Comparing Performance of Parquet and JSON Files for a Particle Swarm Optimization Algorithm on Apache Spark

A project submitted in partial fulfillment of the requirements
For the degree of Master of Science in Computer Science

By
Gavin So

May 2016
The project of Gavin So is approved by:

________________________________________
Dr. Jeff Wiegley  
Date

________________________________________
Dr. John Noga  
Date

________________________________________
Dr. Adam Kaplan, Chair  
Date
# TABLE OF CONTENTS

Copyright Page......................................................................................................................... ii
Signature Page.......................................................................................................................... iii
List of Figures ............................................................................................................................ vi
Abstract ....................................................................................................................................... vii

1. Introduction .......................................................................................................................... 1
2. Background .......................................................................................................................... 2
   2.1 Computer Cluster ............................................................................................................. 3
   2.2 Distributed Computing ................................................................................................... 3
   2.3 MapReduce Paradigm ....................................................................................................... 5
   2.4 Apache Hadoop Cluster Computing Framework ............................................................. 6
   2.5 Apache Spark .................................................................................................................. 7
   2.6 Spark SQL File Formats ................................................................................................. 8
   2.7 Particle Swarm Optimization .......................................................................................... 9
   2.8 Trading Strategies ........................................................................................................ 11
3. JSON and Parquet File Formats ......................................................................................... 14
   3.1 Row Oriented vs. Column Oriented Datasets ................................................................. 14
   3.2 JSON ............................................................................................................................ 15
   3.3 Parquet .......................................................................................................................... 17
4. Adaptive Technical Trading Algorithm ........................................................................... 18
   4.1 Introduction .................................................................................................................... 18
   4.2 Component Rules .......................................................................................................... 18
   4.3 Generating a Signal ....................................................................................................... 20
   4.4 Adjusting Weights ......................................................................................................... 21
   4.5 Calculating Component Rule Profits ............................................................................. 25
   4.6 Algorithm Parameters .................................................................................................. 26
   4.7 ATT Algorithm Overview ............................................................................................. 28
5. ATT Algorithm Datasets ..................................................................................................... 32
   5.1 Improving ATT Efficiency ............................................................................................. 32
   5.2 Dataset Representation ................................................................................................. 32
   5.3 ATT Dataset Applicability ............................................................................................ 34
6. Particle Swarm Optimization .............................................................................................. 36
   6.1 Introduction .................................................................................................................... 36
   6.2 Particle Velocity ........................................................................................................... 36
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Distributing Computing Issues</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>The MapReduce Process</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Structure of a Particle</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>Dataset Example</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>Generation of Component Rules</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>Component Rule Signal Generation Process</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Adjusting Weights Flow Chart</td>
<td>24</td>
</tr>
<tr>
<td>8</td>
<td>Table of ATT Parameters</td>
<td>26</td>
</tr>
<tr>
<td>9</td>
<td>Iterative Flow Map of ATT Algorithm</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>Iterative Flow Map of the PSO Algorithm</td>
<td>39</td>
</tr>
<tr>
<td>11</td>
<td>Table of AWS Instances</td>
<td>41</td>
</tr>
<tr>
<td>12</td>
<td>Cluster Configurations</td>
<td>41</td>
</tr>
<tr>
<td>13</td>
<td>Successful Cluster Initialization</td>
<td>44</td>
</tr>
<tr>
<td>14</td>
<td>Successful Connection to Node</td>
<td>45</td>
</tr>
<tr>
<td>15</td>
<td>Execution Times of Various Test Cases</td>
<td>48</td>
</tr>
<tr>
<td>16</td>
<td>Average Free Memory Reading</td>
<td>49</td>
</tr>
<tr>
<td>17</td>
<td>Number of “Get Block Location” Operations</td>
<td>50</td>
</tr>
<tr>
<td>18</td>
<td>Amount of Data Retrieved From Each Slave</td>
<td>51</td>
</tr>
<tr>
<td>19</td>
<td>Average JVM Garbage Collection Reading</td>
<td>52</td>
</tr>
<tr>
<td>20</td>
<td>Top Performing Component Rules “Basic Materials”</td>
<td>57</td>
</tr>
<tr>
<td>21</td>
<td>Top Performing Component Rules “Retail-Wholesale”</td>
<td>58</td>
</tr>
<tr>
<td>22</td>
<td>Top Fitting Particles “Basic Materials”</td>
<td>59</td>
</tr>
<tr>
<td>23</td>
<td>Top Fitting Particles “Retail-Wholesale”</td>
<td>59</td>
</tr>
<tr>
<td>24</td>
<td>Moving Average</td>
<td>63</td>
</tr>
<tr>
<td>25</td>
<td>Trading Range Breakout</td>
<td>64</td>
</tr>
<tr>
<td>26</td>
<td>Bollinger Bands</td>
<td>65</td>
</tr>
<tr>
<td>27</td>
<td>Relative Strength Index</td>
<td>66</td>
</tr>
<tr>
<td>28</td>
<td>Stochastic Oscillator</td>
<td>67</td>
</tr>
<tr>
<td>29</td>
<td>Moving Average Convergence/Divergence</td>
<td>68</td>
</tr>
<tr>
<td>30</td>
<td>On Balance Volume</td>
<td>69</td>
</tr>
<tr>
<td>31</td>
<td>“Basic Materials” Sector and Change</td>
<td>70</td>
</tr>
<tr>
<td>32</td>
<td>“Retail-Wholesale” Sector and Change</td>
<td>71</td>
</tr>
</tbody>
</table>
ABSTRACT

Comparing Performance of Parquet and JSON Files for a Particle Swarm Optimization Algorithm on Apache Spark

By

Gavin So

Master of Science in Computer Science

Cluster computing frameworks have become an increasingly hot technology with the rise of “Big Data”. Apache Spark is an open source cluster computing framework that is particularly well noted for its ability to exercise fast SQL queries using its Spark SQL module. The Spark SQL module primarily supports two file formats: JSON and Parquet.

Algorithmic stock trading is a security analysis mechanism incorporating the use of computer algorithms to dictate the time, quantity, and direction of operation (buy/hold/sell) of stocks. The profitability of a stock trading algorithm may be reliant on a large number of parameters that need to be properly set. One particular instance of how parameter optimization may be achieved is through building a neural network with the Particle Swarm Optimization learning algorithm.

In this paper, parameters of a stock trading algorithm are permuted with the assistance of the Particle Swarm Optimization algorithm. Frequent SQL queries that occupy most of the Particle Swarm Optimization algorithm execution time are used to compare the performance of the JSON and Parquet file formats under different cluster system specifications. The result is that JSON was found to fare better memory abundant cluster configurations while Parquet was found to better under low memory configurations.
1. INTRODUCTION

Apache Spark is an up-and-coming computer cluster platform that has excellent support for SQL queries. Cluster computing is used to describe a system architecture where multiple computers cooperate to solve a bigger problem. Spark’s support for features such as iterative programming and caching as well as an easy to use API makes it an ideal platform to implement complicated machine learning problems such as the Particle Swarm Optimization algorithm.

Particle Swarm Optimization is a metaheuristic machine learning algorithm inspired by the social behavior of a flock of birds. What it means to be a machine learning algorithm is that the Particle Swarm Optimization helps determine the best parameters for another algorithm. A number of “particles”, that are actually sets of solutions, are guided semi-randomly around a search space to offer potential solutions to the optimization problem. As time goes by, the particles in a group converge like a flock of birds toward good local solutions. Upon completion, the particle with the best fit discovered is considered the solution for the Particle Swarm Optimization algorithm. The algorithm the Particle Swarm Optimization permutes in this paper is a stock trading algorithm that incorporates a number of technical trading rules.

The process of permuting parameters of the particular stock trading algorithm in this paper happens to use many SQL queries. Thus, it is a good opportunity to test out the different dataset types supported by Spark SQL. The two dataset types currently supported are Parquet and JSON. Parquet is a column-oriented dataset inspired by the Google Dremel [1] papers. It is revered for its ability to encode large, complicated nested data structures in a column-oriented format. JSON is a row-oriented dataset using the popular web services data-interchange format. JSON is noted for its easiness to understand and debug as well as its flexible schema. Both JSON and Parquet have characteristics that match and do not match well with the use case.

The execution times as well as other metrics are gathered for JSON and Parquet versions of the Particle Swarm Optimization program on 3 different AWS clusters. The clusters have varying specifications with each cluster doubling the amount of RAM as the previous. The findings of the experiment are discussed in the end.
2. BACKGROUND

2.1 Computer Cluster

A computer cluster is a term used to describe the system architecture where a network of computers cooperate to act as a single system. Each computer in a computer cluster is referred to as a “node”. Nodes communicate and share data through means such as a cluster file system or message passing. The goal of the computer cluster is to improve computation speed and performance compared to that of a single computer.

Popularity of such systems emerged from their ability to use inexpensive, off-the-shelf computers, such as personal computers, as part of the network as well as their ability to solve some very large problems that may be impractical to solve on an individual computer. Cluster computing was also made possible by technological advancements such as the reduced cost of processors, the innovation of high speed networks, and the introduction of software intended for distributed computing.

The computer cluster is distinguished from architectures such as grid computing in that it uses a centralized management system [2]. Cluster management is an area of focus when implementing a computer cluster because the high number of nodes has potential to sharply increase administrative costs compared to other architectures such as those with shared memory in some circumstances. The management software is responsible for functions such as task scheduling and the handling of node failures.

Aside from the primary reasons for deploying a computer cluster, high performance and cost efficiency, computer clusters have a number of other advantages. Computer clusters are fault tolerant, meaning that the system could continue to operate despite the failure of one of its nodes. Computer clusters are also highly scalable [3]. The process of adding new nodes to the cluster is simple and this is especially well suited for problems with potential to increase in size. The fault tolerance of computer clusters may also limit the frequency of maintenance in particular circumstances. Centralized management of nodes may also be viewed as a benefit of using a cluster.
2.2 Distributed Computing

<table>
<thead>
<tr>
<th>Issues</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Scheduling</td>
<td>High Performance</td>
<td>Limited Applicability</td>
</tr>
<tr>
<td>Component Failure</td>
<td>Fault Tolerance</td>
<td>Limited System View (of Nodes)</td>
</tr>
<tr>
<td>Communication Link Failure</td>
<td>High Scalability</td>
<td>Communication Costs</td>
</tr>
<tr>
<td>Communication Means</td>
<td>Low Cost</td>
<td>Asynchronous Nature</td>
</tr>
<tr>
<td>Program/Algorithm Implementation</td>
<td>Centralized Management</td>
<td>High Maintenance*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low Maintenance*</td>
</tr>
</tbody>
</table>

Figure 1. The issues that must be considered when implementing a distributed computing system on a computer cluster architecture. Also, the advantages and disadvantages of this system. *May be high or low maintenance depending on circumstance.

Distributed computing is a discipline of computer science concerning the execution of computer programs on a network of components, originally intended to be a network of computers. Distributed programs are divided into many subtasks which are then assigned to be solved by a group of one or more components in the network. The groups of networked components communicate and coordinate actions to share the results of their subtasks and achieve a common goal of solving a larger problem presented by the program [4]. Distributed computing is best suited to solve programs or problems that have comparably sized tasks, have a number of tasks that scale with the problem size, and avoid redundant computations and accesses to data when parallelized.

Definitions of “distributed computing” have come to expand beyond that of the components referring to a network of computers. Distributed computing has even refer to processes that run on the same computer. While there is no strict consensus for the definition of the components used in “distributed computing”, components in general refer to autonomous computational entities with their own local memory that communicate with one another [5]. The communication is usually achieved with a process known as “message passing”, though there exist alternatives such as “message queues”.

There are a number of issues that need to be handled when using distributed computing. Primary among them is the concurrent, or simultaneous, running of components while lacking a global clock to help coordinate their processing. This issue of how to handle two concurrently running computers with different internal clocks is commonly known as the “clock synchronization” problem. Two other common issues that must be handled are the failure of independent components and communication links.

A number of characteristics of distributed computing should be considered while implementing a distributed algorithm. A lot of information regarding the distributed system may be inaccessible to components during the time of programming as well as during the runtime. Components are not aware of the network topology, the network latency, or the number of computers on the network. Not only that, in some systems these aspects of the network may change during execution. Often, individual computers have an incomplete perspective of the system.

Distributed computing may be implemented using a variety of system architectures. Common among them include the client-server, n-tier, clustered, peer-to-peer, and space based architectures. The architecture is responsible for interconnecting multiple CPUs as well as individual program processes and impacts the communication links of the network. Processes could communicate with one another through a “master/slave” relationship or with the use of a shared database.

It should be noted at this point that there is no formal distinction between “distributed computing” and “parallel computing”. The two have many overlaps and similarities. Both distributed computing as well as parallel computing describe a computation process in which subtasks run simultaneously on a number of independent components. “Distributed computing” is sometimes said to be a more loosely coupled form of “parallel computing” [6]. What loosely coupled means is that the components of distributed computing are less likely to have information regarding other components within the system.

The algorithms of interest in this paper are designed and intended for use with distributed computing on a computer cluster system architecture. Computation is accomplished using a network of virtual computers, known as instances, existing on the Amazon Web Service cloud computing service. Centralized management is provided by the Spark Cluster Computing platform, which will provide a
number of services and functions relevant to cluster computing and well as to programming a distributed algorithm.

2.3 MapReduce Paradigm

MapReduce is a programming model useful for the distributed computing of an algorithm on large datasets [7]. While similar models had been employed in the past, the specific model was named and popularized by a paper from Google Research employees Jeffrey Dean and Sanjay Ghemawat published in 2004 [8]. The MapReduce was used by Google to index the World Wide Web. Other examples of common applications for MapReduce include web access log statistics, distributed sorting, document clustering, inverted index construction, etc.

The two primary steps of MapReduce are the map() and reduce() functions borrowed from the functional programming paradigm. Map() is responsible for partitioning and then filtering a dataset. In the Reduce() phase, the data set returned by Map() is shuffled, sorted, and summarized. This process is reflected in Figure 2. The static inputs and output datasets of MapReduce are frequently stored in a distributed file system while data during or between MapReduce tasks is usually stored locally.
MapReduce is best used to take advantage of the shuffle function of the platform. When using the model, the partition function and amount of data written by Map() are two heavy costs that must be considered. Communication cost, which has potential to be greater than the computation cost, is another aspect that must be considered. MapReduce provides the most benefit for large programs that require scalability or fault tolerance. MapReduce does not provide an advantage for smaller programs that may fit into the memory of a single machine or cluster.

MapReduce is not without its flaws. Because of its rigid two step map() and reduce() process, MapReduce tasks must be written as acyclic dataflow programs with a stateless mapper and reducer executed on batch job scheduler [9]. This limitation causes MapReduce to be imperfect solution for applications requiring the iterative processing of a data set such as in Machine Learning which must pay the overhead of starting and ending multiple MapReduce jobs as well as the costs storing and then reloading data processed in the prior step of the iterative process.

2.4 Apache Hadoop Cluster Computing Framework

Apache Hadoop is a popular, open source cluster computing framework. The framework offers an implementation of the MapReduce programming model and offers libraries useful for the distributed processing and storage of large datasets on a computer cluster. The clusters are built from commodity hardware and are highly scalable in size so that thousands of machines each offering local computation and storage may be supported [10]. A key feature of Hadoop is its fault tolerance. Hadoop detects and hardware failures and resolves the issue at the application layer.

Apache Hadoop traces its origins back to the Nutch project that began in 2002. The internet was much younger technology at the time and a number of companies were interested in a means to index webpages. Resources to accomplish this were limited, however. The goal of the Nutch project was to build an open source web-scale, crawler based search engine. It relied on distributed processing as a necessity and included sort and merge based processing. Nutch was demonstrated on 4 nodes but web scale was far from achieved.
In 2004, the Google File System and MapReduce papers were published that addressed Nutch’s scaling issues. Incorporating the Google ideas, Nutch was able to expand to 20 nodes. Web scale was still far away. Then in 2006, interest from Yahoo! in the Google papers motivated the company to hire Doug Cutting, one of the two primary Nutch engineers. The processing and storage components of Nutch were spun off into Hadoop, which became an open source Apache Software Foundation project. The Nutch web crawler remained within Nutch.

With the help of Yahoo financing, Hadoop was finally able to achieve web scale and scale on to tens of thousands of computers in 2008. Yahoo moved its web index onto Hadoop as well as merged its search and advertising departments into one units to capitalize on the new technology. For-profit companies, such as HortonWorks and Cloudera, emerged that focused on the support and development of Hadoop. Hence, the emergence of the Hadoop industry.

The Hadoop framework is composed of four core modules:

1. Hadoop Common: Utilities that are shared by all modules.
2. Hadoop Distributed File System (HDFS): A distributed file system designed to be capable of reliably storing large files across the cluster.
3. Hadoop YARN: Framework responsible for job scheduling and managing cluster resources
4. Hadoop MapReduce: Hadoop’s implementation of the MapReduce programming model on a YARN-based system.

In addition to these modules, Hadoop has fostered an expansive ecosystem. Notable projects within include HBase, Hive, Mahout, and Pig. Hive and Pig have been attempts to bring SQL or SQL-like query language to Hadoop [11][12].

2.5 Apache Spark

Apache Spark is an open source, cluster computing framework. It is a general processing engine that is compatible with various components of the Hadoop ecosystem, such as HBase, Hive, and Hadoop InputFormat, and can be used to replace the responsibilities of Hadoop MapReduce. Spark may be used
with various different cluster manager and distributed storage systems. Options in terms of cluster
management include native Spark cluster, Hadoop YARN, or Apache MESOS. In terms of distributed
storage, choices include Hadoop Distributed File System (HDFS), Cassandra, and Amazon S3.

Spark traces its origins back as a UC Berkeley AMPLab research project by Matei Zaharia in
2009. It was open sourced in early 2010 and then donated to the Apache Software Foundation in 2013.
Spark became an Apache top level project in February 2014. Spark was the most active project on Apache
in 2014 with over 465 contributors and is among the most active “Big Data” projects.

In contrast to Hadoop MapReduce, Spark is capable of real time streaming and interactive queries
in addition to batch processing. Spark ability to cache datasets so that they may be queried repeatedly as
well as its support for iterative programming makes the framework more suitable for machine learning
applications than Hadoop’s two stage MapReduce model. In addition, Spark offers easy to program APIs
for Scala, Java, and Python.

Spark has been shown to drastically outperform Hadoop MapReduce in a number of different
areas. In terms of SQL queries, Shark, the Spark implementation of Hadoop’s Hive, has been shown to be
80x faster than Hive for a selection query of in-memory data and 5x faster when the data is read from
HDFS [13]. In terms of machine learning algorithms, Shark was found to be 100x faster than Hive and
Hadoop for logistic regression and 30x faster for k-means [9]. In the Daytona Graysort sorting benchmark
competition, Spark was able to sort the same 100 TB of data 3x faster while using 10x fewer machines than
Hadoop MapReduce [14].

In terms of the Spark ecosystem relevant to this paper, Spark SQL provides support for structure
and semi-structured data as well as the SQL language. Spark SQL is the successor to Shark and its behavior
is a focus of this paper.

2.6 Spark SQL File Formats

The file format used for storage in cluster computing programs is very important. The file format
may have a measurable impact on execution time of the program depending on the encoding and decoding
speed of the data. The file format will also have a great impact on the file size of datasets. A compact file format may be desired to fit the dataset into storage or for greater portability. Some common concerns when deciding between file formats are space efficiency, query efficiency, and interoperability.

Spark SQL is a module that is used in the Particle Swarm Optimization algorithm program in this paper to query datasets. The module supports two file formats: Parquet and JSON. The two file formats differ in that one is a columnar-oriented dataset while the other is a row-oriented dataset. In the upcoming subsections, the similarities and differences between the Parquet and JSON file formats will be discussed in greater depth. The two different file formats are compared under different environments in this paper.

2.7 Particle Swarm Optimization

![Figure 3. Structure of a Particle](image)

Particle Swarm Optimization (PSO) is a young machine learning algorithm introduced by Kennedy and Eberhart in 1995 [15]. The inspiration for the algorithm comes from social psychology and the tendency for a group of individuals to share information rather than individual cognitive ability which is more commonly focused on. The goal of the PSO algorithm is to find good parameters for another algorithm that is to have it parameters permuted. The name “Particle Swarm Optimization” comes from an analogy between its solution search methodology and that of an initially disperse flock of birds converging together as they look for corn.

The Particle Swarm Optimization maintains a population of candidate solutions referred to as "particles" [16]. Each particle contains a set of algorithm parameters, represented by a vector of variables, to be permuted. The “search space” refers to the possible values for the vector of variables. The PSO algorithm then permutes the parameters of the other algorithm by iteratively testing and improving the candidate solutions. Particles move about in the search space dictated by a formula that is partly determined
by the particle’s current position and velocity. To clarify, a particle “moves” when its vector of variables are adjusted and how the vector of variables are adjusted are based on the current value of the vector as well as the velocity value that is dictated by the Particle Swarm Optimization algorithm.

The Particle Swarm Optimization algorithm considers the particle’s best known local position as well as the best known position in the search space when moving around particles. Because the velocity of each local particle is influenced by the same best global position value, the result is a “swarm” of particles that converges toward an optimal solution. The entire population of particles, or the swarm, may be thought of as a flock of birds while the individual particles may be thought of as individual birds.

A characteristic that makes Particle Swarm Optimization a good optimization algorithm choice to use in this paper is that it is metaheuristic. Metaheuristic optimization strategies work by guiding the search process for an optimal solution within a search space [17]. Metaheuristic algorithms do not guarantee solving for the optimal solution to a problem nor does it guarantee that a sufficiently good but non-optimal solution is found. Rather, the goal of metaheuristic algorithms is merely to hope to find a sufficiently good solution.

In exchange for these limitations, metaheuristic algorithms offer a number of advantages. It does not attempt to optimize algorithms by analyzing gradient so that non-differentiable algorithms can be solved. This is in contrast with many traditional learning optimization algorithms such as gradient descent. Because of such shortcuts, metaheuristics tend require much less computational work than traditional optimization algorithms and may operate on a set of solutions too large to be fully sampled. It may be generalized from these qualities that metaheuristics is most suitable for problems that are noisy or the optimal solution changes over time.

Given these characteristics, it makes sense why the application of Particle Swarm Optimization in this paper. The trading algorithm in this paper is not differentiable. The optimal solution to trading algorithms change over time so that the potentially non-optimal solution limitation does not apply to the algorithm having its parameters permuted. The reason the solution changes is that trading algorithm is intended to predict future stock market behavior. Future behavior is clearly unavailable to test in the present and a set of parameters may work better in some time frames (ex. the year 2020, the month of December
2016, etc.) than others. Finally, the search space for the problem is so big that sampling all solutions would be impractical with present technology.

2.8 Trading Strategies

The inspiration for the ATT stock trading algorithm implemented in this paper is grounded from investment strategies found in the technical analysis division of the security analysis discipline. These strategies are converted into “trading rules”, which embody popular use cases for the strategies. The ATT algorithm consults the trading rules in order to make decisions whether to buy, sell, or hold a specific stock. In this subsection, the technical analysis strategies of which the ATT’s trading rules are based on will be briefed. More detailed information, specifically how trading rules were implemented, may be found in the Appendix.

Prior to discussion involving specific trading strategies, some basic concepts in developing trading strategies will be discussed. Some technical analysis strategies are based on measuring “trend” or “momentum”. Trend is the general direction of the price of an asset. One common basis for a trading strategy is to observe patterns where stock price is trending upward and identify those where stock price is likely to continue this uptrend. Simply stated, “follow the direction of the market”. Other technical analysis strategies look for “trend reversals”. The idea is to observe pattern where stock price is in a sideways or downward trend and identify those where the trend is likely poised to reverse. Otherwise stated, “Buy low and sell high”. There are also strategies that combine following trend and identifying trend reversals. These use one of the two depending on circumstances.

Moving Average is a technical analysis strategy based on the statistical concept of moving averages [18]. The moving average is a means to smooth out and reduce fluctuations in time series data in order to generalize longer term trends. Moving averages are, therefore, useful in determining if a stock price is trending upward or downward. The Moving Average rule in this paper calculates a short and long term moving average. A buy signal is generated when the short term moving average is rises above the long
term moving average and a sell signal is generated when the short term moving average falls below the long term moving average. This suggests to buy stocks based on strengthening trend.

Trading Range Breakout calculates the highest and lowest price of a stock during a set review period [19]. If the price of the stock rises above the high in the review period, a buy signal is generated. Conversely, if the price of the stock falls below the low in the review period, a sell signal is generated. The idea behind the Trading Range Breakout is that stock price rising past its previous high indicates extraordinary buying demand. This high demand is thought to be likely to continue before gradually dying out. Hence, the strategy suggests to buy the stock.

Bollinger Band is a trading strategy operating much under the same logic as Trading Range Breakout except, instead of using stock price highs and lows, Bollinger Band takes the moving average of the stock price for a given number of days and calculates the value for a given number of standard deviations above and below the moving average [20]. A buy signal is generated if price rises above the higher band, indicating extraordinary demand, and a sell signal is generated if the price falls below the lower band, indicating extraordinary selling pressure.

Stochastics, in technical analysis, is a strategy that measures momentum using support and resistance levels [21]. The term “stochastics” is borrowed from mathematics and refers to the relation between the current price and its price range. In the algorithmic trading rule implementation, a comparison between the current price and the highs and lows of the stock within a specified period is made. A line is calculated using a moving average of stochastic values. Then another line is calculated from a moving average of the previous line. Signals are generated based on intersection of the two lines and overbought and oversold levels based on the stock price at the time of the intersection.

Relative Strength Index (RSI) is a technical analysis strategy that measures the momentum of a stock based on the velocity and magnitude of up and down price changes [22]. The rationale behind the strategy is that when stock price moves up too quickly, it is considered overbought and a decline in stock price is inevitable. The same is true for when a stock moves down quickly and the stock price bouncing back up. The average of all up price movements is divided with the average of all down price movements
in the trading rule. An RSI value is calculated by the rule and buy and sell signals are based on the overbought and oversold levels crossed by the RSI value.

Moving Average Convergence Divergence (MACD) is a technical analysis strategy intended to measure momentum by analyzing the relationship between two moving averages of prices [23]. In the trading rule implementation, the MACD value is the difference between two exponential moving averages of a stock is taken, the former of a shorter window and the latter of a larger window. An exponential moving average of MACD values is then calculated. Buy and sell signals are generated when the current MACD value crosses over the exponential moving average of MACD values.

On Balance Volume (OBV) is a technical analysis strategy that analyzes the relation between volume and price [24]. Volume is the quantity of shares involved in a stock market transaction. The volume for all transactions in a single day is a commonly taken into account in technical analysis strategies. To calculate an OBV value, the volume on days when the stock price is up is added to the value and the volume of the days when the stock price is down is subtracted from the value. The rationale behind OBV is that volume tends to be greater during an uptrend or downtrend rather than a minor fluctuation in stock value. In the application in this paper, the moving average of volumes for a small window is taken as well a moving average of volumes for a larger window and a signal is generated by their intersection.
3. JSON AND PARQUET FILE FORMATS

3.1 Row Oriented vs. Column Oriented Datasets

Prior to discussing the intricacies of the Parquet and JSON file formats, it is important to understand the distinction between row oriented and column oriented datasets. A row oriented dataset is the more common dataset type. It stores data by serializing each row of data. In contrast, a column oriented dataset serializes columns of data. For example, consider the table in Figure 4.

![Figure 4. Dataset Example](image)

In a row-oriented dataset, the data would be stored like this:

01: Apple, Red;
02: Pear, Green;
03: Banana, Yellow;
04: Avocado, Green;

In contrast, in a column-oriented dataset, the data would be stored like this:

Apple: 01, Pear: 02, Banana: 03, Avocado: 04;
Red: 01; Green: 02, 04; Yellow: 03;

Note that it is possible to store more than one row value per record in column-oriented format. Such may improve the query times of such operations as selecting all rows with “Green”, which may be retrieved with a single operation or counting matching records.

If, however, one were to query for the entire row of records, then column-oriented dataset would be less efficient. In this case, the query could be attained with a single operation for the row-oriented dataset but require multiple operations on multiple columns to attain the records for a single row for the column oriented dataset.

Column oriented file formats also tend to have more opportunities for compression than row oriented file formats. This is because each column has a uniform data type. This is particularly useful for
compression algorithms that use similarity of adjacent data as a strategy for compression such as LZW or run-length encoding. Missing and repeated data may be represented by two bit markers. Adjacent compression may come at the cost of query speed, however, as random access becomes more difficult with increased compression and decompression may be required during reading.

Another aspect of uniform data types is that it optimizes processor pipeline dependence, thereby decreasing “bubbles” and improving querying speed [25]. In instruction pipelining, a processor has a number of instructions and, rather than process the instructions sequentially, the processor will process instructions concurrently. For branch instructions, instructions that cause the processor to execute instructions out of order, the processor predicts the next instruction. A series of mispredictions result in a “bubble” that hurts performance. “If” statements, loops, virtual calls, and data dependencies all cause bubbles.

Processor bubbles occur more often in row oriented datasets because, in a row oriented structure, data types in rows are interweaved and must be encoded one at a time. This increases branching. In a column oriented structure, because data types of column are uniform, an array of values can be encoded at once and this makes instruction branching more predictable.

Column oriented datasets also have potential to improve processing speed by reducing CPU cache misses [26]. A cache miss occurs when there is a failed attempt to read data from the CPU cache. This results in the need for main memory access. In a column oriented dataset, when decoding a column, data from other columns does not need to be cached so it fits easier into the processor cache.

3.2 JSON

Java Script Object Notation (JSON) is an open standard file format consisting of attribute-value pairs. JSON is most well known as the most popular file format for data-interchange between a server and a web application. It appears in major web services such as Facebook and Twitter. JSON has also been heavily adapted for use in mobile and web application development. Because JSON is programming
language independent while at the same time adhering to the conventions followed by the ubiquitous C-family languages, it is well situated as a data-interchange language.

In a more recent developments, JSON has been adapted for use as a file format for large datasets [27]. Such is the use case in this paper. As a database file format, a major part of JSON’s appeal is its simplicity. JSON is human readable and easy for readers to understand. This contrasts with other database file formats and makes JSON easier to debug. Human readability is especially relevant when translating between two file formats because having a machine translate two unreadable formats is prone to error.

Another reason for JSON’s increased adoption as a database format is its agility. Large datasets with strict schemas can be difficult to update. Data added needs to adhere to the schema and adjusting the schema of a large dataset can be cumbersome. JSON lacks predefined schema and this means it is easier to update data as a schema does not need to be adjusted or adhered to.

Of course, JSON’s many advantages are not without their disadvantages. JSON datasets could be difficult to encode or decode. One challenge of JSON datasets is that due to the ability to ignore schema all together that when it comes time to define a schema for a database it will be a difficult task. The JSON could have non-uniform, or inconsistent, schemas among a list of objects. An example of JSON with non-uniform schema that would be difficult to handle is:

```
{“field-a”:“apple”}
{“field-a”:“banana”,“field-b”:“yellow”}
```

Other issues JSON datasets face is that JSON files often contain complex nested data structures and could have multi-type fields. This could make accessing specific fields in a JSON dataset difficult when querying. A final major issue that plagues JSON datasets is that, while the files are human readable, they are considered bloated for datasets. Property names must be written for every attribute-value pair in an object and, in a dataset with a uniform schema, this should not be necessary.
3.3 Parquet

While many may be already familiar with JSON, not as many individuals may be familiar with Apache Parquet. Parquet is an open-source, column-oriented file format for the Hadoop ecosystem developed as a result of a collaboration between Twitter and Cloudera. The file format was inspired by the Google Dremel papers [1], which introduced an innovative, efficient new means to represent nested data structures in a column oriented form.

The many potential benefits of column-oriented datasets have already been noted in the “Column-oriented vs. row-oriented” datasets subsection. To review, column-oriented datasets improve performance for queries interested in only the values in a couple columns of a row and not all the values in the row, column-oriented datasets have opportunities for compression that do not exist for row-oriented datasets, column-oriented datasets may be more optimized for pipeline dependance, and column-oriented datasets may reduce the number of CPU misses.

In addition to the benefits of column-oriented datasets, the Parquet file format has a couple extra features. The Parquet file format maintains statistics, such as min and max, for “column blocks” that help reduce the amount of data that needs to be processed during a query [27]. The statistics are useful for “predicate pushdown” process. Predicate pushdown is a strategy to run operations that filter data as close to the beginning of the map reduce pipeline as possible. This reduces network traffic.

The Parquet file format also supports flat schema files [26]. What is meant by this is not only does the Parquet file format omit extraneous metadata used exclusively for nested schemas but it includes encoding algorithms specifically aimed for flat schemas. One such algorithm is Run Length Encoding. Run length encoding is a lossless data compression strategy that identifies “runs”, or patterns of consecutively repeating data, and stores the runs as a single data value and a count.

Bitpacking is another encoding algorithm used by Parquet to improve performance. Bitpacking is an encoding algorithm that inserts non-byte-size data into primitive data types so that they may output in more regular data sizes. Regular sizes increase predictability in the processor pipeline and hence improves CPU efficiency.
4. ADAPTIVE TECHNICAL TRADING ALGORITHM

4.1 Introduction

The objective of this project is to permute a set of effective parameters for the Adaptive Technical Trading (ATT) algorithm using the Particle Swarm Optimization algorithm. The goal of the ATT algorithm itself is to serve as an effective stock trading or investing strategy.

The intended use case of the ATT algorithm is to generate a buy, sell, or hold signal for each stock listing of interest each day. The foundation of its logic are a number of “base rules” that inspired by strategies popular in technical analysis of stock trading. The rules were briefed in the Background section. The base rules are parameterized and different combinations of parameters applied to the base rules then form “component rules”. The signal is generated through the combined input of a number of “component rules”. The success of the ATT algorithm will be measured by the profits it theoretically generates.

What distinguishes ATT from many other trading algorithms is that it autonomously adjusts the degree of input, or “weight”, of each of the component rules on the buy signal depending on their prior performance. The algorithm reviews rule weights every given number of days or a period that may be referred to as the “review span”. There is also a limit on the age of profit history to be considered by the algorithm. This limit is set by the “memory span”.

4.2 Component Rules

It has already been said that the inspiration for the ATT algorithm is grounded from strategies found in the technical analysis division of the security analysis discipline. These strategies are used to derive “base rules”. The base rules, however, are not rigid. Different technical analysts may favor versions of the base rules with different parameters.

For example, one common trading strategy is the Moving Average. In the base rule, two moving averages of stock prices are taken. One moving average has a larger window and accounts for more prices. The other moving average has a smaller window and accounts for less prices. A buy signal is generated
when the moving average with the shorter window crosses over the moving average with the larger window. The Moving Average base rule may be interpreted to contain two different parameters: the size of the large window and the size of the small window. One variation of the Moving Average may have a small window of 5 days and a large window of 20 days. Another variation could have a small window of 20 days and a large window of 100 days. Both variations are valid uses of the Moving Average.

The ATT algorithm strives to account for not only one variant but many variations of trading rules. In this paper, the trading rules that are general ideas without specifying the parameters are referred to as “base rules”. Trading rules with parameters specified, such as a Moving Average with a small window of 5 days and a larger window of 20 days as exemplified, will be referred to as “component rules”. In order to account for a good number of rule variants, a set of possible values for each parameter of a base rule is preset. Then the ATT algorithm generates all possible base rule variants by making unique combinations from the sets of parameters.

The rule generation procedure is illustrated in Figure 5. Say the ATT algorithm is working to form variants of the Moving Average trading rule. The base rule for Moving Averages has two different
parameters, a large window and a small window. If the set of possible values for the large window were \{15, 20, 25, 30\} and the set of possible values for the small window were \{2, 5, 10\}, then the Cartesian product of the two sets is calculated and component rules for Moving Averages that may be generated include (15,2), (15, 5), (15, 10), (20, 2), (20, 5), (20,10), (25, 2), … Referring to the pairs as \((x, y)\), \(x\) is the value of the large window and \(y\) is the value of the small window. So Moving Average (15, 2), Moving Average (15, 5), Moving Average (15, 10) were some of the component rules generated from the sets of possible Moving Average parameters \{2, 5, 10\}, \{15, 20, 25, 30\}.

4.3 Generating a Signal

In order to attain the buy/sell signal of the ATT algorithm for a particular stock on a particular day, or time period, each component rule generates its own individual buy/sell signal. This signal is a number between 1 and -1, with 1 being the strongest “buy” signal and -1 being the strongest “sell” signal. The signal generated by the overall ATT algorithm may be expressed in the following formula:

\[
s_t = \sum_{i=1}^{1059} w_i s_{i,t}
\]

For \(s_t = \) overall signal, \(w_i = \) component rule weight, \(s_{i,t} = \) component rule signal

Each component rule has a “weight”. The weight of the component rule determines the degree of input its signal has on the final signal of ATT. The sum of the weights of all component rules is 1. The signal of each component rule is calculated and multiplied by a value that is its weight. The resulting terms are then summed together and this is the signal for the overall ATT algorithm. This signal should also be 1 or -1.

It would be disingenuous to either buy or sell a stock holding based on weak signal. There are transaction costs associated with buying and selling stocks. Therefore, “buy” and “sell” thresholds exist to ensure only strong enough signals are considered. The signal must either be greater than the overbought threshold or less than the oversold threshold to generate a buy or sell signal. Otherwise, the signal is
deemed insignificant and a “hold” signal is generated so that the stock holding is neither increased nor decreased upon for the day.

Figure 6. Component Rule Signal Generation Process

Figure 6 is a flow chart representation of the component rule signal generation process as well as the context of its use within the broader ATT algorithm. Specifically, the “signal generation” described in this subsection is rather simple with a process to “Begin analysis of new comp rule”, “Calculate comp rule signal”, and then “Record results”. After completion, the ATT algorithm checks for the next component rule and does the “signal generation” calculation for that. When the set of all component rules in the ATT algorithm have had the “signal generation” done, the ATT algorithm finally makes the decision whether to buy or sell a stock. After the decision, the algorithm checks for the next stock symbol covered in the ATT algorithm and repeats the entire process for that.

4.4 Adjusting Weights

This subsection details how component rules are adjusted during the runtime of the ATT algorithm. Component rule weight adjustment during runtime is intended to make the algorithm flexible
and adapt based on market trends. For example, some component rules may be more successful during the fall and winter quarters of the year than the summer and spring. This adjustment helps the rule gain weight when it has more predictive power. Note there exists also an initial weight for component rules, which is the weight for which component rules start off before being adjusted, to generalize performance of rules over the long term; however, this is set using Particle Swarm Optimization prior to runtime and will not be touched on further in this subsection.

There are two primary variables that are used that affect the calculation of the component rule weight adjustment: “review span” and “memory span”. The review span is pretty straightforward. While the ATT algorithm calculates a signal for each component rule every day it, adjusts component rule weights during runtime every “review span” days. The purpose of the memory span is to limit the number of trading days the algorithm looks back when calculating weight adjustment.

The reason for the need for the memory span are multifaceted and involve the fact that weights are adjusted during runtime based on market trends. First, the number of days analyzed could be considered the length of a trend. This length should be the same for the life of the algorithm to have consistent performance during different time periods and the memory span ensures this. Second, it is desirable to regulate the particular length of trends being analyzed. There exist short trends that may last a week as well as long trends that last years. Part of the goal of applying the Particle Swarm Optimization on the ATT algorithm is to find the length of the trend analyzed with the best results.

The specific procedure to adjust weights involves two separate steps. The first step reduces the weight of each unprofitable component rules during the memory span period. The formula used to accomplish this is:

\[ w_i = w_i - \left( \frac{r_f}{N_{tot}} \right) \left( \frac{N_{pos}}{N_{tot}} \right) \], if profit < 0

For \( w_i \) = component rule weight, \( r_f \) = reward factor, \( N_{tot} \) = total number of component rules, \( N_{pos} \) = total number of profitable component rules

Prior to adjusting the weight, the program must evaluate the profitability of each component rule and count the number of component rules that are profitable within the memory span. The number of
profitable rules is reflected by the variable $N_{pos}$. The reward factor ($r_f$) is a preset variable with the sole purpose of adjusting the amount of weight that should be rewards and penalized for component rules. This reward factor is divided by the total number of all component rules ($N_{tot}$), regardless of positive or negative profitability, to make the amount of penalty from $r_f$ proportionate to the number of existing component rules.

The next term divides the number of positive component rules with the total number of component rules ($N_{tot}$). The term is then multiplied with the earlier term calculated by dividing $r_f$ and $N_{tot}$. The reason for the term dividing $N_{pos}$ with $N_{tot}$, like the term calculated in the prior paragraph, is to adjust the amount of weight penalized. As one may notice, $N_{pos}$ divided by $N_{tot}$ is the percentage of all component rules that are profitable. It makes sense, therefore, that the purpose behind the need for the term is to adjust the amount of weight penalized to be proportionate or relative to performance of other rules. The need for this term involves the next step relating to rewarding profitable component rules and will be explained in a little bit. After the first and second terms are multiplied together, they are subtracted from the previous component rule weight and the new component rule weight is calculated.

The formula used to reward profitable rules is rather simple:

$$w_i = w_i + \frac{D}{N_{pos}}, \text{ if profit > 0}$$

For $w_i = \text{component rule weight}$, $D = \text{sum of weight deducted from unprofitable rules}$, $N_{pos} = \text{total number of profitable component rules}$

The sum of weight deducted from unprofitable rules is calculated. Then this sum is divided by the number of profitable component rules. The weight of each profitable rule is then increased by the term. What this effectively does is that it divides up the amount of weight deducted from unprofitable rules and redistributes the weight evenly among the profitable rules.

Skipping back into the discussion of how component rules are penalized, the term $N_{pos}$ divided by $N_{tot}$ is needed to control for the event where there are few profitable rules and many unprofitable rules. If there were a total of 100 components rules and 99 component rules were unprofitable while a single component rule is profitable, the single component rule would attain a large amount of weight. The issues
arising from this are that this could bias the algorithm toward a few rules while the other component rules would be unduly penalized when not performing much worse than each other. By dividing $N_{pos}$ with $N_{tot}$, or the percentage of profitable rules, and multiplying that with the term containing the reward factor, the relative performance of other rules is considered so that a smaller number of profitable rules would receive a smaller share of rewards and vice versa.

![Flow Chart](image)

Figure 7. Adjusting Weights Flow Chart

Figure 7 is a flow chart representation of the “Adjusting Weights” process as well as the context of it use within the broader ATT algorithm. The process begins with “Calculating the profits of each comp rule”. In this step, all the signals generated for each symbol in the specified memory span by the component rule will be considered and used to calculate its profit within the memory span. This will be detailed further in the next subsection.

The results of the “Calculate profits” step are recorded so that in the next step the number of profitable rules can easily be counted. This is used in the aforementioned weight adjustment formula for unprofitable rules. The next step in the “Adjusting Weights” process is to adjust the weight of each of the unprofitable rules. During this phase, the total amount of weight deducted among all rules must be kept track. After deducting weight from every unprofitable rule, the algorithm adds weight to each of the profitable rules. The formula to do this makes uses of the sum of weight deducted from unprofitable rules that was recorded.
4.5 Calculating Component Rule Profits

Much consideration was taken into account with how to calculate the profit for component rules. The process is not as clear cut as it appears because of the “memory span” component of the algorithm. To recall, the memory span is the number of days the ATT algorithm looks back when identifying a profitable component rule. The issue comes from uncertainty over how to initialize the holdings for a component rule when calculating profit. The profit calculation for the component rule can start off with no holdings, some holdings, or either no or some holdings depending on whether the rule had holdings prior to the memory span used for calculation.

The problem with starting with no holdings, while deceptively the natural choice, is that this inherently gives “buy” signals of component rule more weight than “sell” signals. Assuming that there is no shorting or holding a negative number of shares, a “sell” signal that appears before “buy” signal would be ignored while a “buy” signal before a “sell” signal would be considered. Shorting, the holding of a negative number of shares, may be enabled during profit calculation. This, however, inherently is not consistent with the overall ATT algorithm that does not support shorting. It wouldn’t make sense to consider profits of rules attained from shorting when adjusting weights if the overall algorithm cannot short shares.

Then the next logical consideration is to calculate profit based on the amount of holdings the component rule had prior to the memory span. This does not suffer from bias of considering “buy” signals more than “sell” signals. The problem with this is that it conflicts with the purpose of the memory span, which is to limit analysis to only memory span days. Especially disconcerting is if a component rule made a single signal to buy a long time ago prior to the memory span, did not produce any additional signals, and then that buy signal from a long time ago has a large impact on the rule’s profit calculation. Either rewarding or penalizing the component rule based on this single outdated signal would be wrong because it does not reflect any of its current performance.

The ultimate compromise that was used for this project is to start with a set amount of holdings at the beginning of the memory span. This way, a “sell” signal may be produced and considered from the very beginning so there is no inherent bias toward “buy” signals. To control for component rules that produce absolutely no signals during the duration of the profit analysis, profit and loss is not calculated relative to 0
but relative to how much the rule would have profited or lost from holding the number of shares at the
beginning of the profit analysis throughout the duration of the memory span. For example, if profit
calculation is initialized with 1 share of a stock and the stock started off at $11 in the beginning and ended
at $10, then whether the component rule was profitable would be based on whether it made more or less
than negative $1.

4.6 Algorithm Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial component rule weights*</td>
<td>[-1, 1]</td>
</tr>
<tr>
<td>Memory span</td>
<td>[150, 300]</td>
</tr>
<tr>
<td>Review span</td>
<td>[20, 150]</td>
</tr>
<tr>
<td>Reward factor</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>Buy threshold</td>
<td>(0, 0.3)</td>
</tr>
<tr>
<td>Sell threshold</td>
<td>[-0.3, 0]</td>
</tr>
</tbody>
</table>

Figure 8. Table of ATT Parameters

It has been discussed that in this paper the Particle Swarm Optimization will be applied to set the
parameters of the Adaptive Technical Trading algorithm. Given this application, it makes sense that there
should be ATT algorithm parameters that could be permuted through and set by the Particle Swarm
Optimization algorithm. Most of the parameters in this section have already been discussed in prior
subsections of the paper but for the sake of clarity with regards to this important aspect of the Particle
Swarm Optimization application each parameter as well as its possible values will be noted and briefed in
this subsection.

In the “component rules” subsection, it had been established that there exist 582 individual
component rules in the ATT algorithm. Each component rule has a weight that determines its impact on the
ATT buy/sell signal and that weight is adjusted during the duration of the ATT algorithm based on
performance. The weight each component rule begins with, however, is not the same. Lucrative rules based on historical performance should begin with more weight to save on the number of Particle Swarm Optimization cycles required to adjust rule parameters. The first 582 parameters of the ATT algorithm are the “initial component rule weights”.

The “memory span” is a parameter of the ATT algorithm that had been discussed in the “adjusting weight” subsection. In theory, the purpose of the memory span is to limit the number of trading days the algorithm looks back when calculating weight adjustment in order to capture recent market trends. In practice, it is used in the weight adjustment formulae:

\[ w_t = w_{t-1} - \left( \frac{r_f}{N_{tot}} \right) \left( \frac{N_{pos}}{N_{tot}} \right) \] \[ w_t = w_{t-1} + \frac{D}{N_{pos}} \]

where \( N_{pos} \) is the number of profitable component rules given analysis of the past memory span number days. The memory span in the algorithm is set between 150 and 300 so the number of days when considering weight adjustment and component rule performance is between 150 and 300.

The “review span” was also discussed in the “adjusting weight” subsection. The review span is simply the number of trading days that past before component rule weights are evaluated for their recent profitability and adjusted. The review span is between 20 and 150 days. Note that the review span is less than the memory span of the ATT algorithm. That means that certain days will be considered more than once when evaluating component rule weight adjustment.

The “reward factor” parameter was discussed along with the memory span and review span and has a role in affecting component weight adjustment. It appears as \( r_f \) in the weight adjustment formula:

\[ w_t = w_{t-1} - \left( \frac{r_f}{N_{tot}} \right) \left( \frac{N_{pos}}{N_{tot}} \right) \]

used to deduct weight from unprofitable component rules. That means that the higher the reward factor, the greater amount of weight a component rule will lose for being unprofitable. Conversely, the higher the reward factor, the greater amount of reward for profitable rules. The value of the reward factor is between 0 and 1 so the amount of weight deducted from each rule will be between that range.

The “buy threshold” and “sell threshold” parameters were mentioned in the “Generating a Signal” subsection. Recall that the ATT algorithm generates a numeric signal between the values of -1 and 1 whether to buy or sell. Component rules generate either a -1, 0, or 1 as a signal and the sum of the weights
of all component rules is 1. The thresholds exist to determine whether signals are significant enough to act upon or whether the profit to risk ratio is too low to be worth following. The boundaries for the rules are (0, 0.3] and [-0.3 and 0). In the most extreme case for the buy threshold, what this means is that in order for the ATT algorithm to act on a buy signal, component rules totaling 30% or more of component rule weight must generate a buy signal.

4.7 ATT Algorithm Overview

In the prior subsections, individual segments of the Adaptive Trading Algorithm had been described, such as how to generate component rules, generate a buy/sell signal, and how to adjust weights of component rules. In this subsection, the concepts are all put together to walk through the entirety of the Adaptive Trading Algorithm. Figure 9 features the flow map of a sequential implementation of the algorithm.

The ATT algorithm begins by initializing its parameters. These are the initial component rule weights, memory span, review span, reward factor, buy threshold and sell threshold. After this, the first decision the algorithm faces is whether there are any unanalyzed trading days remaining. Since the program has just begun, there should be sufficient number of days and the algorithm proceeds with “yes” to the next step. The next step should be the decision whether it is time to update the component rule weights. When to update is dictated by the algorithm’s review span. Again, since the algorithm has just started, it makes sense to proceed “no”.

The next step in the algorithm is to “Begin analysis of a new time period”. In this paper, the time period is merely a single trading day; but, the time period could potentially be set to seconds, weeks, months, etc. The algorithm then begins a “Symbol Signal Calculation” process. The initialization of the time period is important to note because the “Symbol Signal Calculation” process is performed a number of times equal to the number of symbols (AAPL, MSFT, BAC, etc.) that are being analyzed by the ATT algorithm for each time period.
The first step of the “Symbol Signal Calculation” process is to “Begin analysis of a new symbol”. For instance, this “Symbol Signal Calculation” may be devoted to evaluating the MSFT stock symbol. The algorithm then proceeds to a “Comp Rule Signal Calculation” process. For each “Symbol Signal Calculation” process, the “Comp Rule Signal Calculation” process is performed a number of times equal to the number of component rules that exist in the ATT Algorithm. During the “Comp Rule Signal Calculation”, the algorithm proceeds into the steps “Begin analysis of new comp rule”. An example of a component rule the “Comp Rule Signal Calculation” may be devoted to is the Moving Average (20, 2) rule. The algorithm continues with the “Calculate comp rule signal” and “Record results and add on to total signal” steps.

After the termination of an individual “Comp Rule Signal Calculation” process, the ATT algorithm encounters a decision to check if this was the last existing component rule that needs to be evaluated. If not, it initializes a “Comp Rule Signal Calculation” process for the next unanalyzed component rule. If so, it proceeds to make the decision whether the result of all the component rule signals trigger an overall ATT algorithm buy or sell signal. The formula for generating a signal was covered in subsection 3.3. If either a buy or sell signal is triggered, move on to “Update ATT stock holdings”. Either way, this decision or step marks the end of a single “Symbol Signal Calculation” process.

The next step of the ATT algorithm is the decision whether the symbol evaluated for by the “Symbol Signal Calculation” process was the last symbol. If it is not, another “Symbol Signal Calculation” process starts using the next unanalyzed symbol. If so, the algorithm goes back to the decision to check if there are more unanalyzed dates that was encountered in the beginning of this subsection. If there are more dates to process, analysis of a new time period will be initialized and this starts the cycle of “Symbol Signal Calculations” all over again. Prior to starting a new cycle, however, there is a decision to check for if it is “Time to update comp rule weights?” dependent on the algorithm’s review span. If it is not time to update, the algorithm proceeds to the step to initialize another time period as normal. If it is time to update, it takes a detour into the “Comp Rule Weight Adjustment” process before returning to the “Begin analysis of new time period” step.
Figure 9. Iterative Flow Map of ATT Algorithm
The “Comp Rule Weight Adjustment” process begins with “Calculating the profits of each comp rule”. All the signals generated for each symbol by the component rule within the specified memory span will be used to calculate the profit of the component rule of interest within the memory span. The results of the “Calculate profits” step are used in the next step to “Count the number of profitable comp rules”. The number of profitable component rules is a variable within the weight adjustment formula for unprofitable rules.

The “Comp Rule Weight Adjustment” process proceeds to a decision to check if there are “More unadjusted unprofitable comp rules?” The purpose of this decision is to ensure that each unprofitable component rule is adjusted. If such a component rule exists, the algorithm uses a formula to “Calculate and deduct comp rule deduction”. Then the algorithm proceeds to “Add deducted weight to total weight deducted”. This sum is used in the weight adjustment formula for profitable rules. The algorithm returns to the decision to check for “More unadjusted unprofitable rules”.

When there exist no more unprofitable rules to update, the algorithm proceeds to the decision “more unadjusted profitable rules?” If so, the next step is to “Calculate and add comp rule addition.” After this step, the algorithm returns to the decision. When there are no more profitable rules to adjust, the “Comp Rule Weight Adjustment” process ends. The algorithm redirects to the “Begin analysis of a new time period” step which starts a new loop of “Symbol Signal Calculations”. The ATT algorithm concludes back at the “More unanalyzed dates” decision. When there are no more unanalyzed dates, the algorithm proceeds to tally profits in the “Calculate ATT profits” step and then terminates.
5. ATT Algorithm Datasets

5.1 Improving ATT Efficiency

In the prior subsection, a sequential walkthrough of the ATT algorithm was given. But, in fact, there are more efficient means to go about evaluating the ATT algorithm in this paper. A significant number of calculations within the ATT algorithm are repeated.

Referring back to Figure 9, the “Calculate Comp Rule Signal” step within the “Comp Rule Signal Calculation” dotted box does not need to be calculated more than once. The “Comp Rule Signal” never changes between different particles and different PSO iterations, though, the “Total Signal” generated in the next step will change depending on the array of component rule weights of the particle.

So, in order to save time, rather than undertake the slow process of calculating the “Comp Rule Signal” over and over again, the “Comp Rule Signal” for each symbol on each day is only calculated once and then recorded in a file. When the ATT algorithm executes and reaches the step to “Calculate Comp Rule Signal”, rather than actually calculate the signal, the program can query the result quickly from the dataset.

The dataset that is generated is of particular interest in this paper as it is used to compare the performance of the JSON and Parquet file formats. The dataset that generated containing all the component rule signals is called “cal_trans.parquet” and “cal_trans.json”.

5.2 Dataset Representation

In this subsection, the representation of the “cal_trans” file in either JSON or Parquet file format is discussed. The dataset operated on represents the “component rule transactions”, or the buy/sell signal that each component rule generates for each symbol for each trading day.

An excerpt of the JSON version of the dataset is presented first as JSON is actually human readable. This is the first couple of lines from the file:

{"day":0,"symbol":"AAN","index1":0,"index2":0,"signal":1}
The first attribute, “day”, represents the day the buy/sell signal was generated. The next attribute, “symbol”, identifies the stock symbol the signal was generated for. The “index1” and “index2” attributes are used together to identify the specific component rule the object corresponds to. Finally, “signal” is the buy/sell signal for the stock. Each “row” is represented as an object and the objects are simply delimited by a line break.

The Parquet version of the program is more difficult to describe and multiple abstractions are required. Prior to abstracting the format, there are 3 levels to the file hierarchy that one need be aware of. The first is the highest level and referred to as a “row group”. This is a horizontal logical partitioning of the data by rows. It is prior to dividing the data by columns. A row group contains many “column chunks”.

The next level of is the “column chunk”, which is a chunk of data from a particular column. There exists a column chunk in each “row group” for each column in the data. The column chunk is basically the column bounded by the rows in the row group. Column chunks are guaranteed to be contiguous within a file. A column chunk contains many “pages”.

A “page” is the lowest level on the hierarchy. It divides up column chunks. A page is indivisible in terms of compression and encoding. There are multiple page types interleaved in a column chunk. Together, row groups, column chunks, and pages make up the parts of a Parquet file.

The general layout of a Parquet dataset is represented as follows:

4-byte magic number "PAR1"
<Column 1 Chunk 1 + Column Metadata>
<Column 2 Chunk 1 + Column Metadata>
...
<Column N Chunk 1 + Column Metadata>
<Column 1 Chunk 2 + Column Metadata>
<Column 2 Chunk 2 + Column Metadata>
...
<Column N Chunk 2 + Column Metadata>
...
<Column 1 Chunk M + Column Metadata>
<Column 2 Chunk M + Column Metadata>
...
<Column N Chunk M + Column Metadata>
File Metadata
4-byte length in bytes of file metadata
The object contained within the “<” and “>” symbols represent “column chunks”. Inside the “column chunks” are pages. At the very end of the Parquet dataset after all the data is the metadata. This allows for single pass writing. There exists metadata for the “file”, “column”, and “page header”. The metadata is fairly extensive and will not be detailed in this paper. A “page” contains the value of the data as well as “repetition” and “definition” levels, two values used to reconstruct a nested structure using the “record shredding and assembly” algorithm in the Dremel paper [1]. Since the dataset in this paper is a flat schema, it does not contain “repetition” or “definition” levels.

One column chunk within a Parquet file using the same data from the JSON example may be abstracted like this:

```
Day:01, Day:01, Day:01, Day:01, ...;
Index1:0, Index1:0, Index1:0, Index1:0, ...,;
Index2:1, Index2:2, Index2:3, Index2:4, ...,;
Signal:1, Signal:1, Signal:1, Signal:1, ...,;
```

Of course, the data in the actual Parquet file is likely compressed with an algorithm such as Run Length Encoding given the frequent repeating values and, once again, the column metadata is missing.

5.3 ATT Dataset Applicability

As mentioned before, the “cal_trans” dataset in this program has a flat schema. The schema is rather simple and contains an integer, a string, an integer, an integer, and an integer per data entry. This dataset has many repeating columnar values. This may be inferred from the JSON excerpt example. The primary query exercised on this dataset will ask for millions of rows of contiguous data at a time like this:

```
trans_hundred = DS.sqlc.sql("SELECT * FROM parquetTable
WHERE day >= " + str(q_day) + " AND day < " + str(last_day))
```

Given this knowledge, there are characteristics that make both JSON and Parquet file formats appropriate for the use case in this paper and there are characteristics that make both JSON and Parquet file formats inappropriate for the use case in this paper.
For JSON, the query for the entire rows of data in this program makes its row-oriented data structure optimal. However, the flat schema of the “cal_trans” dataset means that there will be many extraneous lines in a JSON dataset, which has a syntax intended for the support of complex nested structures. Also, the JSON decoder must be designed to support and infer complex schemas, which may be non-optimal for a flat schema.

For Parquet, the flat schema of the “cal_trans” dataset is supported by the Parquet file format. Not only does Parquet avoid extraneous nested schema metadata in a flat schema file, there exist specific encoding algorithms that should be very effective on the flat schema with repetitive values such as runtime encoding. The problem is that Parquet is a column-oriented dataset and the queries in this paper are better suited for a row-oriented dataset.

One of the goals of this paper is to compare the performance of the JSON and Parquet file formats for the on the Spark Cluster Computing platform. The breakdown of advantages and disadvantages of the two file formats in this use case makes this an interesting comparison.
6. PARTICLE SWARM OPTIMIZATION

6.1 Introduction

A brief overview of the Particle Swarm Optimization (PSO) algorithm had been given in the “Background” section 2.5. In the prior Background section, it was explained that the PSO moves around and tests the “position” of “particles”, an analogy for the testing and adjustment of a set of parameters, in order to attain an optimal set of parameters for an algorithm in a search space. This section will go into depth about how precisely the Particle Swarm Optimization was implemented.

The PSO algorithm begins by initializing a set of particles. The set of particles is divided into groups. Each particle has a vector containing the parameters of an algorithm that the PSO algorithm is intended to permute through and set. Random values are assigned to the initial values of the parameters of the particle. The PSO then begins an iterative process of testing the fit of the algorithm using the parameters of each of the particles. Each particle memorizes the best position that it has found so far during the iterative process.

6.2 Particle Velocity

After each testing for the fit of its current position, the position of the particle is adjusted. The adjustment is known as the “velocity” of the particle based on the physics concept of movement based on a magnitude and direction. The velocity of each particle is based on its own best position that had been found, the best position found by a particle in its group, and its current position. Given these conditions, there are many different variations of how the velocity of a particle may be calculated.

The formula used precisely in this paper to calculate the velocity of a particle is:

$$V_{k+1} = \lambda V_k + c_1 r_1 (p_{best} - p_k) + c_2 r_2 (g_{best} - p_k)$$
For $V_{k+1}$ = new velocity, $\lambda$ = inertia weight, $V_k$ = current velocity, $c_1$ and $c_2$ = acceleration coefficients, $r_1$ and $r_2$ = random numbers between 0 and 1, $p_{best}$ = particle’s best position, $p_k$ = particle’s current position, and $g_{best}$ = the global best position

The first term of the formula multiplies the current velocity with the “inertia weight.” The inertia weight dictates the degree of influence the particle’s current velocity has on the new velocity. The name is fitting as the greater the inertia weight the greater the influence of the particle’s current velocity on the new velocity and the less the particle position will move.

The next term of the formula dictates how much the influence of the particle’s former best position will have on the velocity. The current position is subtracted from the best position so that the difference is the velocity needed for the particle to move from its current position to its former position. An acceleration coefficient partly controls how much influence the best position should have and a random number contributes to the search aspect of the algorithm.

The last term dictates how much the influence of the global best position will have on the velocity of the particle. Similar to the best position, the current position is subtracted from the global best position to attain the velocity needed for the particle to move from its current position to the global best position. Another acceleration coefficient partly controls how much influence the global best position should have and another random number contributes to the search aspect of the algorithm.

6.3 Global Exploration and Local Exploitation

At this point, it is important to discuss the mechanisms behind the Particle Swarm Optimization algorithm. The Particle Swarm Optimization algorithm finds a solution by initially testing sets of random algorithm parameters and then guiding the direction of the solution search based on the performance of the sets of parameters. There exists a random aspect to the particle velocity formula and this serves the purpose of hopefully improving on previously best fit set of parameters by chance. Once again, the PSO works because as a metaheuristic algorithm it is applied to problems merely looking for a good enough solution or with no absolute best solution.
Bearing what was said in the prior paragraph in mind, a well implemented Particle Swarm Optimization could be generalized to be separable into two different phases. There is an early global exploration phase where particles are allowed to roam more loosely in the search space so that rough optimal solutions can be found and then a local exploitation phase where particles converge into the rough optimal solutions found in the first phase in order to refine and exact the solutions.

In order to achieve the effect of exploration followed by subsequent convergence, the PSO algorithm manipulates three variables from the velocity calculation formula: the inertia weight $\lambda$ and the acceleration coefficients $c_1$ and $c_2$. The variables are adjusted over time in this paper using the following formulas:

$$\lambda_t = (\lambda_F - \lambda_I) \frac{t}{\max(t)} + \lambda_I$$

$$c_{1t} = (c_{1F} - c_{1I}) \frac{t}{\max(t)} + c_{1I}$$

$$c_{2t} = (c_{2F} - c_{2I}) \frac{t}{\max(t)} + c_{2I},$$

where $\lambda_I$, $c_{1I}$, and $c_{2I}$ are the current values of $\lambda$, $c_1$, and $c_2$, $\lambda_I$, $c_{1I}$, and $c_{2I}$ are the initial values of $\lambda$, $c_1$, and $c_2$ and $\lambda_F$, $c_{1F}$, and $c_{2F}$ are the final values of $\lambda$, $c_1$, and $c_2$.

The values of the variables are set to $\lambda_I = 0.9$, $c_{1I} = 2.5$, $c_{2I} = 0.5$, $\lambda_F = 0.4$, $c_{1F} = 0.5$, and $c_{2F} = 2.5$. Adjustment to the value of the variables have a linear relationship with time. It is important to note that the value of $\lambda_F$ subtracted from $\lambda_I$ is negative and $c_{1F}$ and $c_{1I}$ is also negative. This implies the influence of the inertia weight and particle’s best position have the greatest influence on the velocity of the particle in the beginning encouraging exploration for optimal solutions and decreases with time.

Conversely, $c_{2F}$ subtracted from $c_{2I}$ have positive values. This implies the influence of the global best position on the velocity of the particle increases over the runtime of the PSO algorithm. The impact is that particles converge toward the global best position toward the end and directing the search in its vicinity helps refine that optimal solution. This generates the desired Particle Swarm Optimization phases mentioned earlier.
Figure 10. Iterative Flow Map of the PSO Algorithm
6.4 Particle Swarm Optimization Overview

In the prior subsections, individual segments of the Particle Swarm Optimization had been described, such as how to what happens during an iteration of the PSO as well as how to update the velocity of particles. In this subsection, the concepts are all put together to walk through the Particle Swarm Optimization algorithm. Figure 10 features the flow map of a sequential implementation of the algorithm.

Prior to the walkthrough, it is important to bring to the attention that, in this paper, the PSO algorithm is applied to permute through and set the parameters of the ATT algorithm. This means that the PSO algorithm is a layer of code containing the code for the ATT algorithm and during the PSO process described in this subsection instances of the ATT algorithm will be executed many times. Also, the PSO particles are initialized with the ATT parameters initial component rule weights, memory span, review span, reward factor, buy threshold, and sell threshold that will be successively adjusted during PSO iterations.

The Particle Swarm Optimization begins by initializing particles with a random set of parameters. Refer to Figure 8 for the specific parameters as well as their boundaries. The PSO then enters the “PSO iteration” subprocess. In this subprocess, each particle has its fitness evaluated. Each evaluation of fitness is a full instance of the ATT algorithm and step is in green to indicate the significance of the step. The fitness is then checked to see whether it is the particle’s best fit. After every particle has its fitness evaluated, the algorithm checks to see if the global best fit needs to be updated.

If there are more iterations to come, the PSO algorithm undergoes a “velocity update” subprocess prior to starting a new iteration. In this process, the inertia weight and acceleration coefficients are updated using the formulas in subsection 6.3. After, the velocity of each particle is calculated using the formula in subsection 6.2. The particle velocity is added to the particle’s old position to attain the particle’s new position. The PSO algorithm is now ready to start a new iteration.

After all PSO iterations have completed, the final global best fit is the Particle Swarm Optimization’s optimal solution for the problem.
7. SYSTEM

<table>
<thead>
<tr>
<th>Instance Type</th>
<th>vCPU</th>
<th>ECU</th>
<th>Memory (GiB)</th>
<th>SSD Storage (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3</td>
<td>3.75</td>
<td>1 x 4</td>
</tr>
<tr>
<td>m3.large</td>
<td>2</td>
<td>6.5</td>
<td>7.5</td>
<td>1 x 32</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>13</td>
<td>15</td>
<td>2 x 40</td>
</tr>
<tr>
<td>r3.large</td>
<td>2</td>
<td>6.5</td>
<td>15.25</td>
<td>1 x 32</td>
</tr>
<tr>
<td>r3.xlarge</td>
<td>4</td>
<td>13</td>
<td>30.5</td>
<td>1 x 80</td>
</tr>
</tbody>
</table>

Figure 11. Table of AWS Instances

<table>
<thead>
<tr>
<th>Config.</th>
<th>Master</th>
<th>Slaves</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 x m3.large</td>
<td>4 x m3.medium</td>
</tr>
<tr>
<td>B</td>
<td>1 x m3.xlarge</td>
<td>4 x m3.large</td>
</tr>
<tr>
<td>C</td>
<td>1 x r3.xlarge</td>
<td>4 x r3.large</td>
</tr>
</tbody>
</table>

Figure 12. Cluster Configurations

The program of interest in this paper is intended to run on the Amazon Web Service (AWS) Elastic Cloud Compute (EC2) platform. This is important to note as a cluster with an EC2 system architecture has a number of distinctions from traditional computer clusters that may affect parallel programming. Amazon offers virtual machines, which it calls “instances”, for rent and advertises the instances using proprietary specifications rather than traditional ones [28].

Figure 11 shows the specifications of instances offered by Amazon and used in the program relevant to this paper [29]. Figure 12 shows the various cluster configurations used in this paper to test the Parquet and JSON version execution times. The memory specification doubles for each cluster configuration tested.

A “vCPU” stands for virtual CPU and refers to a hyperthread of an Intel Xeon processor. Memory refers to RAM. Storage should be straightforward. The ECU, however, may be unfamiliar to some individuals. “ECU” stands for “EC2 Compute Unit” and is a proprietary metric used by Amazon to advertise the CPU capacity of its instances. One ECU unit is said to provide the processing capability of a 1.0 – 1.2 GHz 2007 Opteron or 2007 Xeon processor.

Aside from cost constraints, there are limitations to the EC2 service [30]. “On-demand” customers, or customers not renting instances for a contiguous month long duration or more, are restricted
to running a maximum of 20 instances of most instance types. More powerful instance types, however, have more stringent limitations.

There are a number of services and features within the AWS ecosystem. EC2 instances can be stopped, started, restarted, and terminated from the AWS EC2 dashboard. Installation of software on instances such as Spark, Hadoop, Pig, etc. may be quickly achieved using AWS Elastic Map Reduce (EMR) for an additional fee on top of EC2 charges [31]. EMR also offers a GUI interface for executing jobs. Metrics like CPUUtilization, DiskReadBytes, etc. of instances may be monitored using AWS Cloudfront.

The program implemented in this paper will not use either EMR. Software will be installed using a Spark script from SSH. This is a much slower process and more prone to failure but ultimately allows for more customization. Cloudwatch will not be heavily used either. Instead, the program will use the Ganglia distributed system monitor tool. Aside from cost, the advantage of Ganglia over Cloudwatch is that it offers a greater number of metrics, most importantly mem_free (RAM).
8. IMPLEMENTATION

8.1 Introduction

The procedure to execute the Particle Swarm Optimization applied on to the ATT algorithm will be described briefly in this section. Prior to the first step, it is assumed that the Apache Spark 1.3.1 pre-build for Hadoop 2.6 packages have been downloaded from https://spark.apache.org/downloads.html and extracted on to the client computer and that there is an existing Amazon Web Service account set up along with the an access key and access key ID. The client computer should be using the Ubuntu Linux operating system.

8.2 Initializing Cluster and Bootstrapping Spark

The initial step to executing the program in this paper is starting up the cluster. Open up the Ubuntu terminal and navigate to the Spark ec2 directory. This directory contains the script “spark-ec2” necessary to initialize a Spark cluster on EC2. In order to do this, input the following commands:

```
gavin@echelon:~$ cd spark-1.3.1-bin-hadoop2.6/ec2

gavin@echelon:~/spark-1.3.1-bin-hadoop2.6/ec2$ export AWS_ACCESS_KEY_ID=<access key id>
gavin@echelon:~/spark-1.3.1-bin-hadoop2.6/ec2$ export AWS_SECRET_ACCESS_KEY=<secret key id>
```

For the first command, the input for the first directory may vary depending on the location of the Spark installation on the computer. In the example, it had been moved to “/”. The second and third commands export environmental variables necessary to initialize AWS. The values are specific to the AWS user and have been redacted in the example for privacy purposes. Personal values for the variables may be attained through AWS. In the EC2 directory should also be a .pem key file downloaded from Amazon.

The next command to input executes the script that will initialize the EC2 cluster and bootstrap Spark on to the nodes.

```
gavin@echelon:~/spark-1.3.1-bin-hadoop2.6/ec2$ ./spark-ec2 -k Spark150818 -i Spark150818.pem -s 4 -r us-west-2 -t m3.large launch spark150824
```
Accessing AWS EC2 service requires the keypair name preceded by the flag “-k” and then the user’s secret key file preceded by the flag “-i”. The flag “-s” determines the number of slaves on the cluster and the flag “-r” determine the region the cluster will be located. Note that the number of slaves requested must be within the limitations expressed earlier. The “-t” flag determines the type of instance that the cluster will use. Instance specifications may be found in Figure 11. Note that there exists an hourly fee for rental of EC2 instances.

After executing the “spark-ec2” script, it may take as much as 10 – 20 minutes for the computer to start the AWS instances and install all the relevant Spark files. When the process is complete, the output on the terminal should look like this:

![Successful Cluster Initialization](image)

Two urls are output that help monitor the cluster. The first URL has the general cluster monitor page. This shows basic information such as the state of slaves, the jobs submitted to slaves, and the progress of submitted jobs. The second url is for the Ganglia distributed system monitor. Ganglia offers a variety of detailed cluster information regarding CPU utilization, memory utilization, and network performance. A final source of cluster monitoring is through Amazon’s Cloudwatch service. This has overlap and may be considered an alternative to Ganglia.
8.3 Connect to the Master Node

The next step is to connect to the master node. This is the node where jobs are submitted and manages activity on the cluster. In order to achieve this, log on to AWS and navigate to the EC2 dashboard. Select “instances” from the left menu bar. Right click on the master node of the table and select “connect”

A window will pop up and it should contain a command to input into the terminal. For example:

```
ssh -i "Spark150818.pem" root@52.24.133.213
```

This command is fairly straightforward. Once again, after the “-i” flag, input the file location for the secret key. Successful connection to the master node results in the terminal screen in Figure 13.

![Figure 14. Successful Connection to Node](image)

Once connected to the node, project files must be transferred to the cluster. Packaging a python program for PySpark is simple. Merely compress all source code into a .zip file with the exception of the file containing the main method. In addition to the source files, data files will also need to be transferred. In order to transfer the project files on to the master, use the SCP protocol. In a terminal on the client computer, input something similar to the following command for each file that needs to be transferred.

```
scp -i "Spark150818.pem" PS00209.zip ec2-user@52.32.89.92:
```
As always, “-i” is the key file. PSO0209.zip is the name of the file to be transferred, and ec2-user@52.32.89.92: is the address of the master node. The ip address could be attained through selecting “connect” in the EC2 dashboard like in the prior step except the ip address must be preceded by “ec2-user” instead of “local”. The transferred file should be found in the “home/ec2-user/” directory of the master node.

After all relevant files have been transferred, data files need to be uploaded to HDFS (Hadoop file system). The command for accomplishing this on an EC2 instance is:

```
root@ip-172-31-0-228 /]$ ~/ephemeral-hdfs/bin/hadoop fs -put
SP500Industry.json SP500Industry.json
```

The file immediately following the “-put” flag is the location of the file to be transferred and the file after that is the location for the file to transfer to on HDFS. For the program of interest in this project, there are 3 separate datafiles that need to be transferred. These are “DE01 market dates.json”, “DE01 Prices.json”, and “DE01 Symbols.json”.

8.4 Executing the Program

At this point, all the necessary project files should be on the master node and all data files should be uploaded on to HDFS. The only step that remains is executing the program. In order to execute the program, navigate to the “~/.spark/bin/” directory on the master node. The directory contains the “spark-submit” script responsible for executing Spark programs. Input a command similar to below:

```
./spark-submit --py-files /PSO0410.zip /PSO.py
```

The file “/PSO0410.zip” after the “--py-files” flag contains all the program dependencies. “/PSO.py” contains the main method. This will execute the program of interest with default settings. There exist alternate settings to the program. These are selected by passing additional arguments to “spark-submit”.

46
The primary program option is the choice of parallelization or sequential execution method. Selecting a different execution method requires the passing of an additional integer argument at the end. An example of a command is the following:

```
./spark-submit --py-files /PSO0410.zip /PSO.py 6
```

The integers 1, 2, 3, 4, and 6 are valid choices. 6 is the default execution method. This executes the program in parallel. 3 is the recommended default sequential execution method.

After the Particle Swarm Optimization has applied to the trading algorithm, it is necessary to test the resulting particle. The completed execution of the PSO portion of the program will output a “optimal_particle.att” file. Pass the file in as an argument to test the optimal particle.

```
./spark-submit --py-files /PSO0410.zip /PSO.py 6 optimalParticle.att
```

After execution of the program on the cluster, one may desire to save the results. Use the SCP protocol to transfer files generated from the program back to the client computer. Prior to inputting the command, move the program generated files into the /home/ec2-user directory. Then, on the client computer, input the following command.

```
scp -i "Spark150818.pem" ec2-user@52.10.82.10:optimal_particle.att.
```

Following “-i” is the secret key and the argument after that is the IP address of the master node along with the name of the file to be transferred. The last argument, the “.”, is the directory on the client computer to transfer the file to.
9. FILE FORMAT PERFORMANCE

9.1 Execution Results

The execution time of various successfully terminated test cases is graphed in Figure 15. The bars are grouped into three clusters representing the various cluster configurations used during their time of testing. Once again, Configuration A was tested using 4 m3.medium slaves and a m3.large master instance. Configuration B was tested using 4 m3.large slaves and a m3.xlarge master instance. Configuration C was tested with 4 r3.large slaves and a r3.xlarge master instance. The blue bars represent the result for the Parquet version of the program, the orange or red bars represents the result for the JSON version of the program, and the yellow bar represents a version of the program using no data file and only the in-memory SparkSQL DataFrame data structure.

As may be observed, the Parquet version outperformed the JSON version of the program on Configuration B. As a matter of fact, the Parquet version of the program was able to run on Configuration A with notably inferior specifications to Configuration B while the JSON version of the program was unable to complete execution. Instead, it threw a “java.lang.OutOfMemoryError: Java heap space” exception. On Configuration C, with the highest specifications, however, JSON markedly outperforms the Parquet variant. This is a noteworthy observation. Under certain conditions, JSON outperforms and others Parquet outperforms.
It naturally follows that one would inquire “why” this observation holds true. With the assistance of the Ganglia Monitoring System, there are detailed cluster metrics to diagnose and explain the observation. Given that the JSON version of a program failed with an Out of Memory error, it makes sense to check the memory status of the cluster. Figure 16 displays the average free memory reading for each of the text executions at any given time.

Note that not all instances on the graph have the same amount of starting memory. m3.large instances have twice as much memory as m3.medium instances, m3.xlarge instances have twice as much memory as m3.large instances, r3.large instances have the same amount of memory as m3.xlarge memory instances, and r3.xlarge instances have twice as much memory as m3.xlarge instances. So free memory readings will be disproportionately high for the higher specification Configuration C because, for example, 50% free memory on one of its slaves is greater than 50% free memory on one of the slaves on Configuration B.

Bearing the disproportionate specifications in mind, there still exists an unusually low reading for the Configuration A and B clusters relative to the Configuration C cluster. Both the Parquet A and JSON B
executions have little free memory throughout the execution of the program. Parquet B has free memory but only on the master node. For Configuration C, all versions of the program have a generous amount of memory. Parquet C has less free memory on slaves than JSON C and In-Memory C, however, while In-Memory C has less free memory on the master than Parquet C and JSON C.

What this shows is that the Parquet version of the program is better suited for scaling among memory configurations. In a limited memory environment like Configuration A, Parquet nodes can have almost no free memory without crashing unlike the JSON variant of the program while for a generous memory environment like Configuration C the Parquet version of the program is able to utilize more memory usage than the JSON version of the program.

Now that it has been established that both Parquet B and JSON B (as well as Parquet A) are running in limited memory environments, the question becomes about why the two versions of the program using different file systems behave differently. Why would one run faster in a low memory environment than the other? By perusing Ganglia metrics, here are some other notable graphs.

![Figure 17. Number of “Get Block Location” Operations](image)

Figure 17 shows the number of data file system “Get Block Location” operations used on average during a given reading. Note that the count is less than 1 because during most readings there are no “Get Block Location” operations in this program. In a distributed file system, a dataset is split up into “blocks”
and distributed across the cluster. Each slave node, therefore, contains a portion of the dataset. In this paper, the primary dataset would be that in the JSON and parquet files. So the “Get Block Location” operation is used by the master node to locate data in the cluster prior to querying it from a slave.

Figure 18. Amount of Data Retrieved From Each Slave

Once again, JSON B uses an exceptionally high number of “Read Block” operations. What this shows is that JSON B needs to access the dataset much more than other versions of the program. The other versions of the program have relatively similar counts of “Read Block” operations. JSON C has slightly more and In-Memory has none, as expected, because it does not use a data file system.

Figure 18 shows the data file system bytes read from each of the slaves, or data nodes. The readings complement the previous figure and make sense because each “Read Block” operations naturally comes with a number of bytes so the greater number of “Read Block” operations the greater number of bytes that should be read.

The amount of bytes read is overwhelmingly higher for the JSON B test case. Parquet A, Parquet B, Parquet C test cases have more bytes read than JSON C but still trailing JSON B. The In-Memory C version of the program has the lowest bytes read. Once again, it does not use the distributed file system so
its readings should be 0. One noteworthy observation is that while the Parquet A, B, and C versions of the program have less Read Block operations than the JSON C version of the program, the JSON C version of the program has less bytes read. This implies that each Read Block operation for the Parquet versions of the program reads more bytes than the JSON version of the program. This makes sense given knowledge of the nature of querying columnar databases.

Figure 19. Average JVM Garbage Collection Reading

In the final graph presented in this subsection, Figure 19 depicts the average number of JVM garbage collection instances at any given moment. Note that garbage collection does not occur on a regular basis so averaging out many 0 readings results in the low average count for the clusters. Similar to the other graphs, JSON B is shown garbage collecting the greatest number of times. Parquet A and Parquet B show similar amounts of garbage collecting with Parquet A garbage collecting more on slaves while Parquet B garbage collecting more on the master. For the Parquet C, JSON C, and In-Memory C versions of the program, all have relatively close amounts of garbage collection.
9.3 Analysis

Given the Ganglia Metrics presented in the prior subsection, potential reasons for the observation of Parquet B outperforming JSON B while JSON C outperforms Parquet C in terms of execution time may be drawn.

Based on the “Memory Free” bar graph, there exists information that a meaningful difference between JSON B as well as Parquet B and JSON C as well as Parquet C is that one of the clusters lacks free memory while the other has a sufficient amount of free memory. Furthermore, the “GetBlockLocations” graph establishes that there are many more file system queries from the JSON B than all other test cases. Therefore, it makes sense to identify a location in the program code affecting the number of file queries.

This excerpt of code is of particular interest:

```python
# Query for multiple days worth of data
trans_hundred = DS.sqlc.sql("SELECT * FROM parquetTable
WHERE day >= " + str(q_day) + " AND day < " + str(last_day)).cache()

t = 0
while t < num_queried:
    # Query for a day's worth of data
    trans_day = trans_hundred.where(trans_hundred.day ==
    day_off).cache()

    # Other code
    trans_day.unpersist()
    t = t + 1

trans_hundred.unpersist()
```

What the above code accomplishes is that it queries for (last_day - q_day) worth of data, stores it in the trans_hundred variable, and caches the query. After, there is a while loop where a query is made from the result of trans_hundred for a single day’s worth of data and that is stored in a trans_day variable and cached as well. The trans_day data is then manipulated in some “other code” within the while loop, which is not presented here for the sake of brevity.

The code for JSON and Parquet versions of the program are identical except for three lines of code (the number varying depending on how one counts lines of code) where the program initializes the JSON
and Parquet files so this controls for and rules out any remarkable differences in the project code that may affect performance.

Furthermore, the code excerpt is the only non-single use instance in the JSON and Parquet versions of the program using the `.sql()` function, one of the Spark SQL API functions that queries the dataset. That is there does exist other `.sql()` functions within the program; but, these are only executed once per program execution on very small datasets so they should not have any notable impact on program execution time. The excerpt also contains a single `.where()` function, which also queries the dataset and there exists one more `.where()` query not presented in the excerpt but is nested within the while loop containing `trans_day`. This code block is executed hundreds of times during a program execution.

The `.sql()` and the `.where()` functions are the only methods that have the ability to query the data files in this program so the specific source of the GetBlockLocations operation could be isolated to this code snippet. Looking closely at the code snippet, the line of code that may affect the number of times the file system may be accessed is `.cache()`.

What `.cache()` does is it stores the query in memory and reuses it for future functions. Without `.cache()`, file system accesses from `.sql()` and `.where()` would be executed an identical number of times for each JSON and Parquet version of the program. `.cache()` reduces the need to execute a `.sql()` and `.where()`. The problem is that in the Configuration B tests cases, Parquet B and JSON B, there is very little free memory to `.cache()` the queries. If memory used for caching is needed for other parts of the program, the cache will be cleared. In the event memory used for caching is cleared, then the cached data will need to be recalculated the next time it is used.

Based on the `free_memory` graph, this is very likely what is being observed between Parquet B and JSON B. Clearing cache requires garbage collection, which JSON B does a relatively large amount of in the JVM Garbage Collection graph. When the amount of free memory increases, the number of GetBlockLocation operations are much more similar as can be observed from the DFS NameNode GetBlockLocation graph comparing Parquet C and JSON C test cases. This suggests the reason JSON B underperforms in terms of execution time is that it does not have memory for caching while Parquet B does.
Given this information, what are some reasons that JSON file may require more memory in the query process than Parquet file? The JSON file is much larger than the Parquet file. The JSON file is 2.9 GB while the Parquet file is only 101 MB. This is considerably smaller than even the JSON file compressed into a .zip file, which is 169 MB. The JSON file format includes redundant attribute tags while the encoder is focused on the challenges of processing nested schemas rather than the flat schema used in this paper.

The small Parquet file size is likely due to the efficient representation of nested schema in Parquet as well as relevant encoding algorithms such as Run Length Encoding that are well suited for the dataset. The addition of statistics on columns further filters the amount of data that needs to be processed by the decoder. The combination of factors reduces the memory footprint of the Parquet decoding process.

From the comparison of JSON C and Parquet C test cases, JSON is still shown to be capable of outperforming Parquet significantly in a more memory sufficient configuration, however. From the NameNode GetBlocksLocation and DataNode Bytes Read graphs, it is known that Parquet file format could make fewer file system calls than JSON while still requiring more data to be retrieved from the DataNode.

This observation may be explained by the differences in column-oriented and row-oriented datasets. Because Parquet is a column-oriented dataset yet the queries in this algorithm ask for all values in a row, many columns must be searched to attain one row of data. In contrast, attaining a row of data could be accomplished much more efficiently with a row-oriented dataset such as JSON where a single operation can attain all the values in a row.
10. TRADING ALGORITHM PROFITABILITY

10.1 Stock Symbol Datasets

The trading algorithm was applied to two sectors of companies categorized under “Basic Materials” and “Retail-Wholesale” traded on either the NYSE or NASDAQ with history extending back to 1990. The list of the “Basic Materials” symbols is represented in the table in Figure A9 and the list of “Retail-Whole” symbols is represented in the table in Figure A10.

The reason the “Basic Materials” sector of stocks was chosen to be evaluated is because it is speculated to be a less volatile sector with trading patterns that can extend for a longer period of time versus many growth stocks in technology or biotechnology but that have a tendency to bankrupt, be bought out, and experience seemingly arbitrary boom-bust cycles. “Retail-wholesale” was selected as a sector of interest for similar reasons but as a slightly more dynamic alternative.

The set of stocks using industries was chosen over popular indices such as the S&P 500 due to the bias of selecting presently successful companies. Because companies regularly appear and disappear from the indices based on their success or failure, analyzing the set of S&P 500 stocks at any point in time would be prone to selecting only the most relevant and stable stocks during that period. Also, companies in the same sector are more likely to adhere to the same trading patterns than a basket of randomly unrelated stocks.

The year 1990 was specifically chosen as the start date of the dataset after considering a balance of time period and the number of stocks available to analyze from the time period. The first half of the dates from 1990 to the current year 2015 are reserved for training the algorithm and the second half of the dates are used for testing the result.

10.2 Control Groups

Prior to discussing the results of the trading algorithm, it is important to discuss the control groups to make a fair comparison to the algorithm performance. Stock prices have a tendency to increase in time;
however, now all stock prices from every company increases and there is a tendency for certain sectors or industries to outperform others. In this paper, the performance of solely the set of stocks of interest were calculated for the testing period.

In Figure A9, each “Basic Materials” symbol is listed with their total percentage change in value from 2002 to 2015. The average change in value was 323%. That is an annualized rate of return of around 10.27% during that time period. In Figure A10, each “Wholesale-Retail” symbol is listed with their total percentage change in value from the same time period. The average change in value was 238%. That is an annualized rate of return of around 7.49% during that time period.

<table>
<thead>
<tr>
<th>Index</th>
<th>Rule</th>
<th>Parameters</th>
<th>Annual Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4][76]</td>
<td>OBV Average</td>
<td>(125, 100)</td>
<td>27.34%</td>
</tr>
<tr>
<td>[0][14]</td>
<td>Moving Average</td>
<td>(30, 10)</td>
<td>26.66%</td>
</tr>
<tr>
<td>[0][9]</td>
<td>Moving Average</td>
<td>(25, 10)</td>
<td>26.52%</td>
</tr>
<tr>
<td>[0][5]</td>
<td>Moving Average</td>
<td>(20, 10)</td>
<td>26.33%</td>
</tr>
<tr>
<td>[4][130]</td>
<td>OBV Average</td>
<td>(250, 100)</td>
<td>25.89%</td>
</tr>
<tr>
<td>[4][101]</td>
<td>OBV Average</td>
<td>(175, 100)</td>
<td>25.63%</td>
</tr>
<tr>
<td>[0][2]</td>
<td>Moving Average</td>
<td>(15, 10)</td>
<td>24.93%</td>
</tr>
<tr>
<td>[4][13]</td>
<td>OBV Average</td>
<td>(20, 15)</td>
<td>24.76%</td>
</tr>
<tr>
<td>[4][103]</td>
<td>OBV Average</td>
<td>(175, 150)</td>
<td>24.69%</td>
</tr>
<tr>
<td>[4][88]</td>
<td>OBV Average</td>
<td>(150, 100)</td>
<td>24.22%</td>
</tr>
<tr>
<td>[0][43]</td>
<td>Moving Average</td>
<td>(100, 5)</td>
<td>22.83%</td>
</tr>
<tr>
<td>[0][16]</td>
<td>Moving Average</td>
<td>(30, 20)</td>
<td>22.66%</td>
</tr>
</tbody>
</table>

Figure 20. Top Performing Component Rules “Basic Materials”

Aside from comparison with the average return of the set of stocks, the return of each of the component rules individually is also of interest. Over the 12 year testing period for the “Basic Materials” sector, the top performing component rule, OBVA with the parameters (125, 100), was able to attain an average of 27.34% annual gain. This compares very favorably with the average of 10.27% annual gain of the set of stocks. The top rules for the 2002 to 2015 time period using “Basic Material” sector stocks were all variations of the On Balance Volume Average and Moving Average rules. The average return of all rules was 15.79% while the worst performing rule resulted in a 6.95% annual return.
Statistics for performance of component rules for the “Retail-Wholesale” sector are also available. The top 12 performing rules were all Trading Range Breakout variants with different parameters and with annual gains of 56.81%. The duplicate results despite different parameters are not particularly surprising due to the nature of the Trading Range Breakout rules and its tendency to send a “buy” signal as long as a particular stock is trending upward for the period or downward for the period. The next most profitable component rule that was not a Trading Range Breakout rule was an On Balance Volume variant with 39.29% annual return. The average return of a rule was 18.34% so performance is not drastically different than for the “Basic Materials” sector despite the more extreme performance of a few particular rules.

<table>
<thead>
<tr>
<th>Index</th>
<th>Rule</th>
<th>Parameters</th>
<th>Annual Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1][7]</td>
<td>Trading Range Breakout</td>
<td>(45)</td>
<td>56.88%</td>
</tr>
<tr>
<td>[1][6]</td>
<td>Trading Range Breakout</td>
<td>(40)</td>
<td>56.87%</td>
</tr>
<tr>
<td>[1][16]</td>
<td>Trading Range Breakout</td>
<td>(150)</td>
<td>56.85%</td>
</tr>
<tr>
<td>[1][17]</td>
<td>Trading Range Breakout</td>
<td>(175)</td>
<td>56.85%</td>
</tr>
<tr>
<td>[1][8]</td>
<td>Trading Range Breakout</td>
<td>(50)</td>
<td>56.84%</td>
</tr>
<tr>
<td>[1][4]</td>
<td>Trading Range Breakout</td>
<td>(30)</td>
<td>56.82%</td>
</tr>
<tr>
<td>[1][9]</td>
<td>Trading Range Breakout</td>
<td>(60)</td>
<td>56.82%</td>
</tr>
<tr>
<td>[1][5]</td>
<td>Trading Range Breakout</td>
<td>(35)</td>
<td>56.82%</td>
</tr>
<tr>
<td>[1][10]</td>
<td>Trading Range Breakout</td>
<td>(70)</td>
<td>56.81%</td>
</tr>
<tr>
<td>[1][11]</td>
<td>Trading Range Breakout</td>
<td>(75)</td>
<td>56.81%</td>
</tr>
<tr>
<td>[1][12]</td>
<td>Trading Range Breakout</td>
<td>(80)</td>
<td>56.81%</td>
</tr>
<tr>
<td>[1][13]</td>
<td>Trading Range Breakout</td>
<td>(90)</td>
<td>56.81%</td>
</tr>
</tbody>
</table>

Figure 21. Top Performing Component Rules “Retail-Wholesale”

10.3 ATT Performance

In this section, the performance of the ATT algorithm is discussed. Note that not all particle parameters are specified as each particle contains a starting or initial weight for each component rule and it would be impractical to list a thousand weights. The particles are identified by their parameters for memory span, review span, buy signal, sell signal, and reward factor.
Particle | Fit
---|---
‘232 114 0.1199 -0.2602 0.7413’ | 5.06480779397593
‘222 135 0.0015 -0.2583 0.7622 ’ | 4.87649620613494
‘241 20 0.0726 -0.3332 0.1664 ’ | 4.71581057225631
‘290 20 0.1361 -0.3333 0.1807’ | 4.69208665281910

Figure 22. Top Fitting Particles “Basic Materials”

Figure 22 shows some of the best performing particles for the “basic materials” sector. Performance of the ATT algorithm is slightly below that of the average component rule at 14.46% annualized return. There exist a strong pattern of resistant sell signal thresholds with two close to the boundary of -0.3333. Two of the particles have the minimal review span opting for frequent weight updates while the other two opt for less frequent rule weight updates.

Particle | Fit
---|---
‘212 141 0.3292 -0.3333 0.1432’ | 4.35159476386927
‘222 113 0.3251 -0.3333 0.1275’ | 4.242723001759044
‘241 150 0.3371 -0.3333 0.1519’ | 4.135133051382953
‘233 137 0.2941 -0.3333 0.1443 ’ | 3.919557023480024

Figure 23. Top Fitting Particles “Retail-Wholesale”

Figure 23 shows some of the best particles discovered suited for the “retail-wholesale” data set. Performance for the ATT algorithm lagged that of the component rule average at 13.03% annual rate but still beat the general index at a whole which increased 7.49%. Particle parameters clustered fairly closely particularly with the -0.3333 on the sell threshold boundary indicating a consensus to avoid selling shares.
11. CONCLUSION

The introduction of cloud based cluster computing has made the execution of many demanding computer programs possible. In this paper, a Particle Swarm Optimization was implemented on Apache Spark to permute through and set parameters of a stock trading algorithm. The program made heavy use of SQL queries on JSON and Parquet datasets and the program execution time on different clusters was compared to observe performance differences between the two file formats.

It was discovered that in a low memory environment that the Parquet version of the program outperformed the JSON version of the program; but, however, in a memory sufficient environment that the JSON version of the program outperformed the Parquet version of the program. Investigating the issue deeper, it was discovered that the low memory JSON version of the program made many more file system queries than other successfully executed versions of the program.

Based on these observations, it may be concluded that both Parquet and JSON file formats have their use specific use cases. JSON may be better for systems with lots of memory while Parquet may be better for cases when the system lacks lots of free memory and the JSON file size is relatively large. This is an interesting finding because both file formats underwent unideal use cases. The JSON decoder was made to process a flat schema file while the Parquet decoder was forced to execute queries unideal for a row-oriented dataset.
REFERENCES


[22] "Relative Strength Index (RSI) - Fidelity." Relative Strength Index (RSI) - Fidelity. Web. 05 May 2016.


[28] "EC2 Instance Pricing – Amazon Web Services (AWS)." Amazon Web Services, Inc. Web. 30 Nov. 2015.

[29] "EC2 Instance Types – Amazon Web Services (AWS)." Amazon Web Services, Inc. Web. 30 Nov. 2015.


APPENDIX

Appendix contains Figures illustrating the technical analysis trading rules used in this paper as well as more detailed explanations about their implementation.

Figure 24. Moving Average Diagram

In Moving Averages (MA), there are two averages of stock prices over two moving windows of \( n_l \) days and \( n_s \) days as follows:

\[
\text{Avg}_{t,n_l} = \frac{1}{n_l} \sum_{i=t-n_l+1}^{t} p_i, \quad \text{Avg}_{t,n_s} = \frac{1}{n_s} \sum_{i=t-n_s+1}^{t} p_i
\]

Where \( n_l > n_s \), \( t \) is the current trading day and \( p_i \) is the close stock price on day \( i \).

MA initiates a buy signal if \( \text{Avg}_{t,n_s} \) is above \( \text{Avg}_{t,n_l} \) and a sell signal if \( \text{Avg}_{t,n_l} \) is above \( \text{Avg}_{t,n_s} \).

The above figure displays the simple 20 and simple 100 moving average lines for a stock. It shows how the lines intersect every so often generating either a buy or a sell signal. The set of parameters tested are \( n_l = [15, 20, 25, 30, 40, 50, 75, 100, 125, 150, 175, 200, 250] \) and \( n_s = [2, 5, 10, 15, 20, 25, 30, 40, 50, 75, 100, 125, 150, 175, 200] \).
Trading Range Breakout (TRB) calculates the highest and lowest close price of the past n days as follows:

\[ H_{t,n} = \max(p_{t-1}, p_{t-2}, \ldots, p_{t-n}) \]
\[ L_{t,n} = \min(p_{t-1}, p_{t-2}, \ldots, p_{t-n}) \]

A buy signal is generated when \( p_t > H_{t,n} \) and a sell signal is generated when \( p_t < L_{t,n} \).

The above figure illustrates practice of the Trading Range Breakout rule from the period of December 26, 2014 and June 26, 2015. The parameters used for testing are \( n = \{10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 75, 80, 90, 100, 125, 150, 175, 200, 250\} \).
Bollinger bands is a volatility indicator that considers the fluctuations of stock prices. For trading day $t$, BBs calculate an $n$-day moving average of past close prices $\overline{Avg}_{t,n}$, which is the middle band.

An upper band and lower band are $k$ times standard deviations above and below from the middle band, respectively. The upper and lower bands form a price channel. Essentially, a buy signal is generated when the close price fall below the lower band, and a sell signal is generated when the close price is above the upper band.

The above figure exemplifies the drawing of two Bollinger Bands using 20 day moving averages and a standard deviation of 2. Stock price dips and rises below and above the band could be observed. The parameters used for testing are $n = [10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 75, 80, 90, 100, 125, 150, 175, 200, 250]$ and standard deviation $= [1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5]$. 
The Relative Strength Index is a very popular oversold/overbought indicator which measures the velocity and magnitude of directional price movements. An n-day RSI of a trading day $t$ is calculated as follows:

$$RSI = 100 - \frac{100}{1 + \frac{\text{Avg.} \text{U}}{\text{Avg.} \text{D}}}$$

Where Avg. U is the average of all up price moves and Avg. D is the average of all down price moves in an n-day period. The RSI oscillates between 0 and 100 and there are many ways to generate signals with RSI depending on threshold. RSI is an oscillator and emits a null signal when there exists a trend. Refer to trend identification below.

The secondary graph in the above figure illustrates some commonly calculated RSI values while the yellow lines labeled 70.00 and 30.00 indicate the buy and sell thresholds. The parameters used for testing are $n = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]$, buy threshold = [80, 75, 70], and sell threshold = [20, 25, 30].
The Stochastic Oscillator is another momentum indicator which is also very popular with traders. The calculation of STO involves the high, low, and close prices in an n-day lock-back period. The STO on trading day t is given as follows:

$$STO = 100 \left( \frac{p_t - p_{\text{lowest}}}{p_{\text{highest}} - p_{\text{lowest}}} \right)$$

Where \( p_{\text{lowest}} \) and \( p_{\text{highest}} \) are the lowest and highest high prices in the look-back period, respectively. Similar to RSI, STO is between 0 and 100.

The STO is usually smoothed with an m-day moving average to form the fast %K. The fast %K is then smoothed with another m-day moving average to form the fast %D. Various kinds of signal generation methods are proposed and used in practice. Typical method includes an oversold and overbought thresholds. Buy signal is generated when fast %D line is below the oversold threshold and accompanied with that the fast %K line rises above the fast %D line.

The parameters used for testing are \( n = [5, 10, 15, 20, 25, 50, 75, 100, 125, 150, 200, 250] \), \( m = [3, 7, 11] \), overbought threshold = \([80, 85, 90]\), and oversold threshold = \([10, 15, 20]\).
The Moving Average Convergence/Divergence (MACD) indicator is a combination of two exponential moving average (EMA) of close price. An n-day EMA on day $t$ is calculated as follows:

$$E.\text{Avg}_{t,n} = \alpha p_t + (1 - \alpha)E.\text{Avg}_{t-1,n}$$

With $\alpha = \frac{2}{1+n}$.

Similar to MA, there is a long period exponential average $E.\text{Avg}_{t,nl}$ and a short-period exponential moving average $E.\text{Avg}_{t,ns}$. The MACD is the difference between the two exponential averages, which is given by:

$$\text{MACD} = E.\text{Avg}_{t,ns} - E.\text{Avg}_{t,nl}$$

Another m-day EMA of the MACD which is called the MACD signal, is calculated for the signal generation. A buy signal is emitted when the MACD lines crosses above the MACD signal line from below, and a sell signal is emitted when the MACD line crosses below the MACD signal line from the above. MACD is an oscillator and emits a null signal when there exists a trend. Refer to trend identification below. The parameters used for testing are $nl = [20, 25, 30, 40, 50, 100]$, $ns = [5, 10, 15, 20, 30, 40]$, and $m = [7, 9, 11, 13, 15]$. 
As a volume based trading rule, OBVA is the same as MA except that it calculates MA with stock volume instead of stock price. The parameters used for testing are larger window = \([5, 10, 15, 20, 25, 30, 40, 50, 75, 100, 125, 150, 175, 200, 250]\) and smaller window = \([1, 2, 5, 10, 15, 20, 25, 30, 40, 50, 75, 100, 125, 150, 175, 200]\).
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>-0.654090505</td>
</tr>
<tr>
<td>ABX</td>
<td>-0.570114964</td>
</tr>
<tr>
<td>AEM</td>
<td>1.771725246</td>
</tr>
<tr>
<td>APD</td>
<td>3.009902146</td>
</tr>
<tr>
<td>ARG</td>
<td>4.949680942</td>
</tr>
<tr>
<td>ASH</td>
<td>3.082190146</td>
</tr>
<tr>
<td>AVD</td>
<td>0.808240736</td>
</tr>
<tr>
<td>AXLL</td>
<td>-0.966395017</td>
</tr>
<tr>
<td>BHP</td>
<td>1.927036983</td>
</tr>
<tr>
<td>CBT</td>
<td>0.695161344</td>
</tr>
<tr>
<td>CLF</td>
<td>-0.279823209</td>
</tr>
<tr>
<td>CMC</td>
<td>2.102183085</td>
</tr>
<tr>
<td>CRS</td>
<td>2.185623419</td>
</tr>
<tr>
<td>DD</td>
<td>1.530640163</td>
</tr>
<tr>
<td>DOW</td>
<td>1.086241707</td>
</tr>
<tr>
<td>ECL</td>
<td>4.452677236</td>
</tr>
<tr>
<td>FMC</td>
<td>4.934714558</td>
</tr>
<tr>
<td>FOE</td>
<td>-0.332161237</td>
</tr>
<tr>
<td>GLT</td>
<td>1.196243704</td>
</tr>
<tr>
<td>GRA</td>
<td>35.74637681</td>
</tr>
<tr>
<td>HL</td>
<td>-0.689825345</td>
</tr>
<tr>
<td>IFF</td>
<td>3.640405717</td>
</tr>
<tr>
<td>IP</td>
<td>0.657273011</td>
</tr>
<tr>
<td>MOS</td>
<td>4.437623586</td>
</tr>
<tr>
<td>MSB</td>
<td>3.373060973</td>
</tr>
<tr>
<td>MTRN</td>
<td>1.506062732</td>
</tr>
<tr>
<td>MUX</td>
<td>0.129460756</td>
</tr>
<tr>
<td>NEM</td>
<td>-0.481369456</td>
</tr>
<tr>
<td>NEU</td>
<td>21.78720507</td>
</tr>
<tr>
<td>NL</td>
<td>-0.326952521</td>
</tr>
<tr>
<td>NUE</td>
<td>3.444789018</td>
</tr>
<tr>
<td>OLN</td>
<td>0.705696771</td>
</tr>
<tr>
<td>PPG</td>
<td>3.912757104</td>
</tr>
<tr>
<td>SXT</td>
<td>3.747944101</td>
</tr>
<tr>
<td>VHI</td>
<td>0.691537166</td>
</tr>
</tbody>
</table>

Figure 31. “Basic Materials” Sector and Change in Value 2002 - 2015
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAN</td>
<td>3.039703</td>
</tr>
<tr>
<td>BBY</td>
<td>0.076897</td>
</tr>
<tr>
<td>BIG</td>
<td>2.329539</td>
</tr>
<tr>
<td>CATO</td>
<td>3.195029</td>
</tr>
<tr>
<td>CVS</td>
<td>5.349331</td>
</tr>
<tr>
<td>DDS</td>
<td>5.198833</td>
</tr>
<tr>
<td>EAT</td>
<td>1.796708</td>
</tr>
<tr>
<td>FL</td>
<td>3.456784</td>
</tr>
<tr>
<td>GCO</td>
<td>2.904075</td>
</tr>
<tr>
<td>GPS</td>
<td>0.544258</td>
</tr>
<tr>
<td>HD</td>
<td>3.475672</td>
</tr>
<tr>
<td>JCP</td>
<td>-0.56941</td>
</tr>
<tr>
<td>JWN</td>
<td>3.968881</td>
</tr>
<tr>
<td>KATE</td>
<td>-0.44603</td>
</tr>
<tr>
<td>KR</td>
<td>3.674597</td>
</tr>
<tr>
<td>LB</td>
<td>9.098245</td>
</tr>
<tr>
<td>LOW</td>
<td>1.93179</td>
</tr>
<tr>
<td>LUB</td>
<td>0.34</td>
</tr>
<tr>
<td>MCD</td>
<td>5.155158</td>
</tr>
<tr>
<td>PBY</td>
<td>-0.12674</td>
</tr>
<tr>
<td>PIR</td>
<td>-0.66261</td>
</tr>
<tr>
<td>RAD</td>
<td>0.302479</td>
</tr>
<tr>
<td>TGT</td>
<td>1.397858</td>
</tr>
<tr>
<td>TIF</td>
<td>1.192045</td>
</tr>
<tr>
<td>TJX</td>
<td>6.502173</td>
</tr>
<tr>
<td>WMK</td>
<td>0.738754</td>
</tr>
<tr>
<td>WMT</td>
<td>0.311388</td>
</tr>
</tbody>
</table>

Figure 32. “Retail-Wholesale” Sector and Change in Value 2002 - 2015