Implementing a social network analytics pipeline using Spark on Urika XA

Mike Hinchey
Analytics Products Group
Cray Inc.
Pleasanton, CA, US
mhinchey@cray.com

***Abstract*—We intend to discuss and demonstrate the use of new generation analytic techniques to find communities of users that discuss certain topics (consumer electronics, sports) and identify key users that play a role in or between those communities (originators, rebroadcasters, connectors).**

The analytics execution is performed on a Cray Urika XA [1], a cluster of 48 nodes with 4T of RAM, 38T of SSD, and Lustre storage. The software framework used is Apache Spark [2] and HDFS (Hadoop Distributed File System) with the Scala programming language. The Apache Spark framework is similar to Hadoop/MapReduce, written in a functional style, allowing the engine to make efficient use of the full cluster, lazily, in parallel, with failure recovery, but without the user having to code for such complexity. The entire pipeline includes ETL (Extract, Transform, Load), numerous aggregations and joins, and a graph algorithm. Spark Streaming is used to process data as a series of micro-batches, to reprocess historical data or process live streaming data for near real-time results from complex event processing, to identify patterns and changing trends.

***Keywords*-Social network analysis; SNA; Twitter; graph algorithm; analytics; Apache Spark; Urika XA; HDFS; Lustre; Scala; D3.js**

I. Introduction

The customer business use case is to discover communities of users in a social network and identify certain key users as measured by various roles exhibited by the activity in the data. That is, given Twitter tweets, which have users referring to other users, we can infer who are the originators of popular content, rebroadcasters, and the connectors between multiple communities. The entire pipeline includes ETL, numerous aggregations and joins, and a graph algorithm for community detection. Spark Streaming is used for complex event processing on real-time data, to identify patterns, such as long chains of retweeting, and changing trends such as hashtags becoming popular.

The analytics execution is performed on a Cray Urika XA which is a cluster of 48 nodes with a total of 4T of RAM, 38T of SSD, and a Lustre disk. The software framework used is Apache Spark and HDFS with the Scala programming language. Spark has a programming model similar to Hadoop/MapReduce, but easier to use for more complex pipelines. The Spark engine uses in-memory computation with reduced disk usage for much faster execution than MapReduce. Lustre is used as permanent storage of the source data files, which are gzipped JSON records. The SSD is used for HDFS and also temporary cache by the Spark engine.

The source data is taken from Twitter [3], where the live API provides a realtime stream of tweets which match any of a set of keywords for topics of interest. Each tweet record contains the user making the tweet and any users mentioned in the text of the tweet. The ETL phase consists of parsing the JSON record, then re-organizing into structures analgous to relation tables (tweets, users, relationships, hashtags). To create the communities, we choose to look at a network of user pairs that have mentioned each other. This reduces some of the celebrities who are mentioned a lot, but mention relatively few. This network is fed into a Spark-GraphX community detection algorithm. Then, using both aggregations of the source data and the communities, the important communities and their characteristics can be described: topics, size, density, user roles, and relationships between communities. These results are stored in HDFS in CSV format, to later be loaded into a visualization tool for interactive analysis. Web-based D3.js is used, but desktop alternatives include Cytoscape, Spotfire, Tableau and many others.

A Spark program is written in a functional style, allowing the engine to make efficient use of the full cluster, lazily, in parallel, with failure recovery, but without the user having to code for such complexity. Rather than writing loops, the functions of the pipeline are specified in sequence, using familiar dataset operators like map, filter, distinct, join, and group-by. A Spark Streaming pipeline is written in the same style, but will also process periodically.

A. Goals of the Project

The first goal of the project is the business use case, to demonstrate analytics of socia media data, infer communities, and identify users that exhibit certain patterns or roles.

The second goal was to bring together various technologies to demonstrate the architecture of an end-to-end solution.

In addition, the application is used by Cray R&D for future product development and QA for testing Urika-XA.

II. Architecture

One goal of this project was to demonstrate what a production architecture might look like beyond just loading data and finding results. With more common use of Hadoop and related technologies in recent years, more organizations are performing big data analytics and discovering the challenges of maintaining systems with multiple data-processing workflows.

The Urika-XA system, with its central role for Hadoop and Spark, minimizes the maintainence and configuration for a cluster, but requires solution developers to plan for architectural requirements such as data consistency and latency. Below, we discuss some of the popular ideas for the architecture of such systems.

A. Lambda Architecture

The Lambda Architecture described by Nathan Marz [4], is a system to handle both batch and stream data processing, in a way that balances latency, throughput and fault-tolerance.

Data is stored in an append-only, immutable, system of record. The data is structured as timestamped events, so can be appended rather than overwriting previous data.

The batch layer processes large quantities, aims for completeness and accuracy, and may have large latency. Hadoop/MapReduce is the de facto batch system.

The speed layer processes real-time streams with minimal latency, but may sacrifice completeness or accuracy. The results are replaced when the batch layer completes.

The serving layer responds to queries on the pre-processed data.

B. Kappa Architecture

In 2014, Jay Kreps [5] and Martin Kleppmann [6], after having built systems according to the Lambda Architecture, offered criticisms and described some improvements.

With the Lambda Architecture, the batch and speed layers are written in different frameworks, so many algorithms must be maintained in duplicate.

Using the timestamped event nature of the Lambda data, and noting that traditional databases use this same structure internally for transaction logs, Kappa describes processing the batch layer with a streaming technology that can handle both batch and speed. This still requires duplicate processing of the data, but eliminates multiple framewords and duplicate code.

Everything is viewed as a stream of data, from the storage, batch and speed layers, as well as serving and visualization.

C. Apache Spark

Spark is a data processing engine which provides a functional API to execute algorithms on big data across a cluster. The core abstraction is the RDD, resilient distributed dataset, a collection of data items, which may be any Scala object (or Java or Python). The RDD is analagous to a relational table, where the objects are rows. A program may work with multiple RDD's each with a different type of object. An RDD may be transformed into multiple different RDD's, or multiple combined into one. Transformations are familiar set concepts: map, filter, distinct, keyBy, groupBy, reduce, union, join, leftOuterJoin, etc.

The code that invokes those transformation methods does not directly or immediately cause that work to be executed. Instead, it's setting up a graph of jobs to be executed on remote worker nodes as needed. Only when output, such as saving results to a file or database, is requested are the transformations executed. Also, the transformations are only executed on the data items required by the output.

The RDD abstraction can handles datasets that are small or very large. The engine will automatically partition the data, or this can be overridden by the application. The application can also control if the data should be cached in memory or on disk, to avoid the cost of re-computing.

Spark Streaming builds on the core RDD API with the concept of a discretized stream or DStream. This offers a way to process micro-batches in windows of time. For each window, operations on the DStream are executed. The API for a DStream is mostly the same as for an RDD, with transformations like map, join, etc. Internal to a DStream, the operations are executed on one or more RDD.

A DStream can also have a sliding window. Say the batch window is 1 hour and the slide duration is 10 minutes, the operations will be executed every 10 minutes on the last 1 hour of data.

It is the functional style of programming that make this API and engine possible. Many of the parameters to the RDD and DStream methods are functions (called UDF, user-defined functions, in the context of SQL or other databases with a remote query capability). These functions must be pure functions, deterministic and stateless so the output depends only on the input parameters, and free of side-effects, so not modifying any external state such as member data on some object. First, this allows the Spark engine to invoke the function on different machines and JVM's because only the function matters, not any context or state. Second, this allows the functions to be invoked independently of each other, so a particular data value may flow through the entire pipeline before another value is even read from disk, or functions may be invoked multiple times to attempt to recover from errors. The Spark engine may provide more features because the application code is constrained to independent functions, and the application logic is simplified because it is constrained to well-defined data operations rather than complex state machines and syncronization points.

Spark is itself implemented in the Scala programming language, and the API most strongly supports Scala, though Java and Python are also supported. The Scala language compiles to Java bytecode to be run on a JVM (Java Virtual

Machine). It is similar to Java in style of syntax, being object-oriented and statically typed. Scala differs from Java in also supporting functional programming, immutability, type inferencing and other features that make code more concise and high-level. After experimenting with using Spark with the Java language, our conclusion was the type inferencing of Scala, and concise syntax for functions make Scala easier to understand for Spark pipelines.

III. Implementation

We chose to implement the SNA (Social Network Analytis) pipeline on Urika XA with Apache Spark. The source data from Twitter is naturally streaming and ongoing. The results desired are both real-time and some that will require high-volume processing. Spark and its modules GraphX and Streaming include the features required for the various analyses. We chose to follow the principles of the Lamba and Kappa Architectures, both to achieve a consistent system and to demonstrate a production architecture.

A. Streaming Data From Twitter

Data was collected for this project by connecting to the Twitter HTTP/REST API. The download is persistent and re-establishes in case of error, so runs for months without intervention. The full Twitter firehose is about 600M tweets per day. This free account only recieves about 300,000 tweets per hour, so about 1% of the firehose. The tweets downloaded are based on a list of search keywords related to various topics: sports, consumer electronics, and life sciences.

B. Data Storage

The java process that downloads from Twitter recieves the tweets as JSON records, and simply appends each to a text file, periodically (1 hour) closes the file, gzips it and moves it to a permanent location and starts a new file.

This structure allows subsequent processes to reprocess historical data.

C. Streaming Data Load

As described above, the data is downloaded from Twitter in a streaming fashion and is naturally stored as timestamped events appended to the repository, so is consistent with the Lambda and Kappa Architectures.

The software is written to start processing at a given starting timestamp and stream forward. When processing from the stored files, it checks the modification time of the file (which is trusted to be accurate).

In addition to loading past data from files, the system can connect to Twitter to download live tweets and process them in real time.

D. Speed Layer or Fast Lane

Using Spark Streaming, loading the data and performing aggregations can be done with a very small window, such as a few seconds. These aggregations include counting the tweets, unique users and hashtags. We also track most popular hastags within per user and globally.

Any errors are exported to a file to be inspected.

For the sake of presentations, tweets are censored out based on matching a list of keywords.

We also include some Complex Event Processing (CEP), such as spotting trending hashtags or pairs of hashtags.

The small window timeframe and low latency allows the speed layer to find and produce output for events in near real-time.

E. Batch Layer or Slow Lane

The batch layer operates with a larger window timeframe to collect more data, and has higher latency than the speed layer.

The community detection algorithm that is used, Label Propagation, is not implemented in a streaming fashion. Each invocation of the algorithm processes an input dataset to produce output. The next invocation is not directly related to the previous.

The larger the input dataset, the higher quality the output. That is, more input will contain more relationships on which to build the communities.

In addition, a larger input dataset takes progressively longer to process, because a greater number of relationships leads the algorithm to need more iterations to resolve or stabilize.

The input to the community detection is a network of user relationships - also called an edge list of users. A tweet has a creator and may mention zero or more other users. For the purpose of this analysis, we chose to further restrict the network to users that have mentioned each other (within the window), infering they “know” each other. This makes for a stronger set of communities, and minimizes the effect of celebrities that are mentioned by thousands.

Once the community detection is complete, the result is just a collection of edges assigning users to some number of groups, where the groups are nothing more than an identifier. Therefore, we follow that with a number of aggregations and operations to collect more information about the communities.

First are aggregations such as counting the number of users in each communities. We also count the number of mentions relationships between users in the same community. The density is the proportion of relationships to the size of the community.

It is useful to filter out some communities. Those with too few users (less than 10) have too little value. Those with too many users (hundreds or thousands, depending on the input size) are considered to indistinct to be of high value. Those

with too low density will have low value as they are likely users with low rates of communication to anyone.

Next, communities are characterized by aggregations on the members, such as finding the most popular hashtags and topics.

Due to the nature of the algorithm requiring substantial input, and the latency being higher than the streaming aggregations in the Fast Lane, the community algorithms run in a different streaming context, a slow lane.

With the quantity of data being collected (1% of the firehose), the community-related pipeline can handle 1 month or more of data on a Urika XA in less than one hour. It would be practical to use a batch window of 1 month with a sliding duration of 1 day, so that an end user (data analyst) would recieve new results every day based on the last months activity. If the full firehose were being processed, one day's worth of data would require a substantial amount of compute and memory.

F. User Roles

Once the communities have been created, we look more at the relationships between users within each community. Some users originate content which is retweeted by others. Other users mostly retweet content written by others. There is a spectrum between these two behaviors as well as a measure of breadth of such relationship to other users.

G. Community Relationships

Users are assigned to a single community, but will often have relationships to users in other communities. Some users will have strong ties to multiple communities, and when the measure of those ties is balanced, they are considered strong "connectors" between communities. Examples have been seen of sports commentators with ties to communities, where those have strong interests in various local teams, based on the popularity of multiple hashtags.

We also look at relationships between communities, by counting the users that mention or know users in the other.

IV. Conclusion

The Spark framework supports writing analytics pipelines and algorithms of substantial complexity without the limitations of a declarative framework. By organizing the application in a functional paradigm, the framework can distribute the code to a cluster while hiding most of that complexity from the application programmer. The functional approach also lets Spark organize the execution of the application functions for efficiency of memory, and re-execute to recover from errors. In this way, it achieves some of the goals of a declarative framework.

Acknowledgment

Ramesh Menon, Cray Inc, ramesh@cray.com
Venkat Krishnamurthy, Cray Inc, venkat@cray.com

References

[1] Cray Urika XA: http://www.cray.com/products/analytics/urika-xa

[2] Apache Spark: http://spark.apache.org/

[3] Twitter Data: https://dev.twitter.com/streaming/public

[4] Lambda Architecture: http://lambda-architecture.net/

[5] Kreps, Jay, "Questioning the Lamba Architecture", 7/2/2014, http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html

[6] Kleppmann, Martin, "Turning the database inside out with Apache Samza", 9/21/2014, https://youtu.be/fU9hR3kiOK0