Brandon McKune
Professor Raj Sunderraman
CSc 8711: Database and the Web
May 3rd, 2013

Experience BigData

BigData is an easily scaleable storage framework that provides the user with seamless hardware integration and distribution. Within BigData there are a multitude of different open source software pieces that help facilitate this scaleability and seamless integration. In addition to the storage framework, BigData provides the user with the ability to complete different computational tasks via an open source RDF framework called Sesame2, also known as OpenRDF. Once a user has BigData configured and running properly, implementing any sort of application is pretty straight forward. The following sections will provide the problem BigData is trying to address, why BigData was developed, description of the architecture along with implementation guidelines, and provide insight into the included project.

In today's connected world we have more information than can be computationally inspected by a single computer. According to one research firm, the number of objects stored in Amazon S3 was approximately one trillion.[1] Amazon S3 is the cloud service provided by Amazon company. A few other vendors being Google cloud, Microsoft Azure, and iCloud by Apple. This list is not exhaustive but provides insight that the Internet houses large amounts of heterogeneous data. BigData was built in order to address the heterogenous nature of pharmacology, finance, fraud detection, and intelligence analysis.[2] While a single node of information might contain new insights into the next generation; being able to combine this information in order to search out complex relationships and connections could provide meaningful insight to operational decision making.

Addressing large amounts of heterogenous data requires several capabilities that BigData must provide. Before any analyzing can be performed, the system(s) must be able to 'load' the data into a processing capable environment where effective queries can be conducted. This requirement goes beyond the reasonable capabilities of high end server platforms.[2] Another requirement is sometimes overlooked by the common

programmer and that is the datasets across the Internet is always in constant state of flux. That is the data sets are constantly being added to, deleted from, and modified at any rate. This requires BigData to be able to continuously integrate data in an effective and efficient manner. During the constant state of flux the system must also provide a historical account of the changing data in order to provide different functional capabilities. BigData provides these capabilities and more.

BigData utilizes copy-on-write B+Trees to implement their storage structure across a cluster of machines, which includes a cluster of one. The B+Tree architecture provides the necessary Input/Output (I/O) capabilities efficiently over the average time. These capabilities include searching, inserts, and updates of tuples on average in O(log n) time. B+Trees also provide BigData the capability to perform balance operations during insert and delete along with dynamic partitioning of that data. The copy-on-write implementation of the B+Tree in BigData provides a required historical record to be kept on all transactions executed. While adding overhead, this provides BigData with the ability to role back transactions and provide high concurrency without the usual two phase locking (2PL) that normal data storage systems have.[2]

During runtime BigData has a requirement for dynamic partitioning of the stored data in order to easy scale on continuously growing large datasets. In order to satisfy this requirement, BigData stores individual datum into 'tuples' which consist of a key/ value pair (an Index), delete flag, and a timestamp. The key/value pair consists of byte arrays which allows the user to define what is stored during execution. For the example application, this data is Resource Description Framework (RDF) data. Dynamic partitioning of index partitions is pretty straight forward, An index partition is a set of indices over a range of values (called key-range shards). The partitioning mechanism can split, move, and join different index partitions. A split divides a single index partition into two separate index partitions over the same range of values as the original index partition. Moving consists of moving one index partition over to another physical location while joining takes two index partitions and joins them into one partition over the range of values. In order to keep up with the index partitions, BigData provides a service called the Metadata Service.[2]

The Metadata Service is a specialized Data Service that provides the role of a data location service. In order to locate the necessary data requested, the Metadata Service maintains a routing structure that maps the value ranges of the index partitions to a unique identifier. The Metadata service also incorporates Zookeeper to resolve the physical addresses of each Data Service. One Metadata Service can potentially access hundreds of petabytes worth of data. Although the Metadata Service has limits, BigData implemented this service to be scaleable as well, creating a tree like Metadata Service structure, enabling BigData to be boundless on the amount of reachable data.[2]

The Data Service encompasses two different types of data access, the Journal and the Index Segment. The Journal is an append only write buffer used to store data which will later be moved to the index segments for historical purposes. The Index Segments are read-only sections of data that were once Journals. Once a journal overflows, a new journal is created and the old journal is asynchronously transitioned into a new index segment. A single journal can be accessed by multiple writers at the same time which could cause concurrency control issues.[2]

BigData provides a concurrency control mechanism called MultiVersion Concurrency Control (MVCC). This is different than the normal 2PL mechanism typically found in data storage services such as file systems. MVCC allows readers to read data at all times and never block; writers also run concurrently even if those writers access the same memory block. Conflicts are addressed using the timestamps and an algorithm developed by BigData, which includes failing the second transaction if the conflict cannot be resolved programmatically. Although accidental mishaps can occur in any software system, BigData provides a mechanism for database history.[2]

BigData manages the database history in what they call an immortal database architecture. This architecture provides the capability for the user to configure the history retention policy for those items that were removed. This history retention provides the user a period of time in which they can roll the database environment back to a previous time. Not all systems require this type of historical data and thus BigData provides the means to not store any history by providing the value zero. History is one aspect of why BigData can be so valuable while another is its' high availability architecture.

High availability is described as a systems ability to be reliability used with minimal to no downtime. An example of high availability is Google's search engine's website http://www.google.com. BigData is designed such that there is replicated state throughout the cluster. That is the data and services are replicated throughout the cloud such that if one service or store immediately becomes inoperable, then another is available to replace the inoperable node and take over that node's workload. Another key implementation detail of BigData is that it attempts to "obtain maximum performance form local computing resources." [3]

In order to take full advantage of the underlying scaleable architecture BigData provides a computational mesh of Advanced Programming Interfaces (API) and configuration capabilities. The API that BigData provides is directly from Sesame 2, also known as OpenRDF. This API abstracts the complexities of BigData's storage structure from the user. OpenRDF API to store, modify, and query RDF triples within the BigData storage base. One key aspect of BigData is that once the user is familiar with OpenRDF, they can easily transition their projects into the BigData cluster using minimal code changes.[4]

BigData supports three different types of RDF, triple, triples with provenance, and quads. Each type shares common elements from the RDF schema; however, this paper will concentrate on RDF triple stores. Every value within an RDF triple (subject, predicate, object) is converted by the Lexicon service into a locally unique identifier. Locally unique is defined as a unique identifier within an index partition. The Statement service will take the relationship and determine the access path for the pattern presented. That is the Statement will determine the most effective access path from the master node to the last RDF value in the cluster.

BigData operates over a cluster of physical machines, each supporting possibly multiple services. In order for BigData to write an RDF triple value, a distributed program is defined using the Java classes provided. The program, which is termed "a master" spawns subtasks that execute the jobs available as instructed by the master. Each subtask is responsible for handling the task assigned and writing to the localized write buffer. This write buffer is shared by all subtasks within the client, thus allowing for larger writes to the remote object increasing I/O rates. The task responsible for each

buffer is also responsible for removing duplicate RDF triples, handing errors, and provides decoupling from the working subtasks so that they may continue working with minimal latency. Writing data is important in storing information; however, making sense of the stored data is very important in satisfying the stated goals of BigData.[2]

Querying the large datasets is very important process that BigData provides. The two services mentioned earlier, Lexicon and Statement, are very important when satisfying requests from the user. BigData architecture allows for a master/slave query relationship. This method of querying data is called pipeline join. Essentially, the master coordinates all work among the clients. Once the Lexicon translates the RDF values into their perspective identifiers and the statement determines the relationship of those identifiers; the client receives their respective portions. All computation happens within the clients and the results tables are then joined and pushed back into the write buffer from the client.[2] This is when the master reads that data and returns the results to the user. In order for the user to have internal glimpse at the execution, BigData provides a logging service called Log4j.

Log4j is an open source log project that allows users to log data easily using their minimal API set. However, OpenRDF uses another type logging mechanism very similar to Log4j. It should be noted that the user will need to bridge the gap in order to run BigData due to the different loggers. The necessary file informs the system to convert all log4j log messages into slf4j log messages. For the project attached with this report, the file used is slf4j-log4j12-1.6.1.jar.

The project started as an OpenRDF main memory project and grew from there. In order to understand OpenRDF, it is suggested that the user complete the required OpenRDF tutorial. OpenRDF is very similar to JenaAPI project issued in *Database and the Web* class at Georgia State University. OpenRDF requires the user to create a SAIL repository that all RDF triples are stored. This repository can be localized main memory, file system storage, or a remote repository found on an HTTP server throughout the Internet. Once this repository is created, the user must request a connection to that repository and then they can insert new RDF triples, open a file containing an entire RDF database, or connect to an already loaded RDF store. The

user can also query this repository for information in order to satisfy whichever requirement that the software requires.

To switch the OpenRDF project to a BigData project, the user will include the required libraries needed to run BigData. In order to see a list of those libraries, view the attached project within Eclipse under *Reference Libraries*. The user will also need to create a properties file that contains those properties BigData will use to create the cluster. In the example project the minimal full feature properties are included. Once the properties list is created, the user will need to create a properties object in the source code, initialize the transaction service property timeout and exchange the OpenRDF SAIL repository for a BigData SAIL repository using the configured properties object. Once the repository is created, no other code changes need to be made to affectively execute the required behavior.

BigData was created in order to address a real-world issue of large datasets over heterogeneous boundaries. In order to satisfy the constraints required by datasets of this magnitude, BigData must easily be scaleable and provide meaningful computational features. Utilizing the scaleable B+Trees, high availability and MVCC, BigData is able to address scaleable concerns. The implemented OpenRDF API on top of BigData allows the user to effectively manage that data while abstracting the complexity of clusters and physically separated hardware. Once the user understands to OpenRDF API and is able to configure a BigData cluster, there is minimal changes needed to scale along with the data and almost no changes needed in the code base.

References

1. Callaham, John. “Research firm: Over 1 exabyte of data is now stored in the cloud”, February 20, 2013. *Neowin.net*. May 4th, 2013. http://www.neowin.net/news/research-firm-over-1-exabyte-of-data-is-now-stored-in-the-cloud.
2. “BigData Scale-out Architecture Whitepaper” 2009. SYSTAP, LLC.
3. “BigData Scale-out Architecture Whitepaper” 2009. pg 13. SYSTAP, LLC.
4. “Getting Started” September 26, 2011. *Sourceforge.* May 4th, 2013. http://sourceforge.net/apps/mediawiki/bigdata/index.php?title=GettingStarted