Group 1: Justin Baraboo(1), Billy Capps(3), David Rodgers(21), Brendon O'Connor(17)
Paper Review 2

Title: The Emergence of Adaptable Complex Big Data Systems

Authors: Justin Baraboo, Bill Capps, Brendan O'Connor, David Rodgers

Abstract

Some big issues of today's world involve large amounts of data, but not all can be met with basic paradigms that were founded the beginning of the Big Data era. There is an increasing need to solve different data types and with different execution environments; there are many angles to a problem that require different machine learning algorithms and data from different sources. In this paper we talk about 6 fairly new frameworks, such as Pipeline61, Summingbird, Heron and S4 to name a few, and analyze how these new frameworks are meeting the new challenges of Big Data today.

Introduction

Big data is relatively new, and with frameworks such as Hadoop, an assortment of basic computations can be done fairly efficiently, especially in such a MapReduce paradigm. The common case is the word count program, with a simple key value structure, outputting a integer count value with a specific string value. Of course, this can be used in variety of ways, but it is limited. There are new and varying data structures such that problems cannot be simplified into such data types. Moreover, how we obtain data is changing, and using the standard platforms in their singularity such as Hadoop begins to pose computationally costly problems. However, the way in which we analyze big data is evolving to meet these changes at a rapid pace. The current trend of big data processing frameworks is moving towards a more intricate and flexible architecture that is able to tackle the increasingly complex problems that bring along with it a myriad of data types and environments.

Body

Many large-scale big data processing tasks require that data move through several different platforms, known as a heterogeneous execution environment, to complete complex analysis. The data travels in a sequential pipeline, but code is required to facilitate the handoff of the data from one technology to the next. This glue code can represent 95% of the total project code. The Pipeline61 framework is designed to make the development and maintenance of these heterogeneous execution environments easier while also reducing the amount of overall code (Wu). The Pipeline61 framework consists of three major components: data service, dependency and version manager, and execution engine. The data service provides a uniform interface for the data and manages any required conversion (Wu). The dependency and version manager automates much of the administration allowing changes to be tested and monitored through the sending of messages. The execution engine monitors and manages the pipeline activity (Wu). Each pipeline component is referred to as a pipe. Pipes are uniquely named and given version numbers. Pipes also contain information about the type of data that will pass through them and information about the execution context (Wu). The execution context references the data-processing framework associated with the pipe. The three major execution contexts available currently are: Spark, MapReduce, and shell (Wu). Pipeline61 does have a cost for the ease of development. It does not check for data schema compatibility and thus developers must manually check the output of each pipe is suitable for the input of the next pipe. Also, intermediate results often must be written to physical storage when switching contexts. This causes Pipeline61 to much slower than other options (Wu). But despite these drawbacks, the Pipeline61 framework can meet today's changing standards for Big Data. Its ability to adapt to different execution environments make this idea of Pipes an innovative opportunity compared to just the original MapReduce in Hadoop. Indeed, using MapReduce and other execution contexts offers a flexibility of analysis that opens up potential solutions for more complex solutions.

Summingbird is an open-source language designed to bring together online and batch map reduce applications. Summingbird can generate either Hadoop jobs or Storm topologies as needed and without changes to the program logic. The code will look like native Java or Scala collection transformations so the learning curve is minimal, but the results can be run in either batch or online mode as required (Boykin). There are some limitations. Reductions are limited to a specific algebraic structure known as a commutative semigroup. The authors state that this has not been a problem for them at Twitter. The commutativity allows the operands to appear in any order and still arrive at the same result. There are many commutative semigroups that are extremely versatile and useful. One obvious example is the set of integers including zero with addition. This means that Summingbird is capable of handling counting without any problems. Some additional examples of allowed results are bloom filters, hyperloglog counters, and count-min sketch. These are probabilistic data sets that provide approximations of exact datatypes, such as sets or hashmaps (Boykin). Summingbird can complete left joins through the service abstraction. This is very much like joining a dimension table with a fact table. One example is counting how often URLs appear in tweets. The URLs will be in a shortened form in the original tweet. A service is required to lookup the full URL. The full URL can then be joined to the key-pair structure holding the shortened URLs and counts (Boykin).

One of the powerful uses of Summingbird is to operate in hybrid mode where both batch and online options are used together. This can allow periodic processing to be done in Hadoop while recent information from an online system is incorporated. This is completely transparent to anyone downstream. If a query wants a year-to-date count of hashtags, counts for the year until yesterday can come from Hadoop while the current day's counts can come from Storm. Summingbird will combine the counts and provide a single count (Boykin). The hybrid is a clear example of the flexibility of analysis that Summingbird can achieve. While the example the authors use are for a specific task, the task itself could be more complex or could even potentially combine pieces of a lambda-like architecture, where a streaming component can use a model that is updated by the batch component. This design could handle both data that is streamed and already collected while at the same time performing both training and testing phases, all potentially in (or close to) real-time. Indeed, Summingbird can provide useful opportunities for data scientists today.

Deep learning of traffic flow can benefit from a stacked autoencoder architecture. When looking at traffic data, researchers found that a stacked autoencoder architecture reduced error drastically from other deep learning and machine learning techniques. The traffic flow problem is defined by the following description: given a sequence of sets of chronological traffic flow quantities, that is each element of a set is a traffic flow quantity observed within that time partition, one wishes to predict the traffic flow at a time interval t plus a prediction horizon. Previous solutions belonged to parametric models, nonparametric models, and simulations, these techniques included: time series models, k-nearest neighbor and neural network models, and traffic simulations. This problem is rich with many specialized and research techniques such as ARIMA, KARIMA, SARIMA, k-NN, SVR, Bayesian networks, and neural networks (Lv). The researchers found that most neural network architectures were shallow with only one layer, due to difficulty in gradient-based training; however, they also found new research of state of the art methods used for deeper neural networks making it feasible to apply to this problem.

By itself, an autoencoder is easy to understand. It attempts to encode the information input onto a smaller hidden layer and then reproduce it as output on an output layer. This may seem redundant as, if done perfectly, will produce the same output for a given input; however, the power of the autoencoder is that the hidden layer is smaller, thus reducing the dimension of the given information. If the autoencoder did have the same or larger amount of nodes as the input layer, it could essentially be learning the identity function which isn't useful. This can further be overcome by using non-linear autoencoders with sparsity constraints, creating a sparse autoencoder. This not only tries to minimize reconstruction error but also uses information criteria to provide the sparsity constraint to determine the overall error. The autoencoders are stacked to form a stacked autoencoder architecture (Lv). It does this by taking input from the output of a previous autoencoder. The number of hidden layers and nodes were chosen by the researchers (probably to minimize error) and were tested at different time intervals looked at for traffic flow. The models were trained not by the regular backpropagation technique but by a recent greedy unsupervised learning technique. This design and architecture outperformed back propagation neural networks, support vector machines, rw, and rbf nns, by allowing for a more deep architecture (Lv).

For the four categories investigated, the MRE of the stacked autoencoder architecture only had around 6% MRE, while BP NN hovered around 10.5%, RW varied greatly about 15% , SVM varied about 15%, and RBF varied greatly around 16% (Lv). The greatest variance for the last three models was formed by being able to be accurate for short time intervals and less accurate for large time traffic flow. In fact all models except for BP NN did well for the short 15 min traffic flow prediction. Stacked autoencoders did well for each one (Lv). This stacked architecture, while complex, can also adapt to learn in a more complex way, even when unsupervised, which is preferable if you are wanting a more accurate model. The use of deep learning in neural networks is thus becoming an emergent approach to model complex solutions that require a lot of data with a significant number of parameters.

Heron is twitter's improved platform that built on the successes of storm. The main purpose for Heron coming into existence was that the original storm system began to seem inefficient and unwieldy to use at the scale Twitter needed it to. One of the key features that prompted the need to rebuild storm was debug ability. Storm topologies can break for a variety of reasons in order for the system to perform at an industrial level it needed to be able to be easily fixed (Kulkarni). Since Heron is a rebuild of the storm system the designers also made sure to make all storm topologies compatible with the heron system. This was to save any development time already spent.

One part of the architecture that is unique to Heron is the Topology Master. The topology master is responsible for managing the topology throughout its existence. Rather than being involved in the processing this is only involved in high level operation of the topology. Each topology has its own stream manager as well (Kulkarni). This part of the architecture is going to be responsible for managing the flow of the tuples through the system. One key benefit of this is it allows for back pressure in the system. This prevents the problem where one part of the topology isn't processing as fast as others and the tuples start to build up as a bottleneck. Another unique aspect to the architecture is the heron instance. This is different from storm in that each one is a JVM process. This significantly simplifies the debugging that, as we mention earlier, is a key motivation for the system (Kulkarni).The combination of these features leads to an overall increased transparency in how the system works which makes it easier to deal with and debug.

The final major considerations that Heron addresses are as follows: ability for users to interact with their topology, the ability for users to view metrics on their topology, ability to see exceptions that arise in their topology, the ability to see their logs. These features are implemented through Heron Tracker, Heron UI, and Heron Viz. The tracker as its name suggests is primarily concerned with tracking metadata on the topologies. It provides a simplified REST API for ease of use and extension (Kulkarni). Building on top of the tracker the UI helps provide a clean interface for users to access the information on the topologies. Finally the Viz portion is essentially a dashboard for viewing the topology metrics. These metrics are categorized into health, resource, component, and stream manager (Kulkarni). Collectively all these improvements in Heron hasn't sought to fundamentally alter what one is trying to accomplish from storm (parallelized stream processing). Rather, this system is a more refined tool that is more efficient and more simple to implement.

While the tool itself is more refined, and in a way very similar to Storm, it owes its success to its ability to handle complex inputs. The ability to schedule different topologies with the aurora scheduler is something vastly complex compared to Storm, and the ability to manage multiple topologies provides the ability to develop multi-faceted streaming analysis with one system. This increasingly complex architecture thus creates better, more efficient solutions.

The Simple Scalable Streaming System (S4) has a similar stream processing framework as MapReduce (Neumeyer). Today there are many widely used search engines that utilize certain well-developed algorithms to dynamically gauge the probability of a click on the ad given the context (e.g., user preference, location, prior clicks, etc.) Especially with major search engines like Google, thousands of queries need to be processed per second.

S4 was born out of the need to process user feedback that had a low latency and was scalable. The authors envisioned an architecture that could meet the standards for research and everyday production (Neumeyer).. Research environments would require much flexibility of deploying algorithms quickly, which provides the ability to test these algorithms live without many resources and minimizing time; production environments require scalability and high availability. Given this information, the authors developed S4 with the following design goals: provide a simple interface for processing data streams; design a cluster with high availability and scalability; minimize latency using local memory in each processing node; use a decentralized and symmetric architecture, such that no central node has specialized tasks; use a pluggable architecture to maintain a customizable design; make the design "science friendly", or flexible and easy to program (Neumeyer).

S4 uses Processing Elements, or PEs, for computations. Each PE state cannot be accessed by a different PE. PEs only share the events that they consume and/or emit. Such a structure enables the user to route events to certain PEs and, if necessary, create new PE instances. It has 4 components: functionality, event type(s), keyed attribute, and value attribute. (Neumeyer) The PE prototype is a unique PE object that has functionality, event type and keyed attribute components but an unassigned attribute value. For any value, the PE prototype is able to clone itself to create fully qualified PEs of that class that has the exact configuration and value (Neumeyer). Processing Nodes, or PNs, are the hosts to PEs. A PN's main roles are listening to events, performing given operations on such incoming events, dispatching events (with help from the communication layer) and emitting the event's output. The authors designed these PEs and PNs so that all events that have a certain value of a keyed attributed are guaranteed to arrive at a specific PN in addition to being routed to the appropriate PE instances within in it (Neumeyer).

The authors conducted a few different experiments. They did an online experiment in which they looked at live search traffic over 2 weeks. The goal was to compute click-through rate (CTR) for a query and ad quickly. They also did an offline experiment that performed a similar task but not in real-time; its goal was to evaluate performance at increasing event rates. Overall with these experiments, S4 could not process the event stream fast enough, but on events where it could handle the speed, the CTR increased by 3% without revenue loss (Neumeyer). They then added an online parameter optimization system (OPO). In real-time, S4 could keep up with the events and increased revenue by 0.25% and click yield by 1.4% (Neumeyer).

There are some modest performance increases, but for a new dynamic system, this seems promising, and in some use cases such as this one, profitable. It also tackles relevant issues of tackling large amounts of data quickly and effectively, but with an innovative approach (namely the use of the processing elements and nodes in a decentralized system) to quickly adapt to the immense scale of data used, especially in the offline experiments. It is an architecture that data scientists should keep an eye on, as the S4 architecture has much potential to tackle today's complex issues.

The authors of *A Simulator of Human Emergency Mobility following Disasters: Knowledge Transfer from Big Disaster Data* want to be able to simulate greate natural disasters and their effects on human movements and evacuations, which they argue is key to plan effective relief, disaster management and long-term societal reconstruction. While this is fairly difficult to study, the authors creatively use GPS trajectories data found on people's cell phones. Their main priorities is to gather knowledge about human mobility post-disaster using big data and to develop a model of such for simulations. The type of data is important. They use "big and heterogeneous data", or a high number of user's GPS records (1.6 million) over a few years in addition to thousands of instances of earthquake data and data collected from news reporting, transportation network, etc (Song). To analyze such data, they utilized 5 computers to build a Hadoop cluster with 32 cores, 32 GB of memory, and 16 TB of storage. It had the ability to run 28 tasks simultaneously. The model they used for computations is the HMM model, or Hidden Markov Model, to model the dependencies between the various states (i.e., home location, working location, unknown location, etc.)(Song). They used the Expectation Maximization (EM) algorithm to estimate the likelihood of the location of transfer sequences; to determine the correct number of hidden state, they utilized the Bayesian Information Criterion (BIC), taking the number of states that correspond to the lowest BIC, which typically indicates a better model (Song).

To test their results, they selected 80% of the data for model training, and 20% for testing. They compared their model to 3 different models: Most Frequented Location Model (MF), Gaussian Model (GM), and Periodic Mobility Model (PMM). Compared to these models, their model performed the best, with an overall accuracy of 65.26% (with the second best from PMM at 63.18%) and a 90% matching rate of 45.33 (with the second best from PMM at 42.69%) (Song).

In general, it is difficult to predict a natural disaster's effect on human mobility, but their utilization of big data from various sources combined with the hidden markov model, EM algorithm and BIC improves our ability to learn and simulate such seemingly chaotic and unpredictable effects**.** Moreover, it is noteworthy that this multi-faceted approach with the 3 main machine learning algorithms are able to both handle more complex and varied data types as well as produce an overall more accurate model of what is seemingly unpredictable. This underscores the importance of providing a more complex model to tackle the more complex problems that the world poses today.

Conclusion

In sum, there are many new systems emerging today that are able to adapt to complex problems. Pipeline61 can offer the ability to run multiple execution environments to do MapReduce tasks with potential analysis from Apache Spark libraries. Summingbird uses commutative semigroups to better deal with probabilistic data while also providing the ability to use both online and batch processing mechanisms. With Heron, its structure becomes more involved with the addition of management tools, but these managers (i.e., Aurora Scheduler and Topology Master) greatly improves performance across the board. Moreover, S4 uses processing elements and nodes, a non-inconsequential piece of its architecture, to process large amounts of streaming and batch data with great accuracy. Additionally, the authors who created a model to predict human movement after a natural disaster owes its success in large part to its intricate machine learning algorithm and its ability to handle a large volume of data from different sources. In conclusion, problems are no longer from one source and require deeper analyses to fully learn, understand and model the issues we encounter, but with these frameworks, we can see that we are continually evolving to meet these challenges of Big Data today.

Future Research Directions

The focus of this paper was to survey some of the existing technologies that have emerged in recent years to address the challenges being faced across many fields looking to leverage their data. There remains many open and challenging problems in this field. Some of the limitations of these current systems have been mentioned throughout this paper. The limitations of the current systems present obvious areas of improvement. Much of these systems represent the fields first attempts to deal with data in a distributed manner and on a scale not previously seen. As seen with initial attempts at software development, we expect much more research to be done in optimizing and simplifying commonly performed tasks on data. The tools that are currently available also lack much flexibility in use. We expect much more research to be done in expanding the scope of these tools and use cases. As the set of applications and capabilities expand so too shall our expectations and desires to attempt to do different things with the data. This will lead to new technologies to address these new demands. As these technologies are still in their infancy it is likely we will see more fragmentation among tools and available options.

References

Boykin, Oscar, et al. "Summingbird: A framework for integrating batch and online mapreduce computations." Proceedings of the VLDB Endowment 7.13 (2014): 1441-1451.

Kulkarni, Sanjeev, et al. "Twitter heron: Stream processing at scale." Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015.

Lv, Yisheng, et al. "Traffic flow prediction with big data: a deep learning approach." Intelligent Transportation Systems, IEEE Transactions on 16.2 (2015): 865-873.

Neumeyer, Leonardo, et al. "S4: Distributed stream computing platform." 2010 IEEE International Conference on Data Mining Workshops. IEEE, 2010.

Song, Xuan, et al. "A Simulator of Human Emergency Mobility following Disasters: Knowledge Transfer from Big Disaster Data." Twenty-Ninth AAAI Conference on Artificial Intelligence. 2015.

Wu, Dongyao, et al. "Building Pipelines for Heterogeneous Execution Environments for Big Data Processing." IEEE Software 33.2 (2016): 60-67.