A Survey of Big Data Methods, Assessments, and Approaches

Sotera Defense Solutions

November 2012

Abstract

The purpose of this survey was to determine how to best match technology platforms to classes of analytics. Several prominent Big Data technology stacks were compared using a set of diverse use cases that commonly occur in real world data. We perform a set of quantitative benchmarks to compare the technology stacks and also include a qualitative assessments of lessons learned.

1 Introduction

For XDATA, we set out to produce a series of benchmarks to provide quantitative comparisons between different "Big Data" technology approaches. This evaluation not only includes an analysis of the runtime associated with a particular approach, but also considers the code complexity and overall development time of said code. Given that analytics span a wide range of capabilities, the evaluation must account for technologies that are purpose built to accommodate specialized models and analytic approaches. In other words, we must be careful not to use one tool for every problem or else we run the risk of "forcing a square peg into a round hole" - to use a phrase from British novelist Edward Lytton.

In this paper we survey several popular "Big Data" technologies and paired them with a handful of analytical case studies. The technologies that we evaluated ranged from Hadoop MapReduce approaches, to RAM based platforms, to Bulk Synchronous Parallel techniques. Specifically, the following technologies were leveraged in our current work:

- Hadoop Streaming (MapReduce)
- Apache Hive (MapReduce)
- Cascalog (MapReduce)
- Mahout (MapReduce)
- R (Radoop / RHadoop / rPy / R - MapReduce)
- Giraph (Bulk Synchronous Parallel)
- Spark (in-memory cluster computing)

Given the technologies above, for each case study we attempted to match the most appropriate technology stack to the problem and provide both qualitative and quantitative evaluation as to their appropriate usage patterns. Before describing each of these approaches, it is important to note that these are initial prototypes and that full optimization may not have been attained in each case. However, reasonable effort was exercised in each case to optimize each approach wherever possible.

As a baseline, we chose four disparate analytic models for analysis and implementation. Each of these analytics served as an example implementation strategy and paradigm. The use case analytic models chosen for evaluation are as follows:

- **N^2 Calculation** – For the ultimate test of scaling behavior, we chose an analytic model that innately does not scale. Problems that are $O(N^2)$ are certainly a challenge, especially due to data locality issues. For this experiment, we calculated the correlation co-efficient of a set of time series vectors against itself. This was implemented with two approaches:
    - Brute Force on many different technology platforms
    - Approximation using Google's published technique for Google Correlate[i]
- **Time Series** – This use case is focused on data sources that are a time series. The case study focused on an anomaly detection algorithm and leveraged Auto-Regressive Integrated Moving Average (ARIMA)[ii].
- **Data Aggregation** – Representative of the analysis typically done in the Business Intelligence (BI) world, this use case focused on applying aggregation techniques to tabular data. Two primary techniques were specifically implemented:
    - Dwell Time
    - Aggregate Micropathing
- **Graph Traversal Analysis** – To represent a network based structure and analytic, we also targeted a graph traversal based algorithm. For this analytic, we used a time-based transaction network and analyzed it for time-restricted graph loop detection.

2 Implementation

In this section, we detail the various technical implementations for each of the case studies.

2.1 N^2 Calculation

This analytic was the simplest of our case studies and warranted an approach on a wide variety of Big Data technology platforms. The core principle of the analytic is a full outer join of data against itself. The dataset used for this study was the fictionalized VAST 2012 data, which contained the health status (an integer value) of various IP addresses at 15 minutes intervals. These health ratings were constructed into a vector, where each subsequent vector position denoted the health for the next 15 minute interval. The length of each vector was 192 and we had 1 million IP addresses to sample from. For analysis, we would pull different sized sets of IP addresses from this source dataset to evaluate scaling performance.

The algorithm used for this was Pearson product-moment correlation coefficient and was applied for every unique pairing of IP addresses in a given set. Pearson's correlation ranges from -1 to +1 with 0 meaning no correlation, +1 meaning perfect positive correlation, and -1 meaning perfect negative correlation. Calculating this pairwise between all IP addresses in a given set poses an $O(N^2)$ problem and can also show how data locality affects performance.

For simplicity and to provide an accurate baseline, this was first implemented in RAM by using a Python NumPy script. All vectors were paired together in a matrix and sent into NumPy's corrcoef function, which performs the calculation. While this approach obviously doesn't scale, it provided us a baseline of results.

Apache Hive was our next implementation. For this approach, we used Hive's built in Hadoop streaming capability to parallelize the RAM based version of the problem. First, all of the vectors were loaded into Hive – one row per vector. Then, Hive performed a full outer join on all rows and sent the joined output to the same Python NumPy corrcoef function for evaluation and output.

Mahout, which is a popular MapReduce based Artificial Intelligence library that runs on Hadoop, was the next choice for evaluation. We chose this library for two reasons. First, it was designed and optimized for this type of analysis. Secondly, it has a built-in Pearson's matrix correlation engine. To complete this task, we simply took the vectors and loaded them into a Mahout formatted HDFS file and called Mahout's "rowsimilarity" option with the SIMILARITY_PEARSON_CORRELATION argument selected to produce its results.

Cascalog, a Clojure-based technology that has been gaining traction in industry, implemented a similar strategy to that of Hive by using a full outer join. Each vector was sent in as a row and all rows were combined against one another. When the two vectors were combined, the Apache Commons Math library was then used to calculate correlation between the two vectors. While leveraging a similar approach to Hive, the nature of the Lisp-inspired functional language yielded a completely different development experience.

Giraph was the next technology to explore as it represented a fundamentally different approach using its Bulk Synchronous Parallel (BSP) methodology. Giraph operates such that it takes in a graph of data, shards that data across the machines, and then elevates that entire graph in RAM for expedient calculation. As such, each vector was loaded into a vertex within a graph and each vertex sent its data as a message to every other vertex with an ID higher than it (so as to prevent duplicated calculation). Whenever a vertex receives a message with a vector of data, it uses the Apache Commons Math library to calculate the correlation between the received vector and its own data.

Spark was the last technology that we used on this analytic use case. Again, the full outer join strategy was used and the results were piped into the Apache Math Commons Library for evaluation. Spark is Scala-based and provided a very natural interface for performing this join operation.

Finally, attempts were made to also build the analytic in the R-based RHIPE and RHadoop platforms. However, after many repeated failed attempts to get these technologies to compile, build, and operate on a virtual machine with all of the previous technologies listed, this was abandoned. This was largely due to the specific version requirements of technology stacks for each of these (version requirements for R, Java, Hadoop, etc) as well as other unrelated build problems. The burdensome weight of the setup paired with the short timeframe convinced us to pass on these technologies for the moment – especially since they should exhibit similar performance to the other MapReduce approaches described above. These technologies will be revisited in future work.

This case study was used both for the gathering of benchmarks and was also used to explore approximation methodologies as discussed in the Approximation and Evaluation section.

2.2 Time Series

This use case focused on how to analyze time series data on big data platforms. The obvious approach to solve this problem application was simply to leverage a MapReduce framework around some time series analysis library. In this case, we leveraged R to perform our time series analysis and wrapped it in a MapReduce architecture for parallelized execution. Whereas the correlation engine compared time series vectors pairwise, the goal of this use case was to analyze each individual vector and discover anomalous activity. For this, we used flight records data that provides the history of every airplane take-off and landing since 1987. Our goal was to detect when there were major events that disrupted air traffic travel. Each vector represented an airport and each element within the vector represented the total number of flights that had taken off within that Year-Month-Day-Hour.

To determine anomalous activity, we decided to leverage autoregressive integrated moving average (ARIMA), since it incorporates both aspects of seasonality and moving averages. The ARIMA was calculated for each vector using the (p,d,q) model parameters as automatically provide by the forecast[iii] library. Then, standardized residuals were extracted and filtered to keep events that constituted a +/-4 stddev to find the events.

The vectors for each airport's activity were first built and processed in Hive, but were then sent via Hadoop Streaming into a Python script that called the R based "forecast" library (which provided auto-fit for the p,d,q variables in the ARIMA model). Thus, the ARIMA algorithm always ran on a single vector in a single thread, but Hadoop parallelized the job out to many threads – allowing a parallelization factor equal to the number of vectors in the dataset. Again, this was ideally suited for a RHIPE / RHadoop implementation, but build and environment conflict issues prevented usage of those technologies. Instead, the Python scripts read / wrote directly to and from R to process the data.

The end results can be seen with graphs like the one shown in Figure 1, which shows how September 11, 2001 registers as a major deviation from normal activities. Also note the spike in residuals due to storm activity and Thanksgiving.

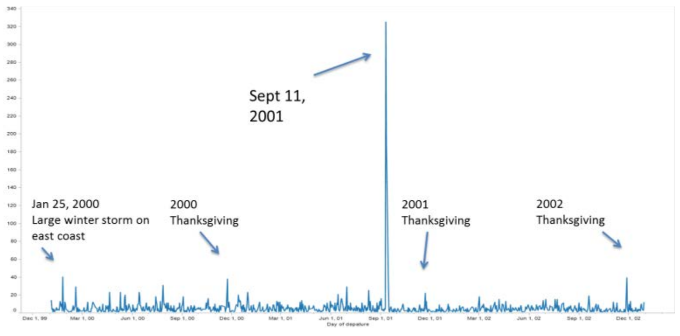

Figure 1: ARIMA Event Detection

2.3 Data Aggregation

These use cases provided the most straightforward and well understood analysis problems in the parallelized environment. Data warehousing technologies like Hive, Pig, Cascalog, Impala, and Spark's Shark were all built for this type of purpose. This is the "Sweet Spot" for the MapReduce architecture. For this use case we performed two styles of data aggregation: dwell time analytics and aggregate micropathing

2.3.1 Dwell Time Analytics

For this use case, we took ship tracking data and analyzed it for places where vessels were stationary for prolonged periods of time. To do this, we first gridded up the world into small boxes divided by latitude / longitude. Then, we measure the overall time spent stationary in each of those boxes. For example, if a vessel remained stationary in the grid box $B_{x,y}$ for a 15 minute period, then $B_{x,y}$ would have a score in seconds of:

$B_{x,y} = B_{x,y} + (15*60)$

Thus, each grid cell captures the total "dwell" time spent by ALL ships within that grid cell.

This analytic was implemented with both a Hive and a Cascalog variant that are very similar in operation. First, the raw data is ingested and sorted by vessel ID. Next, for every vessel ID, its entire track is analyzed to extract out time spent stationary. Finally, that stationary time is mapped to an overarching grid cell, which aggregates the total time spent.

2.3.2 Aggregate Micropathing

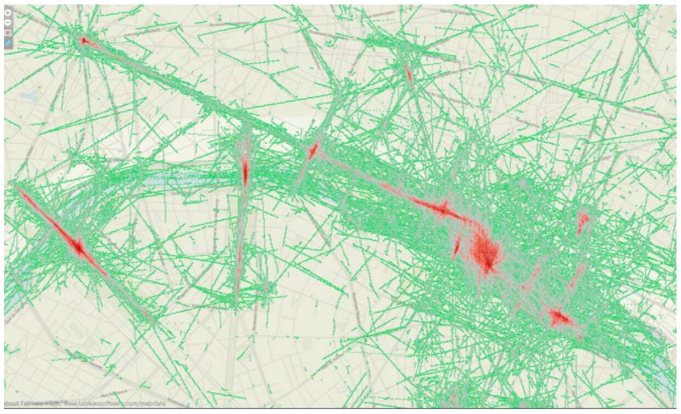

Figure 2: Aggregate micropathing example on Flickr data

This analytic is used to extract aggregate movement profiles. For this analytic, we take any device data, such as vessel movement data or even public photo metadata, and extract overall patterns of movement. Figure 2 shows this analytic as applied to the Flickr / Panoramio metadata in a prominent world city. Red denotes high level of activity with frequent photos – allowing you to discover movement patterns within the city.

This analytic is built via a four stage process (simplified for ease of explanation as follows):

1. Group all records associated with a device / vessel
2. For each device, walk through each event ordinally and extract and filter line segments based on distance and time between events
3. Grid an entire region with "triplines" and compare every line segment for crossings along any gridline
4. Aggregate all tripline crossings across the entire grid

This analytic was implemented using Hive, which took in the raw data and then runs Hadoop Streaming or native Hive jobs for each of the steps outlined above.

2.4 Graph Traversal

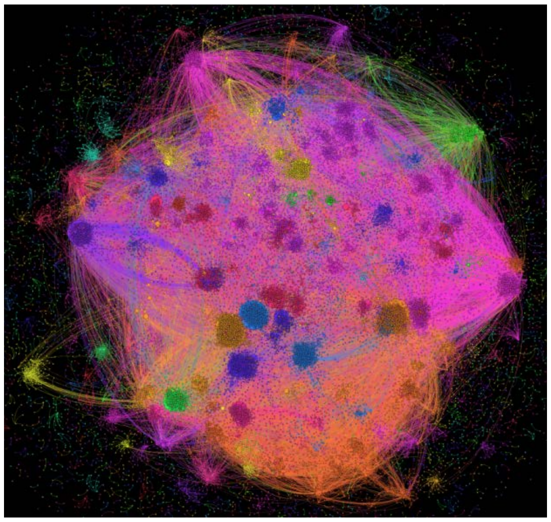

Figure 3: Enron email network

Network data is the primary goal of this last use case. While MapReduce may be efficient for extremely shallow breadth-based graph traversals, tabular processing methodologies like MapReduce, traditionally do very poorly in the field of graph analysis. This is especially the case when trying to perform operations with an indeterminate traversal depth. As such, for this use case, we took the public Enron email dataset and created a graph where all vertices represent an email address. The edges between pairs of vertices represent an email transaction between those two vertexes at a distinct point in time. For the analysis, we want to find a graph traversal pathway from a Vertex A, such that all graph hops are subsequent to one another in time and such that they ultimately return to Vertex A. For example, suppose we have a graph with Vertices A, B, C, D with the following edge list

Source	Target	Date
A	B	2012-09-01
B	C	2012-09-02
C	A	2012-09-03

The edges above would form a graph loop A->B->C->A through time, since all edges occur subsequent to one another.

While this algorithm is difficult to realize in a MapReduce context, it is a completely natural fit for the Bulk Synchronous Parallel approach of Giraph or Spark's Bagel. As such, Giraph was used to load the Enron email graph (Figure 3 shows an image of that email network) into RAM and leverage BSP to solve this problem. On the first superstep, each node broadcasts messages to each of its edges. On every subsequent superstep, each vertex checks its incoming message's dates and sees if it has any outgoing edges with a greater date to propagate forward. If a message ever reaches the node that originally sent it out, then that message is halted and recorded as a loop. Currently, the algorithm has a parameter that determines the depth of traversal that will be conducted. Furthermore, random walks and graph sampling may be leveraged for greater efficiency.

4 Approximation

When initially building out the brute force correlation engine mentioned in Section 2.1, there was an important question as to how we would manage an algorithm that scaled up at $O(N^2)$. For this, we looked to the work that Google performed when they built their system Google Correlate. Google Correlate is an engine that performs an approximation of the pairwise correlation at orders of magnitude faster than $O(N^2)$. More information can be obtained by reading their publication[i].

To quickly recap the methodology, the incoming vectors are transformed through M x P Gaussian IID matrices and then run through *k*-means clustering. Vector quantization on the vectors using those cluster center points then provide an approximate distance that can be compared to test series (thus providing a massive dimensionality reduction).

This capability was built up entirely within Spark (both the training and test capability), as we needed rapid and interactive response capabilities. In the future, it might be possible to use Cloudera's Impala technology as another alternative. The engine allows any arbitrary vector (of the same length as the training vectors) to be inserted into the engine for analysis. In real time, the engine will return all the vectors that were found to be closest via the approximation metric.

5 Evaluation

Evaluations on analytic runtimes, code complexity, and development time all are major factors when deciding how to implement a particular Big Data strategy. This section aims to capture our observations of those factors.

5.1 Quantitative Correlation Engine Metrics

Building the correlation engine on many different platforms allowed us to calculate quantitative performance for each platform. For this, we leveraged Amazon's EC2 cloud to run our tests on the exact same hardware for each approach for comparative analysis. The following hardware configurations were used:

- 5 small instances
- 10 small instances
- 20 small instances
- 5 high memory extra large
- 5 high CPU extra large

Additionally, to measure the scale-up behavior on each hardware profile, we used several different sized vector (4K, 8K, 16K, 32K, 64K) matrices that we sent in for processing. Each approach was given no more than an hour to complete end to end and the time to completion for each approach was measured and recorded.

5.1.1 Brute Force

Figure 4, Figure 5, and Figure 6 below show the runtimes for each hardware configuration and technical approach. Bars are left blank where the technology / hardware configuration exceeded the one hour time limit. It can be seen that Giraph and Spark both perform the best – likely due to the fact that they both pull the data completely into RAM before processing. None of the processes finished the 32K or 64K vector set and only Giraph and Spark made it to the 16K vector set on the largest machine configurations.

Other observations to note are that when using Giraph, you want to use as FEW nodes as possible – where each node has the maximum amount of RAM you can put on it. This is due to the network I/O hit at every BSP synchronization stage. Conversely, MapReduce based implementations desire the opposite – providing as many spinning disks as possible attacking the problem will provide better speed and performance.

Finally, it should be noted that Mahout works faster than its other MapReduce counterpart (Hive). However, despite Mahout being optimized and built for the correlation problem, it scored as one of the worst performing configurations due to its MapReduce backing.

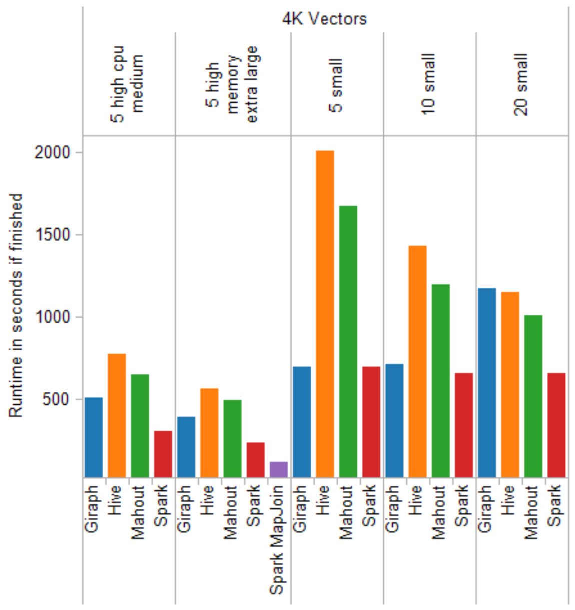

Figure 4: Correlation runtimes for 4K vectors by hardware configuration

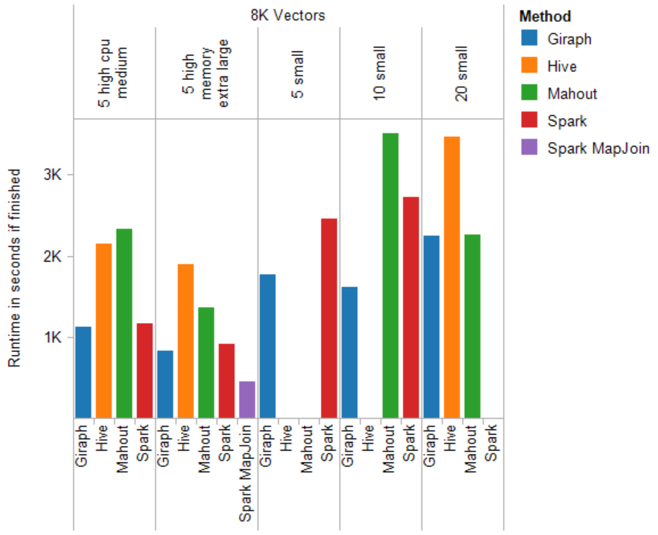

Figure 5: Correlation runtime for 8K vectors by hardware configuration

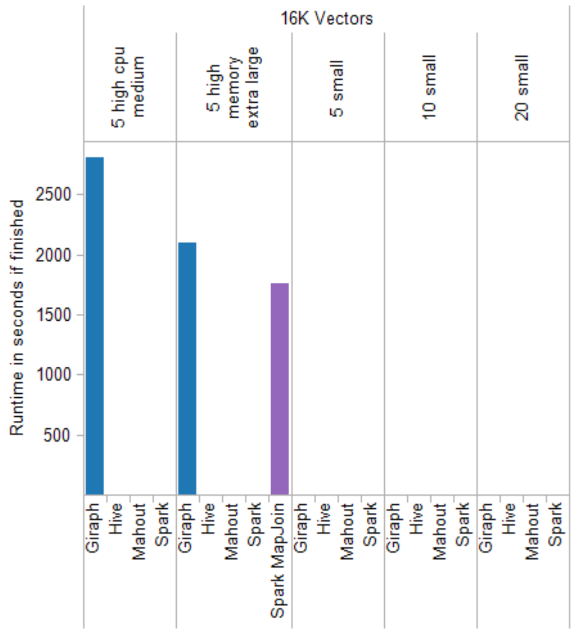

Figure 6: Correlation runtimes for 16K vectors by hardware configuration

5.1.2 Approximation

The approximation engine showed considerable improvement and speed-up when compared to the brute force methods. Naturally, the better performance came at the cost of recall, but this at least allows the user to decide the correct balance between recall and time spent waiting for the calculation.

The approximation engine fared much better in terms of scaling up using a K = 255, M = 8, and P = 8 on the Google Correlate algorithm. However, it should also be noted that one should not keep these parameters the same as the data size increases. Instead they should be adjusted upwards to prevent a loss in recall. However, for the purpose of these benchmarks, the settings above were used and computed at every data size as seen in Figure 7.

The most notable improvement is that we actually finished results on both the 32K vector and 64K vector sets. To demonstrate what speedup this was, we compared the best brute force times against the approximation engine and also included measured the speedup (using the same K, M, and P parameters above) and recall performance.

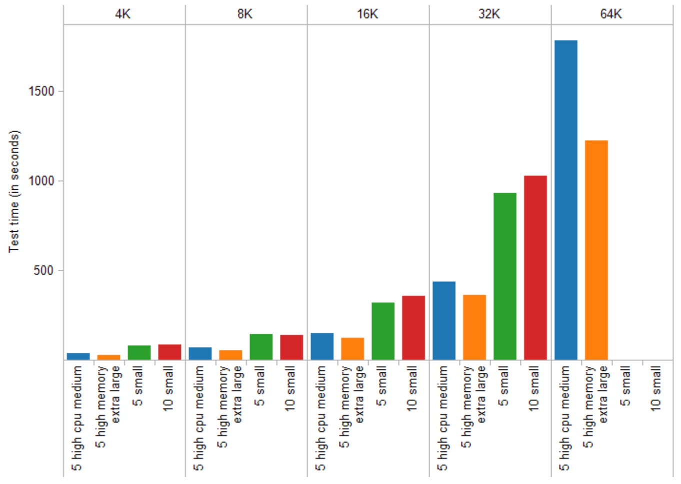

Figure 7: Correlation runtimes by vector size using a Spark-based approximation engine

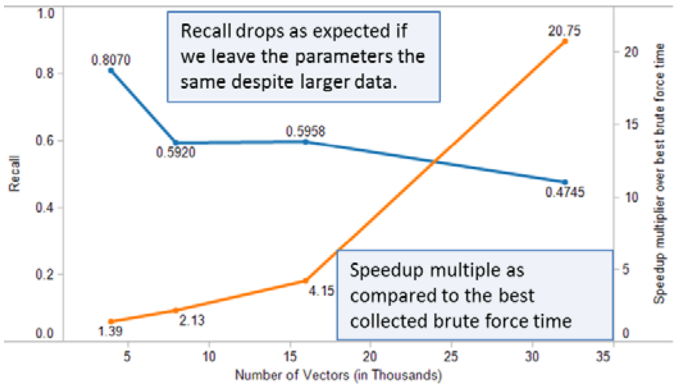

Figure 8: Correlation approximation engine speedup and recall

5.2 Qualitative Assessments and Observations

Aside from the quantitative metrics, we gathered a lot of qualitative data on each of the technology stacks as well. Development time and code complexity each were major factors that are often overlooked in the evaluation of an analytic technique. The following assessments are purely opinion in nature and based on the work of a few people on these key systems. These experiences may not be representative of other people's experiences with the same system – especially for those specializing in one particular system.

5.2.1 MapReduce vs. Bulk Synchronous Parallel

In general, a good rule of thumb is to use MapReduce for tabular operations (aggregations and jobs that are easy to split apart), but to use Bulk Synchronous Parallel (BSP) for any traversal operations (i.e.: graph and networks). Some network algorithms that perform only a single traversal or two may be manageable in MapReduce, but the BSP framework will yield far less complex code overall and should be used instead.

5.2.1 Technology Stack Overviews

For MapReduce, we found Hive to be extremely efficient in terms of the # of lines of code and also development time. The development abstraction over writing a custom mapReduce job is worthwhile and saves much code and time. This does come at the cost of less flexibility and lower performance on the cloud (sometimes much lower performance for certain operations that are abstracted too much). The code also tends to be messier and the pipeline for moving data between processes is ugly at best. However, by far, this was the most straightforward and easiest solution for parallelizing problems with an obvious parallelization solution (whether you're piping data into R, NumPy, or some other program). For ad hoc analytics development, where the development of rapid analytics is the key, Hive is a clear choice for these situations (or at least to get started).

Cascalog, a newer and more elegant option allows for pipelining of the data from beginning to end. It also allows for much better modularity and maintenance of code. Both of these are a significant advantage over Hive. However, the development times for simple algorithms tended to be longer and the coded tended to be more complex. The Lisp-based dialect also instills a steeper learning curve for those less inclined to the functional programming world. Cascalog also imposes steep performance penalties for the abstraction that it provides – worse even than Hive in the cases we compared them. That said, the elegance and maintainability should keep Cascalog as a consideration moving forward. It is constantly being improved upon and its ability to pipeline data from beginning to end in a sane fashion is quite attractive. As a final note, Cascalog was not benchmarked for the correlation engine as there were troubles configuring it to operate at scale. Rather than risk showing an incorrect Cascalog performance metric, we decided to omit the Cascalog metrics we have until we can further optimize its execution. This is an ongoing point of future work.

While Mahout is a popular library, the code complexity required just to get data in and out of the system is very burdensome (and sometimes more code than the actual analytic you are trying to use!). Mahout has a few nice capabilities, but many of its capabilities work far better in other platforms. A good case in point is the iterative *k*-means clustering, which will work far better in something like Spark, which can hold intermediate state in RAM. Mahout is useful for very specific classes of analytics and only operates in a MapReduce framework, which may not always be the best approach for the problem at hand. That said, Mahout is useful when you need an implementation of some machine learning technique that someone has already implemented – if you can stomach writing endlessly annoying serialization / de-serialization code to get your data in and out.

Giraph and Bagel are both great options for the BSP side of the world on graph processing. The simplicity and ingenuity of the BSP compute method makes writing graph algorithms extremely simple. Code complexity seems to be far less burdensome in Bagel because Giraph requires a lot of custom definition of input / output serialization procedures. Bagel avoids a lot of the serialization procedures because it keeps everything in RAM.

Spark showed as one of the most interesting solutions in the assessment. Though it had a performance boost mainly thanks to the distributed memory, the elegance of the Scala code that was produced to build the analytics was astoundingly clean when compared to all the other options. Pair that with extremely fast computation time (especially for iterative operations, like *k*-means) and it makes for a very powerful system. Bagel and Shark also exist on top of Spark which make the package even more attractive. In all our tests, the Spark systems came out with the fastest results, cleanest code, and least development complexity.

RHadoop and RHIPE look extremely promising, especially when installing them on a new system by themselves. However, trying to create an environment where these technologies work alongside all of the other technologies above can prove to be a challenge. Version requirements, compiler issues, distributions, and other factors all played a role in what can be summed up as an installation nightmare. Provided they are installed on a system, the Divide and Recombine features should be useful – though probably only as a mirror to the other MapReduce process that can be done using the other technology stacks. However, abstraction of these capabilities is certainly desired and further investigation and diligence is warranted in trying to get these technologies operational alongside the other systems.

6 Future Work

Future work would include measuring the scale-out potential on different hardware configurations for the Graph Loops, Time Series, and Aggregation techniques. Additionally, current efforts are working to provide benchmarks on the correlation engine leveraging other technologies where appropriate, such as:

- Spark's Bagel
- GPU CUDA
- Storm (if we ever find streaming data)
- Impala

- RHIPE
- Cascalog

References

[i] Mohebbi et al. “Google Correlate Whitepaper”. Google, pp. 5-6.

[ii] http://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average

[iii] http://cran.r-project.org/web/packages/forecast/index.html