both QUARTS using Hadoop Streaming and QUARTS Java scaled linearly by adding an additional worker node to the cluster for each day of data to be analyzed.

Framework Selection

QUARTS Java on Hadoop was approximately 10% faster than Hadoop Streaming. While not insignificant, this was less performance improvement than anticipated. Considering the amount of work involved in converting QUARTS to Java, it is questionable whether the conversion was a worthwhile endeavor.

Run time comparisons of QUARTS using Hadoop Streaming, and QUARTS Java on Hadoop on a single machine versus a cluster indicate that most of the execution time is spent moving data over the network. If that is the case, it makes sense that for this application there is not a huge performance gain to be expected by running Java over Python.

PARAMETER OPTIMIZATION

The Hill Climbing approach for parameter tuning proved to reduce the amount of work significantly as well. With an average of nine generations at six candidates per generation, the number of jobs was reduced by 75%. Not much accuracy was sacrificed using this approach over brute force. The actual optimal parameter combination was found three times and the second best was found two times, out of five total runs.

OBSERVATIONS AND LESSONS LEARNED

My experience with Hadoop going into this project was comprised of a series of exploratory projects. All of the coding I had done was using MapReduce and had only used PIG a handful of times for quick data exploration.

Framework Selection

In previous work I found that the MapReduce code I had written contained a lot of repeated code dealing with job creation and configuration, and that it was difficult to debug in general. Higher-level abstractions on top of MapReduce, Cascading in particular, seem to make code more concise, shorten development time, and reduce the number of bugs introduced.

I found that having done some MapReduce coding was useful prior to working with Cascading, since I had a general idea how the jobs would be constructed. For example, when changing a Hadoop configuration setting dealing with reduce tasks, it is useful for the developer to know what will be executed as a reduce task.

Learning Approach

The learning approach I took was to start working first, then fill any knowledge gaps as I ran into issues. Through personal observation it seems that this works fairly well when learning a new technology since documentation tends to be dense and difficult to understand with no context. However, I think it would be advisable to do some upfront study on Hadoop architecture and job execution. When running jobs on a cluster, debugging consists of analyzing log files created by different nodes running different tasks. Haphazardly reading log files is not likely to be productive.

General Software Engineering

The difference in speed between Java and Python is itself significant. However, rewriting a system completely for a 10% improvement might not be worthwhile time investment. Finding a way to do some upfront tests to validate the improvement assumptions before doing the bulk of the work would have been good time investment to make.

Cloud Service Provider Selection

It is difficult to make a true comparison between Elastic MapReduce, a mature product, and HDInsight, which was a product preview at the time. The approaches taken by EMR and HDInsight are subtly different in that EMR is more of a fully managed service. During job setup, the user specifies job and configuration details and from that point it is more like a black box. HDInsight provides a cluster with Hadoop installed, configured, and running. The user connects to the name node and runs jobs.

The only real frustration with HDInsight was that while a user can connect remotely to the name node with administrative privileges, there is no clear way to connect to the worker nodes. The ability to connect to the worker nodes to perform server level configuration tasks would fix this issue, or if it is possible to connect currently, to provide documentation about how to connect. EMR has the advantage in this area in that while more is hidden from the user, the user has the ability to perform configuration level tasks.

SUGGESTIONS FOR FUTURE WORK

- Cross validation on parameter optimization to prevent overfitting.
- Other, more advanced, machine learning techniques for optimizing parameters.
- Exploring other technical analysis indicators and a wider range of parameter combinations to determine if any would be useful in practice.
- Exploring the optimization of QUARTS parameters, such as trading start and stop times, and trailing stop settings.

Bibliography

[1] Jr William A. Darity, Ed., *Efficient Market Hypothesis*, 2nd ed. Detroit, USA: Macmillan Reference USA, 2008, vol. 2.

[2] E. F. Fama, "The Behavior of Stock-Market Prices," The Journal of Business, vol. 38, no. 1, pp. 34-105, Jan., 1965.

[3] E. F. Fama, "Efficient Capital Markets: A Review of Theory and Empirical Work," in *Papers and Proc. 28th Annu. Meeting American Finance Association*, New York, NY, year, pp. 383-417

[4] J. D. Schwager, *Hedge Fund Market Wizards: How Winning Traders Win*.: John Wiley & Sons, 2012.

[5] E. Berlekamp. Financial Engineering. [Online]. http://math.berkeley.edu/~berlek/fineng.html

[6] M. Robbins. "Still making a killing; Parts of the hedge fund sector may have suffered during the credit crunch, but it is business as usual for the industry's big names, who earned staggering sums again last year. Mathieu Robbins reports." The Independent (London) (March 26, 2009), p.44.

[7] J. Lu, QUARTS: A Quantitative Research and Trading System , 2013, Univ. Texas.

[8] TicTacTec LLC. ta-lib.org. [Online]. http://ta-lib.org

[9] G. Meissner, "The RSI Revisited," *Futures: News, Analysis & Strategies For Futures, Options & Derivatives Traders*, vol. 30, no. 10, 2011.

[10] "Yahoo's transforms data mining with open-source Hadoop," *The Australian*, November 2008.

[11] D. Borthakur et al., "Apache hadoop goes realtime at Facebook," in *In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11)*, New York, 2011, pp. 1071-1080.

[12] Twitter. (2012, June) twitter.com. [Online]. https://blog.twitter.com/2012/twitter-hadoop-summit

[13] storm-project.net. [Online]. http://storm-project.net/

[14] R. Bearden. (2013, June) hortonworks.com. [Online]. http://hortonworks.com/blog/hortonworks-announces-50-million-in-new-financing-and-welcomes-new-investors-to-accelerate-next-phase-of-growth/

[15] J. Novet. (213, June) GIGAOM. [Online]. http://gigaom.com/2013/06/26/hadoop-hype-spawns-8m-for-datatorrent-product-opportunities-for-splunk-teradata/

[16] P. Nathan, *Enterprise Data Workflows with Cascading*.: O'Reilly Media, Inc., 2013.

[17] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*.: Morgan & Claypool, 2010.

[18] hadoop.apache.org. [Online]. https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html#Example%3A+WordCount+v2.0

[19] Concurrent. (2012, July) github. [Online]. https://github.com/Cascading/Impatient/tree/master/part2

[20] V. Alexeev and F. Tapon, "Testing weak form efficiency on the Toronto Stock Exchange," *Journal of Empirical Finance*, vol. 18, no. 4, pp. 661-691, September 2011.

[21] D. Paraschiv, S. Raghavendra, and L. Vasiliu, "Stocks scanner evaluator for stocks or options," in *IEEE Symposium on Computational Intelligence for Financial Engineering 2009*, 2009, pp. 28, 35.

[22] M. Ding et al, "More convenient more overhead: the performance evaluation of Hadoop streaming," in *In Proceedings of the 2011 ACM Symposium on Research in Applied Computation (RACS '11)*, New York, 2011, pp. 307-313.

[23] D. Simon, *Evolutionary Optimization Algorithms*.: Hoboken: Wiley, 2013.