Getting Your Big Data Priorities Straight: A Demonstration of Priority-based QoS using Social-network-driven Stock Recommendation

Rui Zhang, Reshu Jain, Prasenjit Sarkar and Lukas Rupprecht*,[1]
IBM Research - Almaden, *Imperial College London
{ruiz, jainre, prsarkar}@us.ibm.com, lr12@imperial.ac.uk

ABSTRACT

As we come to terms with various big data challenges, one vital issue remains largely untouched. That is the optimal multiplexing and prioritization of different big data applications sharing the same underlying infrastructure, for example, a public cloud platform. Given these demanding applications and the necessary practice to avoid over-provisioning, resource contention between applications is inevitable. Priority must be given to important applications (or sub workloads in an application) in these circumstances.

This demo highlights the compelling impact prioritization could make, using an example application that recommends promising combinations of stocks to purchase based on relevant Twitter sentiment. The application consists of a batch job and an interactive query, ran simultaneously. Our underlying solution provides a unique capability to identify and differentiate application workloads throughout a complex big data platform. Its current implementation is based on Apache Hadoop and the IBM GPFS distributed storage system. The demo showcases the superior interactive query performance achievable by prioritizing its workloads and thereby avoiding I/O bandwidth contention. The query time is 3.6× better compared to no prioritization. Such a performance is within 0.3% of that of an idealistic system where the query runs without contention. The demo is conducted on around 3 months of Twitter data, pertinent to the S & P 100 index, with about 4×10^{12} potential stock combinations considered.

1. INTRODUCTION

Despite increasing focus on big data technologies, little has been done with regard to providing Quality of Service (QoS) guarantees to applications and users. Indeed the comment “One of the next frontiers of Hadoop performance is QoS (Quality of Service) [...].” [11] remained a prominent message in Hadoop forums as late as 2013. This demo addresses one fundamental QoS aspect, the capability to *prioritize applications* when resources are stretched.

Big data applications are often hosted in an environment shared with other applications in order to minimize infrastructure, data duplication and management costs. It is without surprise that major public cloud providers such as Amazon provide MapReduce capabilities (EMR) and shared storage (S3) to support big data applications. However, this sharing creates complex resource interference and contention, making it difficult to provide performance guarantees for high priority applications or more important sub workloads within a single application. This is especially problematic for near real-time decision making applications such as stock purchases or personalized recommendations. These applications require fast responses to users, which can not be achieved without intelligent prioritizing throughout the environment due to resource contention.

The aforementioned QoS capability entails addressing two problems: classification and protection. Any big data application has to be assigned a class and identified consistently in a multi-tier, distributed big data environment. Protection is then provided to the properly identified workloads, using actuators within different resources such as CPU, memory, cache, storage and/or network. Big data applications typically cross many distributed software and hardware components. A typical big data stack may consist of a query engine such as Hive [6], a multi-node parallel computing platform such as Hadoop [9], a multi-node distributed storage system such as GPFS [5] and other resources, e.g., the network.

Additionally, a shared infrastructure such as a public or private cloud will often not only host a single big data system, but rather multiple, different systems [2]. These systems might have individual components on higher layers but are built on top of shared components. Consider a cluster setup as depicted in Figure 1. The distributed file system is shared by the processing engines while the engines themselves are shared by higher level libraries. For example, as shown in Figure 1 Hive and Mahout (a machine learning library for Hadoop) [10] share the Hadoop instance. Ensuring QoS *across* these components is difficult as top-level components are independent of each other. If prioritization only happens at those layers, the enforced priorities will get lost as soon as the shared layers are reached because there is no common notion of priorities.

To ensure QoS for all hosted applications across all systems, a coordinated *end-to-end* solution for classification and protection must be in place throughout the entire en-

[1] Work done during internship at IBM Research.

Proceedings of the VLDB Endowment, Vol. 7, No. 13
Copyright 2014 VLDB Endowment 2150-8097/14/08.

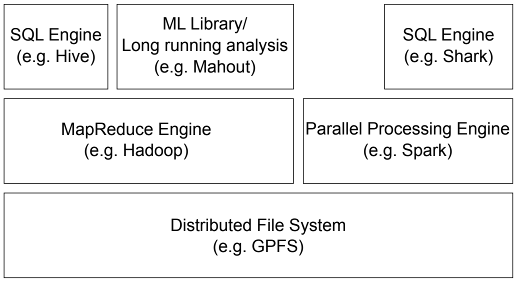


Figure 1: Multiple big data systems running in a shared environment.

vironment. If application priorities are passed from layer to layer, shared layers can act accordingly and maintain QoS policies. Consider the example stack from Figure 1 with Shark [8] queries having higher priorities than Hive queries having higher priorities than Mahout jobs. Mahout is usually used for longer-running analysis while Hive and especially Shark are designed for interactive exploration. With an end-to-end QoS mechanism, Hive and Mahout jobs are prioritized at the Hadoop and file system layer while Shark and Hive queries are prioritized at the file system layer. This guarantees the desired performance for the different systems.

Existing work falls short of guaranteeing the same. Firstly, many big data infrastructure components are emerging systems still being stabilized. QoS capabilities are lacking or even non-existent. For example, the community behind Hadoop only recently started to plan for QoS support [11]. Secondly, legacy QoS solutions do not readily apply to these new infrastructures. There are various approaches regarding legacy systems concerning job scheduling, data storage, networking and some limited combinations of these [1, 3, 7]. Effective they may be within their confines, they lack the end-to-end correlation needed for big data applications. For instance, a storage QoS solution may be able to provide QoS to any I/Os in data storage. However, one must know which I/Os belong to which jobs, applications and/or users, for the right I/O control to be performed. A storage QoS solution alone can not provide that correlation.

This paper demonstrates a solution that bridges the very gaps above. Our research contributions are as follows:

- We provide the design of an end-to-end QoS solution for prioritizing big data applications,
- we present a specific implementation based on an infrastructure consisting of the commercial grade big data components Hive, Hadoop and IBM GPFS,
- we demonstrate our approach using a big data application that makes stock purchase recommendations based on analyzing around 3 months of live Twitter data. We show how prioritizing the interactive recommendation query over periodic twitter sentiment analysis can ensure timely decision making, despite intense I/O bandwidth contention in distributed storage.

2. OVERALL DESIGN

We consider running an application as the equivalent of executing tasks on different resources. A *task* can be fetching cached query results, executing a mapper/reducer, issuing I/O requests to storage etc.

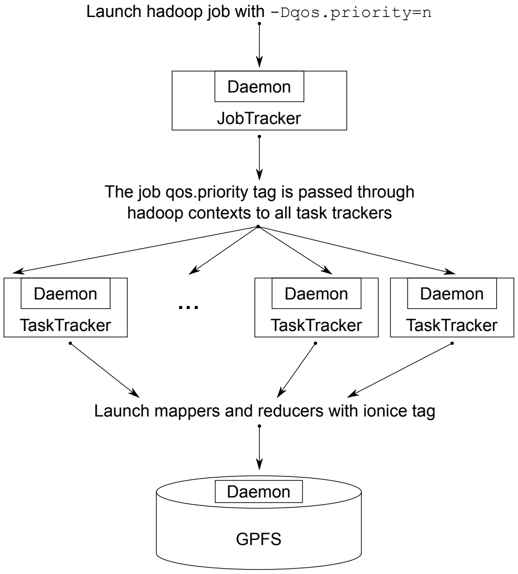

Launch hadoop job with -Dqos.priority=n
Daemon
JobTracker
The job qos.priority tag is passed through
hadoop contexts to all task trackers
Daemon
TaskTracker
...
Daemon
TaskTracker
Daemon
TaskTracker
Launch mappers and reducers with ionice tag
Daemon
GPFS

Figure 2: A prototype for Hadoop over GPFS

The key principle of our solution is *end-to-end* QoS (e.g. via application prioritization). The framework we put forward consists of inter-communicating daemons that are carefully planted into each type of required resource. Each daemon is a long-running program with two missions: classifying tasks and prioritizing tasks.

Classification involves assigning a priority and/or other QoS information to a task on a resource and forwarding the classification to other daemons downstream. For instance, a daemon in Hadoop tags a set of Hadoop MapReduce tasks as important. It then forwards that tag to the daemon in storage for further classification. It is through such forwarding that consistent classification is achieved across all the shared resources.

Prioritizing tasks entails favoring them in resource sharing according to their classification. This could mean placing a job at the front of the Hadoop scheduling queue, giving more CPU cycles to a MapReduce task, higher I/O bandwidth to an I/O stream, and/or setting the QoS fields in TCP/IP accordingly.

3. PROTOTYPE IMPLEMENTATION

Figure 2 illustrates our prototype for a big data platform comprising Hadoop and IBM GPFS. Note that GPFS replaces HDFS as the distributed storage behind MapReduce. The users/administrators specify the priority of an application via the Hadoop command line or configuration files. The goal of the QoS infrastructure is to pass the priority information from the job submission front-end through to the storage back-end. The MapReduce model employed by Hadoop spawns a large number of parallel processes known as mappers and reducers for each single job. Hence one key challenge in the implementation is to identify all these processes, to which we pass on the priority information, as the I/Os from these processes also have to be properly tagged.

As illustrated in the figure, the prototype features a daemon for the job tracker, a daemon for each task tracker (per node) and a daemon for the distributed storage system. In a nutshell, each type of daemon repeatedly receives priority information from upstream (another daemon or the user) and passes it downstream.

The job tracker daemon extracts the job priority information from the job submission interface and subsequently inserts it as a field into a Hadoop `JobContext` object. Most of this context (including the priority information) is copied to each task tracker.

Each task tracker daemon extracts the priority information and maps the priority into an I/O tag ranging from 1 to 7. A Hadoop task tracker launches all the mappers and reducers for the application on a node. The daemon intercepts the launch command and prefixes it with a `ionice` command and the I/O tag using this template: `ionice -n io_tag java Mapper/Reducer.class`. The Linux/Unix OS automatically inserts the ionice tag into a kernel structure for each I/O.

The storage daemon retrieves the ionice tag from the kernel structure for each I/O. It then prioritizes important applications by rate-liming any less important I/O stream whose aggregate I/O rates exceed its allocated bandwidth.

4. DEMONSTRATION

In this section we introduce the example application which we use in our demonstration in detail, present the demo design, and provide some preliminary evaluation results.

4.1 Application

Investors have overly abundant (confusing) choices of what stocks to purchase. Effective indicators to help narrowing down their choices quickly are thus tremendously valuable. It has been demonstrated that Twitter sentiments are strong indicators of pending stock market movement (see Figure 3) [4]. Herein, we base our demo on an application that leverages this very correlation. The application recommends the most promising stock portfolios (i.e. one or more stocks), whose corresponding tweets have the most positive sentiment. Positive tweets can include those praising the company's strategies or performance results, and predicting a rise in its stock price.

As illustrated in Figure 4, the demo application conducts the following tasks: it crawls tweets related to companies in the S & P 100 index (WL1), it periodically runs a Hadoop job to compute an aggregate sentiment score for each possible portfolio of up to 10 stocks, some 4×10^{12} potential portfolios (WL2), and, using Hive, it ranks the stock portfolios to recommend those whose latest (e.g. last hour) sentiment scores are the highest or within a user-specified range (WL3). WL2 and WL3 are the workloads of particular interest to our demo and we elaborate them further:

- The Sentiment Compute Batch Workload (WL2): A periodic, long-running batch workload computing the latest hourly sentiment for each portfolio. Each tweet is labeled positive (1), neutral (0) or negative (-1) using a trained Bayesian classifier. Each portfolio (e.g. \$MSFT + \$GOOG + \$IBM + \$YHOO) gets a summary sentiment $[-1, 1]$ based on the sentiment of its individual tweets.

- The Stock Portfolio Query Workload (WL3): Consists of a Hive query for the top k most promising portfolios and recommending these to the user. The query is:

```
SELECT * FROM portfolios ORDER BY sentiment
LIMIT k
```

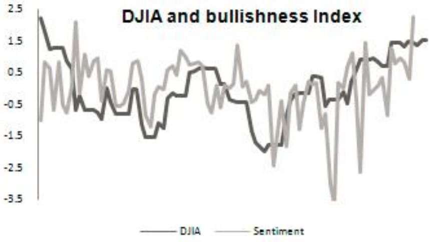


Figure 3: Correlation between twitter sentiment and the Doug index

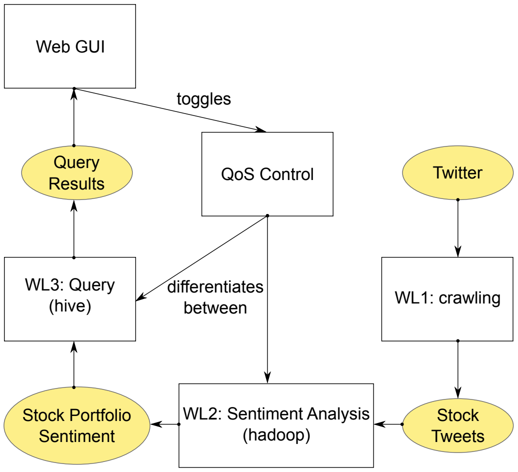


Figure 4: The demo architecture

Among other resources, these two workloads most eminently compete for storage bandwidth as they both rely on the sentiment score table. As the Hive query is needed for fast decision making, this is the workload we aim to prioritize and protect during I/O bandwidth contention.

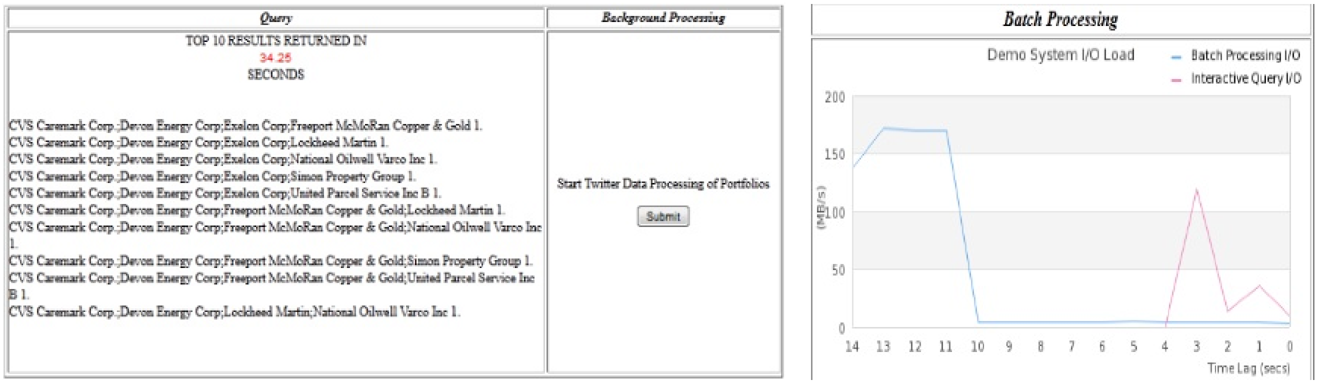
4.2 Goals and GUI Design

The demo showcases: (1) The high impact of QoS control (i.e. workload prioritization) on the performance of the interactive workload and (2) live "behind-the-scene" system activity when workload prioritization takes place.

The demo GUI is a single PHP page featuring 3 scenario planes with 2 side-by-side windows in each plane. The first window of a plane (see Figure 5(a)) is capable of launching jobs and displaying the results while the second window shows the I/O activity of the different workloads (see Figure 5(b)). The 3 scenario planes allow for a clear single-screen comparison view that highlights the QoS impact from a macro (query completion time) and micro (I/O activity) perspective. Via the side-by-side windows, we can observe the interaction of I/Os from the two workloads for the different scenarios and the effect of prioritizing bandwidth allocation for the query.

4.3 Demo in Action

The demo resides in a small cluster with Apache Hadoop 1.1.2 and IBM GPFS 3.6. Each node is equipped with 8 Intel Xeon(R) 2.5 GHz CPUs, 8 GB of memory, and a 250 GB hard disk and runs RedHat Linux.



Query

TOP 10 RESULTS RETURNED IN
34.25
SECONDS

CVS Caremark Corp.;Devon Energy Corp;Exelon Corp;Freeport McMoRan Copper & Gold 1.
CVS Caremark Corp.;Devon Energy Corp;Exelon Corp;Lockheed Martin 1.
CVS Caremark Corp.;Devon Energy Corp;Exelon Corp;National Oilwell Varco Inc 1.
CVS Caremark Corp.;Devon Energy Corp;Exelon Corp;Simon Property Group 1.
CVS Caremark Corp.;Devon Energy Corp;Exelon Corp;United Parcel Service Inc B 1.
CVS Caremark Corp.;Devon Energy Corp;Freeport McMoRan Copper & Gold;Lockheed Martin 1.
CVS Caremark Corp.;Devon Energy Corp;Freeport McMoRan Copper & Gold;National Oilwell Varco Inc 1.
CVS Caremark Corp.;Devon Energy Corp;Freeport McMoRan Copper & Gold;Simon Property Group 1.
CVS Caremark Corp.;Devon Energy Corp;Freeport McMoRan Copper & Gold;United Parcel Service Inc B 1.
CVS Caremark Corp.;Devon Energy Corp;Lockheed Martin;National Oilwell Varco Inc 1.

Background Processing

Start Twitter Data Processing of Portfolios

(a) The job launching and result window

(b) The I/O activity window

Figure 5: Screenshots of the demo GUI

Scenario	Query completion time
Showpiece Scenario, QoS off	126.13 s
Showpiece Scenario, QoS on	34.25 s
Baseline Scenario	34.17 s

Table 1: Query completion times for the three different scenarios

We highlight the impact of prioritizing the query workload over the sentiment batch workload using three scenarios for comparison: (a) the *showpiece scenario* in which both workloads are running and QoS is enabled, (b) the *baseline scenario* with both workloads but without QoS, and (c) the optimal *upper-bound* scenario where the query workload is running alone.

Assume that a user wants to retrieve the 10 portfolios comprising 4 stocks with the highest sentiment score. The simultaneously running batch and interactive workloads cause I/O bandwidth contention. As shown in Table 1, this causes a very high query response time in the case of no QoS. When QoS is enabled, the query completion time is reduced significantly by 3.6×, due to the difference prioritizing makes in alleviating the negative impact of contention. This completion time is also within 0.3% of the optimal scenario where the query is running alone on the system and has dedicated access to all resources.

Figure 5(b) demonstrates the reason for the above observations. When the Hive query is launched to query the sentiments for the last full hour, the long-running batch workload is often already running and computing the new sentiments. As shown, the batch workload is heavily consuming I/O bandwidth. The drop of I/O consumption marks the effect of QoS, when the Hadoop job for the query is started. The drop happens before any actual query processing in order to prioritize the startup phase and later I/O processing of the query. Once startup is complete, the query is processed and enjoys as much bandwidth as it needs, while the batch I/Os are limited.

Note that the improvement factor of QoS over the scenario without QoS is somewhat constrained by the rather limited memory and CPU cores the current demo cluster has. The lack of memory and CPU cores restricts the number of parallel MapReduce tasks. We are migrating the demo to a larger environment to support more parallel tasks which will cause more I/O contention and longer query times in the non-QoS case. We then expect to see even higher improvement factors when turning on our QoS mechanism.

5. REFERENCES

[1] Y. Georgiou. *Contributions for Resource and Job Management in High Performance Computing*. PhD thesis, LIG, Grenoble-France, 2010.

[2] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[3] J. Lu, J. Wu, C. Wu, Y. Li, X. Xu, and S. Deng. Qos control method of cloud storage system based on differentiated service, Aug. 31 2011. CN Patent App. CN 201,110,116,396.

[4] C. Oh and O. Sheng. Investigating predictive power of stock micro blog sentiment in forecasting future stock price directional movement. In *ICIS*, 2011.

[5] F. B. Schmuck and R. L. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[6] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB Endowment*, 2(2), 2009.

[7] C. Wan, C. Wang, Y. Yuan, H. Wang, and X. Song. Utility-driven share scheduling algorithm in hadoop. In *Advances in Neural Networks*, volume 7952 of *LNCS*. Springer Berlin Heidelberg, 2013.

[8] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD*, 2013.

[9] Apache Hadoop. `http://hadoop.apache.org/`.

[10] Apache Mahout. `http://mahout.apache.org/`.

[11] RPC Support for QoS. `https://issues.apache.org/jira/browse/HADOOP-9194`.