Adaptive MapReduce Scheduling in Shared Environments

Sougandhika Narayan Vinay M, Vinay A, Sagar R, Nithin S
Department of Computer Science and Engineering,
K S Institute of Technology, Bengaluru

Abstract— MapReduce task scheduler for shared environments in which MapReduce is executed along with other resource-consuming workloads, such as transactional applications. All workloads may potentially share the same data store, some of them consuming data for analytics purposes while others acting as data generators. This kind of scenario is becoming increasingly important in data centers where improved resource utilization can be achieved through workload consolidation and is specially challenging due to the interaction between workloads of different nature that compete for limited resources. The proposed scheduler aims to improve resource utilization across machines while observing completion time goals. Unlike other Map Reduce schedulers, our approach also takes into account the resource demands for non MapReduce workloads and assumes that the amount of resources made available to the MapReduce applications is variable over time. As shown in our experiments, our proposal improves the management of MapReduce jobs in the presence of variable resource availability,increasing the accuracy of the estimations made by the scheduler,thus improving completion time goals without an impact on the fairness of the scheduler.

Keywords—MapReduce, Scheduling, Distributed, Analytics, Transactional, Adaptive, Availability, Shared Environments.

I. INTRODUCTION

The traditional database RDBMS is capable of processing small and medium data but not large data .The proposed system introduces an efficient analytical engine using hadoop big data which has HDFS environment and MAPREDUCE as programming language. Robust and customizable planner engine & pluggable and reusable helper component is created to perform analysis for two problems weather data analysis and server logs analysis. RDBMS is incapable of processing large data sets, due to relations among tables. If everything is in one column problem is normalizations (no duplicate values, no null values)

Instead of running these services in completely dedicated environments, which may lead to underutilized resources, it is becoming more common to multiplex different and complementary workloads in the same machines. This is turning clusters and data centers into shared environments in which each one of the machines may be running different applications simultaneously at any point in time: from database servers to MapReduce jobs to other kinds of applications [1]. This constant change is challenging since it introduces higher variability and thus makes performance of these systems less predictable.

To solve the scalability problem, a secret polynomial based message authentication scheme was introduced in [3]. The idea of this scheme is similar to a threshold secret sharing, where the threshold is determined by the degree of the polynomial. This approach offers information-theoretic security of the shared secret key when the number of messages transmitted is less than the threshold. The intermediate nodes verify the authenticity of the message through a polynomial evaluation. However, when the number of messages transmitted is larger than the threshold, the polynomial can be fully recovered and the system is completely broken.

The Reverse- Adaptive Scheduler, that allows the integrated management of data processing frameworks such as MapReduce along with other kinds of workloads that can be used for both, transactional and analytics workloads. The scheduler expects that each job is associated with a completion time goal that is provided by users at job submission time. These goals are treated as soft deadlines as opposed to the strict deadlines familiar in real-time environments: they simply guide workload management. We also assume that the changes in workload intensity over time for transactional workloads can be well characterised, as has been previously stated in the literature [5].

Existing previous work on MapReduce scheduling involved estimating the resources that needed to be allocated to each job in order to meet its completion goals [6], [7], [8]. This naïve estimation worked fine under the assumption that the total amount of resources remained stable over time. However, in a scenario with consolidated workloads we are targeting a more dynamic environment in which resources are shared with other frameworks and availability changes depending on external and a priori unknown factors. The scheduler proposed in this paper proactively deals with dynamic resource availability while still being guided by completion time goals.

MOTIVATING EXAMPLE

Consider a system running two major distributed frameworks: a MapReduce deployment used to run background jobs, and a distributed data-store that handles transactional operations and serves data to a front-end. Both workloads share the same machines, but since the usage of the frontend changes significantly over time depending on the activity of external entities, so does the

availability of resources left for the MapReduce jobs. Notice that the demand of resources over time for the front-end activities is supposed to be well characterized [5], and therefore it can be predicted in the form of a given function f(t) known in advance.

II PROBLEM STATEMENT

We are given a cluster of machines, formed by a set of nodes N = {1,...,N} in which we need to run different workloads. We use n to index the set of nodes. We are also given a set of MapReduce jobs J = {1,...,J}, that has to be run in N. We usej to index the set of MapReduce jobs.

Each node n hosts two main processes: a MapReduce slave and a non-MapReduce process that represents another kind of workload. While MapReduce usually consists of a tasktracker and a datanode in Hadoop terminology, we summarize both of them for simplicity and refer to them as the tasktracker process hereafter. Similarly, The non-MapReduce process could represent any kind of workload but we identify it as data-store in this paper.

III PROGRAMMING MODEL

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of

the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.*Map*, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function. The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation.

The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

TYPES

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

map (k1,v1) ! list(k2,v2)
reduce (k2,list(v2)) !
list(v2)

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values. Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

Reverse-Adaptive Scheduler

The driving principles of the scheduler are resource availability awareness and continuous job performance management. The former is used to decide task placement on tasktrackers over time, while the latter is used to estimate the number of tasks to be run in parallel for each job in order to meet performance objectives, expressed in the form of completion time goals. Job performance management has been extensively evaluated and validated in our previous work, presented as the Adaptive Scheduler [6] [7]. In this paper we extend the resource availability awareness of the scheduler when the MapReduce jobs are collocated with other time varying workloads. One key element of our proposal in this paper is the variable Sfit, which is an estimator of the minimal number of tasks that should be allocated in parallel to a MapReduce job to keep its chances to reach its deadline, assuming that the available resources

will change over time as predicted by f(t). Notice that the novelty of this estimator is the fact that it also considers the variable demand of resources introduced by other external workloads. Thus, the main components of the Reverse-Adaptive Scheduler, as described in the following sections, are:

- Sfit estimator.
- Utility function that leverages Sfit used as a per-job performance model.
- Placement algorithm that leverages the previous two components.

A. Intuition

The intuition behind the reverse scheduling approach is that it divides time into stationary periods, in which no job completions are expected. One period ends and starts in instants in which a job completion time goal is expected. When a job is expected to complete at the end of a period, the scheduler calculates the amount of resource to be allocated during the period for the job to make its completion goal. If the available resources are not enough, the amount of pending work is pushed back to the immediately preceding period. Notice that the amount of the available resources for the period is determined by the function f(t), that estimates the resources that will have to be committed to the non-MapReduce workloads. When more than one job co-exists in the same period, they compete for the available resources, and they are allocated following a fairness criteria that will try to make all jobs obtain the same utility from the decided schedule.

For the sake of clarity, Figure 3 retakes the example presented in and shows how the placement decision is made step by step. Starting at the desired completion time, which is represented by the deadline of the last job, we assign as many tasks as possible from the jobs that are supposed to be running within that timeframe, compressed between that deadline and the previous one. In this case only J3 is running and we are able to assign most of its tasks, as shown in Figure 3(a). Next we estimate the timeframe between time 7 and 15 as shown in Figure 3(b), in which we would like to run all the tasks from J2 and the remaining ones from J3. The scheduler is able to run the remaining tasks from J3, but since there aren't enough resources to run all the tasks from J2, the remaining ones are carried to the last timeframe. Similarly, in the final step of the estimation as shown in Figure 3(c), the scheduler evaluates the timeframe between 0 and 7, in which it is supposed to execute J1 and the remaining tasks from J2. Once the estimation of expected availability is completed, the scheduler is aware of all the steps needed to reach its desired state from the current state, and therefore proceeds to create the next placement of jobs that will satisfy its final goal.

B. Estimation of the resources to allocate to each job

We consider a scenario in which jobs are dynamically submitted by users. Each submission includes both the job's completion time goal (if one is provided) and its resource consumption profile. This information is provided via the job configuration file. The scheduler maintains a list of active minimum number of map tasks that should be allocated concurrently during the next placement cycle, sj fit, by reversing the expected execution assuming all jobs meet their completion time goal Tj goal, and relying on the observed task length (μj) and the availability of resources over time (Ωtt).

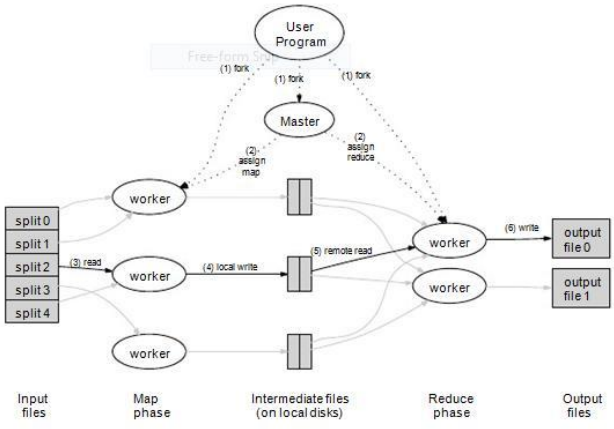

Fig.1 Scheduler Architecture

Figure 5 illustrates the architecture and operation the scheduler. The system consists of five components:

Placement Algorithm, Job Utility Calculator, Task Scheduler, Job Status Updater and Workload Estimator. Each submission includes both the job's completion time goal (if one is provided) and its resource consumption profile.

Most of the logic behind the scheduler resides the utilitydriven Placement Control Loop and the Task Scheduler. The former is responsible for producing placement decisions, while the latter is responsible for enforcing the decisions made by the former. The Placement Control Loop operates in control cycles of period T. Its output is a new placement matrix P that will be active until the next control cycle is reached (current time + T). The Task Scheduler is responsible for enforcing the placement decisions. The Job Utility Calculator calculates a utility value for an input placement matrix which is then used by the Placement Algorithm to choose the best placement choice available. Upon completion of a task, the TaskTracker notifies the Job Status Updater, which for any job j in the system, triggers an update of sjpend and rjpend in the job descriptor. The Job Status Updater also keeps track of the average task length observed for every job in the system, which is later used to estimate the completion time for each job. The Workload Estimator estimates the number of map tasks that should be allocated concurrently (sjreq) to meet the completion time goal of each job, as well as the parameter Sjfit. In this work we concentrate on the estimation of the parameter Sjfit that feeds the Placement Algorithm, as well as the performance model used by the Job Utility Calculator. The major change in this architecture compared to the scheduler presented.

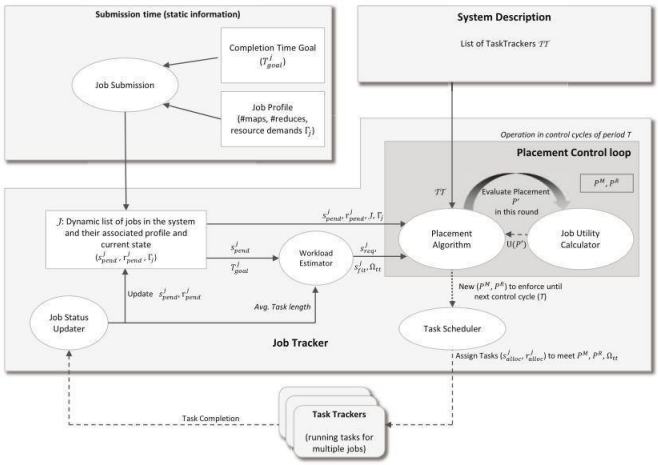

Fig.2 Use case diagram

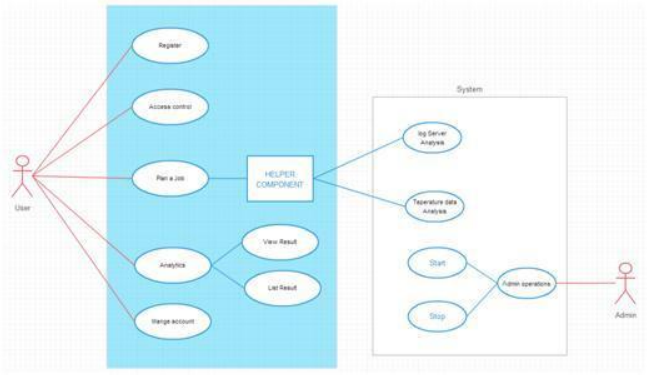

Fig.3 Data Flow Diagram

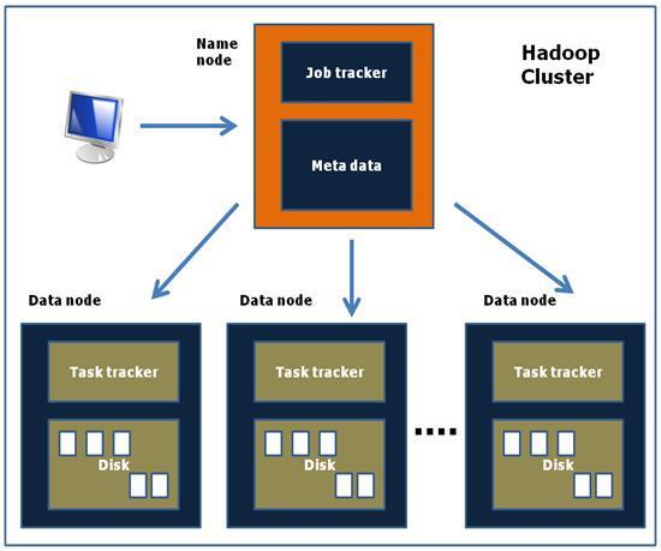

Fig.4 Evaluation

This section includes the description of the experimental environment, including the simulation platform we have built, and the results from the experiments that explore the improvements of our scheduler compared to previous existing schedulers: the default FIFO scheduler, the Adaptive Scheduler described in [7], and the Reverse-Adaptive scheduler proposed in this paper.

In Experiment 1 (Section V-B) we consider the standard scenario in which MapReduce is the only workload running in the system and thus the performance of the scheduler should be similar to previous approaches. In Experiment 2 (Section V-C) we introduce an additional workload in order to gain insight on how does the proposed scheduler perform in this kind of shared environment. And finally, Experiment 3 (Section V-D) shows the impact that the burstiness of transactional workloads may have on the scheduler.

IV CONCLUSION

In this paper we have presented the Reverse-Adaptive Scheduler, which introduces a novel resource management and job scheduling scheme for MapReduce when executed in shared environments along with other kinds of workloads. Our scheduler is capable of improving resource utilization and job performance. The model we introduce allows for the formulation of a placement problem which is solved by means of a utility-driven algorithm. This algorithm in turn provides our scheduler with the adaptability needed to respond to changing conditions in resource demand and availability of resources.

The scheduler works by estimating the need of resources that should be allocated to each job, but in a more proactive way than previously existing work, since the estimation takes into account the expected availability of resources. In particular, the proposed algorithm consists of two major steps: reversing the execution of the workload and generating the current placement of tasks. Reversing the execution of the workload involves creating an estimated placement of the full workload over time, assigning tasks in the opposite direction: starting at the desired end state and finishing at the current state. The reversed placement is used as an estimation to know how many tasks are left at the current state, which allows the scheduler to determine what's the need of tasks for each job and how should they share the available resources. The presented scheduler relies on existing profiling information based on previous executions of jobs to make scheduling and placement decisions.

The goal of the scheduler is to determine the best possible placement of tasks across the tasktrackers so as to maximize resource utilization in the cluster while observing the completion time goal for each job. To achieve this objective, the system dynamically manages the number of slots each tasktracker will provision for each job, and controls the execution of their tasks. Our experiments in a simulated environment driven by representative MapReduce workloads demonstrate the effectiveness of our proposal. To the best of our knowledge this is the first scheduling framework to take into account other non-MapReduce workloads, such as transactional workloads, in addition to leveraging resource information to improve the utilization of resources in the system and meet completion time goals on behalf of users.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation. San Francisco, CA: USENIX Association, December 2004, pp. 137–150.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," ACM Trans. Comput. Syst., vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in Proceedings of twentyfirst ACM SIGOPS symposium on Operating systems principles, ser. SOSP '07. NY, USA: ACM, 2007, pp. 205–220.