Albatross Enhancements

Nikhil Brahmankar
nbrahman@hawk.iit.edu

Ketankumar Juneja
kjuneja@hawk.iit.edu

Shalin Chopra
schopr10@hawk.iit.edu

ABSTRACT

Data Analytics has become very popular on large datasets in different organizations. It is inevitable to use distributed resources such as Clouds for Data Analytics and other types of data processing at larger scales. Frameworks such as Hadoop and Spark, which are mainly designed for Big Data analytics, have been able to allow for more diversity in job types to some extent. However, centralized architectures of these systems become a bottleneck on large scales and under heavy task loads. In order to achieve high efficiency, scalability, and better system utilization, it is critical for a modern scheduler to be able to handle over-decomposition and run highly granular tasks. Albatross works great in order to address all the above mentioned issues.

Similar to other distributed systems, Albatross performance is likely to be limited by I/O and File operations. [2] The addition of Cache can help reduce theses bottlenecks keeping the latest data into memory and which reduces disk accesses. Hence we propose adding a caching layer over Albatross, which will help to improve Albatross performance by reducing the Albatross overheads of I/O and Data Locality by certain extent through minimizing the amount of unnecessary disk operation. Using our evaluations, we will prove that Albatross outperforms Spark and Hadoop at larger scales and in the case of running higher granularity workloads.

In addition, adding a Cache layer to Albatross will also help to support iterative applications using in-memory data storage similar to Spark. Modern day data analytics require multiple iterations to process and fine tune the data. Machine learning systems require multiple iterations to fine tune the training data results and apply them to testing data. Albatross currently supports only single iteration of Map Reduce. We propose development of iterative application over Albatross, thereby taking advantage of platform features and extending them to machine learning applications.

Author Keywords

Albatross, FusionFS, Distributed File System, Fabriq, ZHT, Data Analytics, Task Scheduling, Distributed Systems, Spark, Hadoop, Distributed Task Execution, Distributed Message Queue

ACM Classification Keywords

Data Analytics, Task Scheduling, Distributed Systems, Spark, Hadoop, Distributed Task Execution, Distributed Message Queue, Distributed File System

Paste the appropriate copyright/license statement here. ACM now supports three different publication options:

- ACM copyright: ACM holds the copyright on the work. This is the historical approach.
- License: The author(s) retain copyright, but ACM receives an exclusive publication license.
- Open Access: The author(s) wish to pay for the work to be open access. The additional fee must be paid to ACM.

This text field is large enough to hold the appropriate release statement assuming it is single-spaced in Times New Roman 8-point font. Please do not change or modify the size of this text box.

Each submission will be assigned a DOI string to be included here.

INTRODUCTION

The massive growth in both scale and diversity of Big Data has brought new challenges as industry expectations of data processing loads continue to grow. Data analytics frameworks such as Hadoop and Spark were proposed to particularly solve the problem of data processing at larger scales. These frameworks distribute the data on multiple nodes and process it with different types of tasks. However, both the above systems have bottlenecks as they use a centralized approach which limits scalability and load balancing. In addition, these systems actually push the tasks to Workers instead of let workers pulling them depending upon their current status. Therefore, these frameworks are not suitable for workloads that generate more tasks in shorter periods of times.

To overcome these shortcomings, Albatross was proposed. Albatross is a fully distributed cloud-enabled task scheduling and execution system that utilizes a Distributed Message Queue as its building block. Albatross uses a pull based approach rather than a central push based approach, which completely eliminates the single point of failure issue.

Albatross system currently lacks a way to effectively utilize memory and reduce disk operation. Addition of Caching mechanism will help Albatross to perform better for application which involve intensive data operations. Besides an effective memory management will also help Albatross to better store intermediate results of Map Reduce operation.

Data analytics operation are more iterative in nature which requires multiple Map Reduce at each iteration. Albatross currently does not support iterative Map Reduce operation. Albatross features like data locality and job movement could be extended to iterative application which would result in better performance.

Motivation

Map-Reduce scenario uses coarse-grained tasks to do its work, which are too heavyweight for iterative algorithms which require lightweight tasks. Another problem is that Map-Reduce has no awareness of its intermediate data to be stored in memory for faster performance. Instead, it flushes

intermediate data to disk between each Map and Reduce step. Combined, these sources of overhead make algorithms requiring many fast steps unacceptably slow. For these steps to be executed we want them to be as fast and lightweight as possible. Thus, to overcome the disadvantage of flushing intermediate results to disk we need a better memory management layer.

This issue is addressed by Cache. Similar to Cache Memory available to processor in Motherboard, Application Cache will help to store the frequently used data, intermediate data and input data within Memory to avoid multiple Disk Operations. This is one of the major reason Spark is having better performance than Hadoop.

Machine learning application which are iterative in nature are supported by various distributed frameworks like Spark. These applications take advantage of the current Map Reduce programing model and run their tasks as multiple Map Reduce for each iteration. Albatross uses the same Map Reduce paradigm and hence can support machine learning applications. The intermediate results are very important in case of iterative application. The addition of memory layer which keeps the intermediate results in memory will help the iterative application perform better. Thus, this will also help to increase the wide array of applications supported by Albatross.

To enable Iterative Application support, we planned to implement Logistic Regression in Albatross.

ALBATROSS ENHANCEMENTS

This section explains the changes done in Albatross to implement Cache Storage and Iterative Application support.

System Overview

Albatross' features like Load Balancing, Data Locality and overcoming the disadvantages of Centralized Scheduler bottlenecks helps it to outperform traditional Map Reduce System. Albatross coupled with a better memory utilization layer can boost its performance and help it support a variety of application without disk bottlenecks.

Caching

Caching in Albatross helps to fetch and store frequently accessed data with minimum possible disk accesses. In case of Map Reduce Systems, the intermediate data is often ignored and flushed onto disk. Thus, if a reducer requires an intermediate result from a mapper operation it has to make a disk access which is quite slower and hence the performance of the system is affected.

Caching in Albatross uses a Least Recently Used (LRU) based replacement scheme for memory management. This method was chosen because in case of Iterative Map Reduce model, typically the intermediate data from earlier iterations except the latest one may not be required in the future operation. This saves a lot of space and helps keep only the latest data in memory.

Caching Architecture

The figure below explains the Caching Architecture in Albatross.

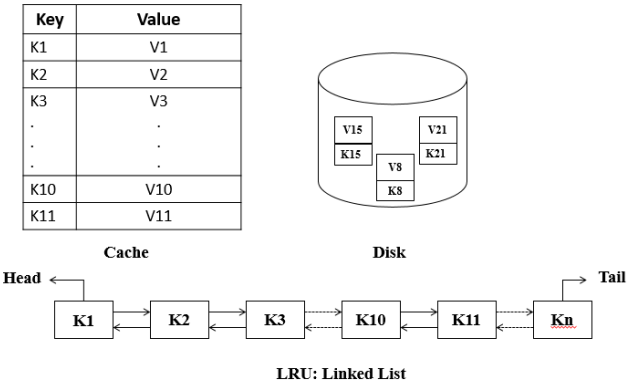

Figure 1: Caching Architecture

Caching is implemented with a simple Hash Map coupled with a Linked List data structures. Linked List maintains the latest accessed records from memory. The Hash Map helps to access data with a look up time complexity of O(1). The Hash Map uses a key value based storage. The Linked List structure is used to keep track of recently used keys. The Linked List structure stores only the key of latest accessed data. The most recently accessed key is placed at the head of the Linked List. Whenever the cache is full the tail node is evicted and a new node is created and inserted at the head of the Linked List depending upon the size of available Cache and the data to be inserted in Cache.

Caching Decision Making Flow

The typical Caching decision making flow is explained below.

- If value corresponding to Key already exists in Disk
  - Read existing Value from file on Disk
  - Store the read value in Cache and return it
- If the value corresponding to Key doesn't exist in Disk, then check if value corresponding to Key already exists in Cache. If yes then
  - Read existing Value from cache and return it
  - Move the node corresponding to latest accessed Key to head of LRU Linked List
- If the value corresponding to Key doesn't exist in Disk and Cache both, then check if sufficient space is available in Cache to store the Key – Value pair. If yes then,
  - Store existing Value in cache and return it
  - Insert the node corresponding to latest accessed Key to head of LRU Linked List
- If sufficient space is unavailable in Cache, then check if LRU is enabled in Albatross. If LRU in disabled then,
  - Store the Key – Value pair in Disk

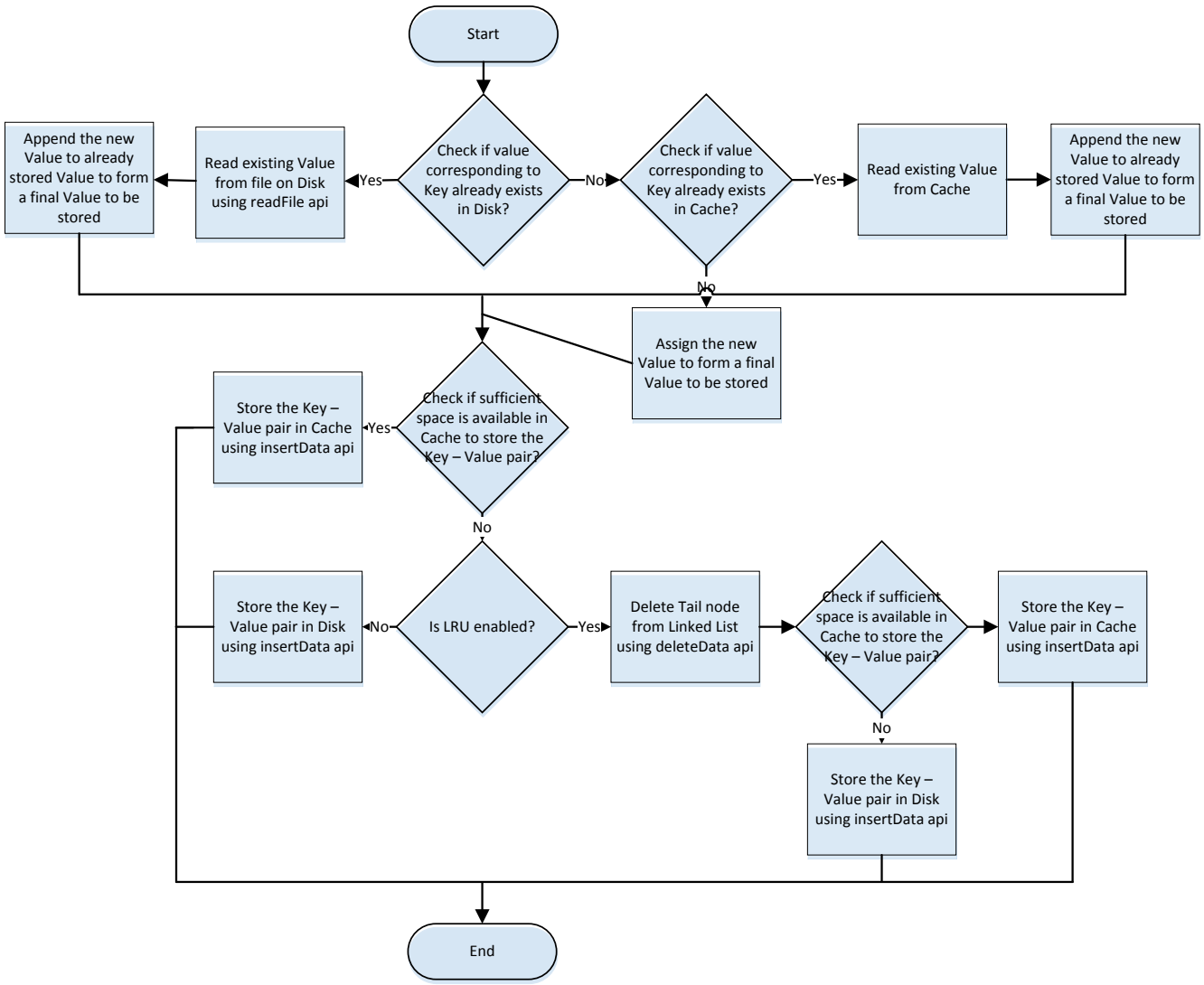

Figure 2: Caching decision making flow

- If LRU is enabled then,
  - Delete Tail node from Linked List
  - If sufficient space is unavailable in Cache, then
    - Store the Key – Value pair in Cache
  - If still there is no sufficient space in Cache, then
    - Store the Key – Value pair in Disk

Logistic Regression

This is a classification algorithm in which we predict / classify the data to its appropriate class label. It is one of the most popular and most widely used learning algorithms today. Logistic Regression is used for predicting the probability of occurrence of an event by fitting the data to a logistic curve. For Albatross, we have focused on the binary classification problem in which there are only two classes having values, 0 and 1.

For Logistic Regression to be implemented we have some parameters which play an important role in classification. These are:

- The weight (parameter): θ
- Learning Rate: η

These two parameters help for convergence of the algorithm and generate a final weight parameter i.e. θ which is useful for further prediction.

We have implemented Logistic Regression using Stochastic Gradient Descent rule.

There are two equations which help to update the value of θ. This update occurs iteratively until maximum numbers of iterations are reached or convergence threshold is reached.

The hypothesis takes the form of:

$$\frac{1}{1 + e^{-\theta^T X}}$$

This is called as the logistic function or the sigmoid function.

The stochastic gradient descent rule is given below, which is used to update θ.

$$\theta = \theta - \eta \sum_{i=1}^{m} \big(h(\theta) - y(i)\big)x(i)$$

The Summation part is called as "gradient".

Architecture

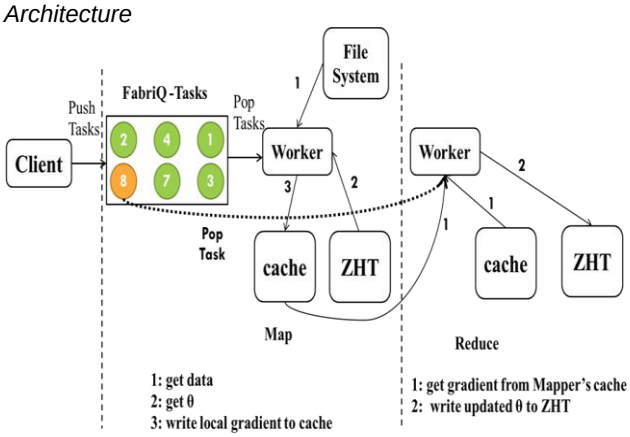

Figure 3: Iterative Application Architecture in Albatross

The figure above explains the Iterative application architecture in Albatross. Its based on the building blocks based architecture which uses components like FabriQ, ZHT and Caching mechanism as building blocks. In case of Logistic Regression, we have multiple Mappers and a single reducer.

Client

- Puts the tasks into FabriQ for the workers to fetch.
- These tasks are Map and Reduce tasks

FabriQ

- FabriQ is a queue which holds all the tasks generated by the client.
- It puts the tasks which have pcount =0, onto the ready queue which can be fetched by the worker, the other tasks are kept in non-ready queue which are available once pcount=0
- In case of logistic regression, for the first iteration the map tasks pcount = 0, while the reduce task pcount = no. of map tasks.
- For next iteration onwards, the map task has a pcount = no. of reducers and so on.

Map phase

- Once a map task is fetched, the mapper firstly gets the data from disk in case of iteration no. 1, from next iteration onwards the data is available in cache to be fetched.
- After data is fetched the worker get the value of θ from ZHT and starts computing the local gradient and writes it back to its own cache.
- Every mapper calculates its local gradient and stores it in cache for the reducer to fetch.

Reduce Phase

- The reducer gets local gradient from all the mappers and sums the up to forma global gradient.
- Then using the update equation, it updates the value of θ, and writes this updated value of θ into ZHT, for next iteration mappers to fetch.

EVALUATION

This section evaluates Albatross by measuring its throughput and latency using weak scaling mechanism. We have used Logistic Regression application to evaluate Albatross' support for Iterative Applications. We executed 50 iterations for Logistic Regression during evaluation. Each iteration is combination of Map phase executing multiple mapper tasks and Reduce phase executing a single reduce task. Mapper and Reduce tasks are distributed to workers using FabriQ queue. Each worker is setup with its own Cache to store the input and gradient data into memory for faster performance. The iterative application is run by keeping the data size constant per node (weak scaling). The size for input training data is 5GB per node. We have used twenty features for classification in logistic regression. We have benchmarked Albatross' performance with Spark using the same dataset and same node configuration. The parameters considered for benchmarking are throughput, latency and average time per iteration. The following graphs explains the comparison of the referred parameters between spark and Albatross

Testbed Configuration

Albatross is developed in C++. All experiments use Amazon EC2 [5] m3.xlarge instances. These instances use High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge or Sandy Bridge) Processors with hyper-threading, SSD-based instance storage for fast I/O performance and they are balance of compute, memory, and network resources. Each of these instance offers 4 cores, 15 GB RAM and SSD storage up to 80 GB.

The graph in figure 4 shows that the throughput of Albatross and Spark is quite similar even with the increase in the scale. Initially, for small number of nodes the throughput is the almost the same. It can also be seen that as the number of nodes increases the throughput increases with the constant and consistent scale for both Albatross and Spark. Spark seems to slightly better than Albatross with increase in scale.

Throughput

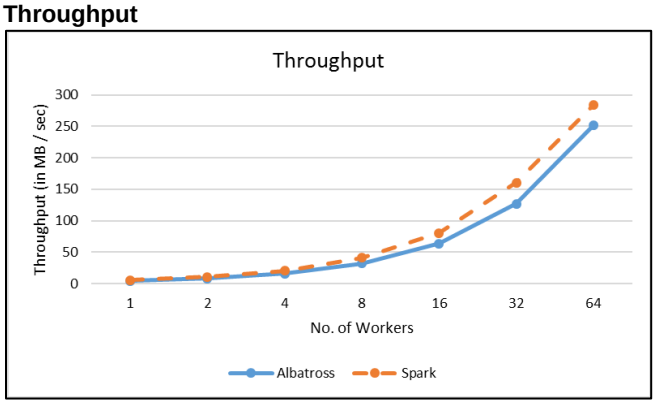

Figure 4: Throughput comparison between Albatross and Spark

Latency

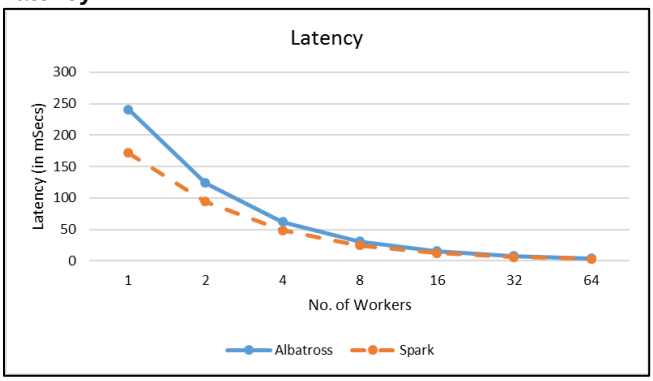

Figure 5: Latency comparison between Albatross and Spark

The graph in figure 5 above has similar trend. The initial value for latency is quite high and it decreases with increase in number of nodes. The trend is the same for both Albatross and Spark, but Spark performs much better than Albatross.

Average Iteration Time

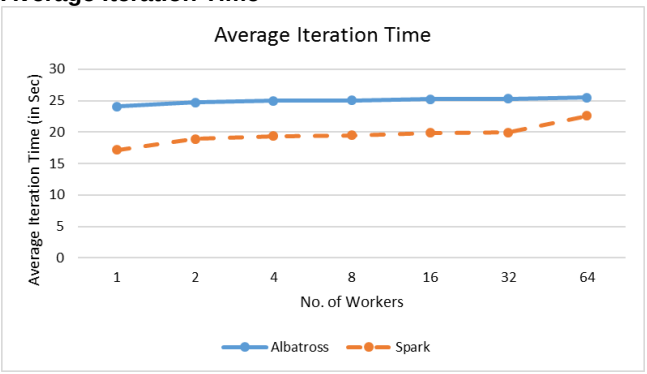

Figure 6: Average Iteration Time comparison between Albatross and Spark

The graph in figure 6 above shows the average time taken for a single iteration of Logistic Regression. The average iteration time is calculated by dividing the total time with the number of iteration. Albatross' average iteration time is constant and is higher than Spark initially. However, as the number of worker nodes and data volume is increased, Spark's average iteration time starts increasing and approaches that of Albatross, the trend shows that if the scale is increased further, Spark may even take more time to complete an iteration than Albatross.

Evaluation Summary

Summary for our analysis and comparison between Albatross and Spark is listed below.

- Spark's better throughput is achieved only if Spark's library function for Logistic Regression is used which is highly optimized. In case if we try define our own Mapper and Reducer for Logistic Regression then same efficiency will not be achieved
- On a larger scale, the latency for Albatross and Spark goes towards zero
- The difference between Albatross and Spark Throughput can be explained as time required for multiple conversions (like string to vector, string to matrix, matrix to string, vector to string, etc.) required to communicate the data between various mappers and reducers
- As discussed earlier, average iteration time for Spark will increase further as an effect of increased scale
- In addition, Albatross serves its main purpose of avoiding bottleneck of centralized scheduler

CONCLUSION & FUTURE WORK

Machine learning applications like Logistic Regression being very compute intensive and requires in-memory data to perform computations. Logistic Regression being iterative in nature requires the intermediary data from previous computations to be stored in memory. Hadoop on the other hand flushes this intermediary data onto disk which causes it to be slow when compared with Spark and Albatross. Albatross enhanced its architecture to support these in-memory CPU bound computations by adding Caching concept. The in-memory cache helps to store the intermediary results in memory and helps in performing faster computations.

Our evaluation proves Albatross to be a better choice for a larger scale system with better performance and flexibility. We can always consider difference in throughput as trade-off between centralized (Spark) and distributed (Albatross) scheduling system.

Furthermore, improvements like Distributed Cache, avoiding multiple data conversions for communicating the data across nodes can definitely improve Albatross performance and make it a better choice over Spark.

Link to the repo: https://bitbucket.org/schopr10/albatorss-iterativeapplication

ACKNOWLEDGMENTS

We gratefully acknowledge the support and guidance of our advisor Prof. Ioan Raicu and Iman Sadooghi. Without their thoughtful encouragement, supervision and guidance, this research would never have taken shape. We are grateful to them for providing us their timely feedbacks and showing us the correct directions to make this research a useful one. We will also like to thank Sami Ahmad Khan for supporting us in Spark evaluation and share its results.

REFERENCES

1. Iman Sadooghi, Geet Kumar, Ke Wang, Dongfang Zhao, Tonglin Li, Ioan Raicu. "Albatross: an Efficient Cloud-enabled Task Scheduling and Execution Framework using Distributed Message Queues"
2. Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. "FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems", IEEE International Conference on Big Data 2014; 18% acceptance rate
3. Iman Sadooghi, Ke Wang, Dharmit Patel, Dongfang Zhao, Tonglin Li, Shiva Srivastava, Ioan Raicu. “FaBRiQ: Leveraging Distributed Hash Tables towards Distributed Publish-Subscribe Message Queues”, IEEE/ACM BDC 2015
4. T. Li, X. Zhou, et. Al. “ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table,” in Proceedings of the IEEE IPDPS, 2013.
5. Amazon EC2 Instance Types (https://aws.amazon.com/ec2/instance-types/)
6. Andrew Ng - CS 229 (http://cs229.stanford.edu/notes/cs229-notes1.pdf)
7. Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, Ion Stoica “PACMan: Coordinated Memory Caching for Parallel Jobs”