Middle-East Journal of Scientific Research
24 (Recent Innovations in Engineering, Technology, Management & Applications): 33-40, 2016
ISSN 1990-9233; © IDOSI Publications, 2016
DOI: 10.5829/idosi.mejsr.2016.24.RIETMA107

Optimization of Dynamic Slot Allocation Scheme In Hadoop MRV1

A. Vanathi and G. Praveena

Department of Computer Science and Engineering, Priyadarshini Engineering College, Vaniyambadi

Abstract: Map reduce is a large scale computing paradigm in cloud computing system for data processing. Poor performance is result in slot-oriented Map Reduce system. To overcome this, we introduce three resource allocation schemes. First, because of pre-configured distinct map slots and reduce slots which are not used, slots can e fully utilized. We proposed alternate technique called Dynamic slot allocation Scheme (DSAS), it allows slot to reallocate or reduce slots based on its need. Second, we propose Execution Performance balancing Scheme to balance the performance criteria between single job and group of jobs. Finally we proposed a Slot Prescheduling technique that can data locality with cost fairness. Thus we produce a system called DyanamicMR to improve the performance Map Reduce data set significantly. The experimental results shows the DynamicMR improve the performance of Hadoop mr1sustantially and also maintains the fairness.

Key words: Cloud Computing · Resource allocation · MapReduce · Slot Allocation

INTRODUCTION

In recent years, MapReduce has become the parallel computing paradigm of choice for large-scale data processing in clusters and data centers. A MapReduce job consists of a set of map and reduce tasks, where reduce tasks are performed after the map tasks. Hadoop [1], an open source implementation of MapReduce, has been deployed in large clusters containing thousands of machines by companies such as Yahoo! And Facebook to support batch processing for large jobs submitted from multiple users (i.e., MapReduce workloads). In a Hadoop cluster, the compute resources are abstracted into map (or reduce) slots, which are basic compute units and statically configured by administrator in advance. Due to 1) the slot allocation constraint assumption that map slots can only be allocated to map tasks and reduce slots can only be allocated to reduce tasks and 2) the general execution constraints that map tasks are executed before reduce tasks, we have two observations:

- There are significantly different performance and system utilization for a MapReduce workload under different job execution orders and map/reduce slots configurations and
- Even under the optimal job submission order as well as the optimal map/reduce slots configuration, there can be many idle reduce (or map) slots while map (or reduce) slots are not enough during the computation, which adversely affects the system utilization and performance.

In our work, we address the problem of how to improve the utilization and performance of MapReduce cluster without any prior knowledge or information (e.g., the arriving time of MapReduce jobs, the execution time for map or reduce tasks) about MapReduce jobs. Our solution is novel and straightforward: we break the former first assumption of slot allocation constraint to allow (1). Slots are generic and can be used by map and reduce tasks. (2). Map tasks will prefer to use map slots and likewise reduce tasks prefer to use reduce slots.

In other words, when there are insufficient map slots, the map tasks will use up all the map slots and then borrow unused reduce slots. Similarly, reduce tasks can use unallocated map slots if the number of reduce tasks is greater than the number of reduce slots. In this paper, we will focus specifically on Hadoop Fair Scheduler (HFS). This is because the cluster utilization and performance for the whole MapReduce jobs under HFS are much poorer

Corresponding Author: A. Vanathi, Department of Computer Science and Engineering, Priyadarshini Engineering College, Vaniyambadi

(or more serious) than that under FIFO scheduler. But it is worth mentioning that our solution can be used for FIFO scheduler as well. HFS is a two-level hierarchy, with task slots allocation across ”pools” at the top level and slots allocation among multiple jobs within the pool at the second level [2]. We propose two types of Dynamic Hadoop Fair Scheduler (DHFS), with the consideration of different levels of fairness (i.e., poolleveland cluster-level). They are as follows:

Pool-Independent DHFS (PI-DHFS): The dynamic slots allocated from the cluster-level, instead of pool-level. It is a typed phase-based dynamic scheduler, i.e. the tasks for map have priority in the use of map slots and tasks reduce have priority to reduce slots (i.e., intra-phase dynamic slots allocation). Only when the respective phase slots requirements are met can excess slots be used by the other phase (i.e., interphase to the dynamic slots allocation).

Pool-Dependent DHFS (PD-DHFS): The assuming that each pool is selfish, i.e., each pool will always satisfy its own map and reduce tasks with its shared map and reduce slots between its map-phased pool and reduce-phased pool (i.e., intra-pool dynamic slots allocation) Then, sharing the unused slots with alternative overloaded pools (i.e., inter pool dynamic slots allocation).

It has been to designed and implemented the two DHFSs on top of default HFS. We evaluate the performance and fairness of our proposed algorithms by the synthetic workloads. Both schedulers, PI-DHFS and PD-DHFS, have used for it.

MapReduce: Initially map reduce proposed by Google [3]. It is a popular programming model for processing large data sets. Now it has been a standard for large scale data processing on the cloud computing. Hadoop [1] an open-source java implementation of MapReduce. When a client submits jobs to the Hadoop cluster, its system breaks each job into multiple map tasks and reduces tasks. Each map task processes (i.e. scans and records) a data block and outcomes is intermediate results like the key-value pairs. The number of map tasks for a job is determined by input data. In hadoop there is one map task per data block. In mapreduce execution time for a map task is determined by an input block of data size. The reduce tasks consists of shuffle/sort/reduce phases. The shuffle phase, the reduce tasks fetch the intermediate outputs from each map task. In the sort/reduce phase, the reduce tasks sort intermediate data and then aggregate the intermediate values for each key to produce the final output. The number of reduce tasks for a job is not determined, which always depends on the intermediate map as outputs.

The number of reduce tasks for a job to be 0:95× or 1:75×reduce tasks capacity [4] hadoop, there are many job schedulers, i.e., FIFO, Hadoop Fair Scheduler [2] Capacity Scheduler [4] he job scheduling in Hadoop is performed by the jobTracker, which manages a set of taskTrackers. In taskTracker has a limited number of map slots and reduce slots, configured by the user by advance. They can perform in one slot per CPU core in order to make CPU and memory management on slave nodes. In task Trackers report periodically to the jobTracker the number of free slots and the progress of the running tasks. In jobTracker allocates the free slots to the tasks to free jobspace. The map slots can only be allocated to map tasks and reduce slots can only be allocated to reduce tasks in mapreduce function.

Hadoop Fair Scheduler [5] a multi-user mapreduce job scheduler that allow group of organization to share a large cluster to multiple users. It ensures that all jobs get roughly an equal share the slot resources phase. It organizes jobs into pools and shares resources fairly across all pools. Each user have allocated a separate pool and, gets an equal share of the cluster how many jobs they submitted. The pool consists of two parts: map-phase pool and reduce-phase pool. Within each map/reduce-phase pool, is used to share map/reduce slots between the running jobs at each phase. Pools can also be given weights to share the cluster.

Related Work: In research work based on large that focuses on the performance development for MapReduce jobs. Broadly, it can be classified into the following two categories,

Data Access and Sharing Optimization: Jiang *et al*. [6] oppose a set of ideas of low-level optimizations include improving I/O speed, utilizing indexes, using fingerprinting for faster key comparisons and block size tuning. They were focused on fine-grain tuning with muliple parameters to reach performance development.

A grawal *et al*. [6] proposed a method to maximize scan sharing by grouping MapReduce jobs into batches so that sequential scans of large files are shared among as

many simultaneous jobs as possible. MRShare [7] is a sharing framework provides three possible work-sharing opportunities, include scan sharing, mapped outputs sharing and Map function sharing across multiple MapReduce jobs, thereby reduce total processing time and to avoid performing redundant work. MapReduce Online is a modified MapReduce system to support online aggregation for MapReduce jobs that can perform by pipelining data within a job and between jobs. LEEN addresses the integrity and data localities. Our approach can be incorporated into these modified MapReduce frameworks (e.g., MRShare, MapReduce Online [8] for further performance improvement. In contrast, our work belongs to the computation and scheduling optimization. Inscence improving performance for MapReduce workloads by maximum the cluster computation utilization.

Computation and Scheduling Optimization: In some computation optimization and job scheduling work are related to our work [9, 10, 11, 12, 13]. Assume job ordering optimization for MapReduce workloads. They model the MapReduce as a two-stage hybrid flow shop with multiprocessor tasks [14], In job submission orders will be different result in varied cluster utilization and system performance. There is an assumption that the execution time for map and reduce tasks for each job, which may not be available in many real-world applications.

In order for only suitable for independent jobs, but fails to consider those jobs with dependency, e.g., MapReduce workflow. In DHFS is not constraint by such assumption and can be used for any types of MapReduce workloads (i.e., independent and dependent jobs). Hadoop configuration optimization is another approach, including [15, 16]. For example, Starfish [11] is a self tuning framework based on the Hadoop's configuration automatically for a MapReduce job such that the utilization of Hadoop cluster can be maximize, based on the cost based model and sampling technique.

In an optimal Hadoop configuration, e.g., Hadoop map and Hadoop reduce slots, there is still room for performance development of a MapReduce job by maximizing the utilization of map and reduce slots. Guo *et al*. [17] method to enable running tasks to steal resources reserved propose a resource stealing for idle slots and give them back proportionally whenever new tasks were assigned, by adopting multithreading technique for running tasks on multiple CPU cores. where it cannot work for the utilization development of those pure idle slave nodes without any running tasks. Polo *et al*. [18] a resource based scheduling technique for MapReduce multi-job workloads for improving resource utilization with the help of extending the abstraction to traditional 'task slot' of Hadoop to 'job slot', with an execution slot that is bound it to a particular job band task type (map or reduce) within that job. In contrast, our proposed schedulers has traditional task slot model and maximize the system utilization by dynamically allocating unused map (or reduce) slots to overloaded reduce (or map) tasks.

Proposed System: Hadoop MRv1 uses the slot-based resource model with the static configuration of map/reduce slots. There is a strict utility constrain that map tasks can only run on map slots and reduce tasks can only use reduce slots. Due to the rigid execution order between map and reduce tasks in a MapReduce environment, slots can be severely under-utilized, which significantly degrades the performance.

In contrast to YARN that gives up the slot-based resource model and propose a container-based model to maximize the resource utilization via unawareness of the types of map/reduce tasks, we keep the slot-based model and propose a dynamic slot utilization optimization system called DynamicMR to improve the performance of Hadoop by maximizing the slots utilization as well as slot utilization efficiency while guaranteeing the fairness across pools.

It consists of three types of scheduling components, namely, Dynamic Hadoop Fair Scheduler (DHFS), Dynamic Speculative Task Scheduler (DSTS) and Data Locality Maximization Scheduler (DLMS).

Our tests show that DynamicMR [19] outperforms YARN for MapReduce workloads with multiple jobs, especially when the number of jobs is large. The explanation is that, given a certain number of resources, it is obvious that the performance for the case with a ratio control of concurrently running map and reduce tasks is better than without control. Because without control, it easily occurs that there are too many reduce tasks running, causing the network to be a bottleneck seriously.

For YARN, both map and reduce tasks can run on any idle container. There is no control mechanism for the ratio of resource allocation between map and reduce tasks. It means that when there are pending reduce tasks, the idle container will be most likely possessed by them. In contrast, DynamicMR follows the traditional slot-based model.

In contrast to the 'hard' constrain of slot allocation that map slots have to be allocated to map tasks and reduce tasks should be dispatched to reduce tasks, DynamicMR obeys a 'soft' constrain of slot allocation to allow that map slot can be allocated to reduce task and vice versa. But whenever there are pending map tasks, the map slot should be given to map tasks first and the rule is similar for reduce tasks.

It means that, the traditional way of static map/reduce slot configuration for the ratio control of running map/reduce tasks still works for DynamicMR. In comparison to YARN which maximizes the resource utilization only, DynamicMR can maximize the slot resource utilization and meanwhile dynamically control the ratio of running map/reduce tasks via map/reduce slot configuration.

The proposed system, to address the mentioned problems, this paper presents DynamicMR, a dynamic slot allocation framework to develop the performance of a MapReduce cluster via optimizing the slot utilization. Specifically, DynamicMR focuses on Hadoop Fair Scheduler (HFS). Because the cluster utilization and performance for MapReduce jobs under HFS are much poorer than that under FIFO scheduler.

DynamicMR consists of three optimization techniques,

- Dynamic Hadoop Slot Allocation (DHSA)
- Speculative Execution Performance Balancing (SEPB)
- Slot PreScheduling.

The performance and slot utilization of a Hadoop cluster can be optimized with the following step by step processes.

- If a slot is idle, then DynamicMR will first attempt to improve the slot utilization with DHSA technique. It will evaluate based on numerous constraints like fairness, load balance and decide whether to allocate the idle slot to the task or not.
- If the allocation is true, DynamicMR will further optimize the performance by improving the efficiency of slot utilization with SEPB. It works on top of Hadoop speculative scheduler to check whether to allocate the accessible idle slots to the pending tasks or to the speculative tasks.
- When to allocate the idle slots for pending/ speculative map tasks, DynamicMR will be able to additional improve the slot utilization efficiency from the data locality optimization aspect with Slot Pre Scheduling.

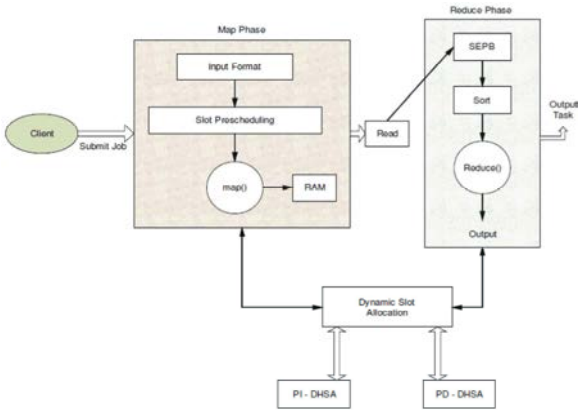

Fig. 1: DynamicMR Framework

Dynamic Hadoop Slot Allocation (DHSA): In the current configuration of MapReduce as an under-usage of the slots with the quantity of map and reduce tasks with the shifts over the long run. Our dynamic slot allocation approach is taking into account the approach that at the peculiar time there will be idle map slots(or reduce), as the jobs continues from map stage to reduce stage. We may utilize the unused map slots. Those overburden reduce tasks to enhance the execution of the MapReduce workload and the other way around.

For further make utilization of idle reduce slots for running map tasks are used. We break the certain phase for an current MapReduce structure that the map tasks can just run on map slots and reduced tasks can just run on reduce slots. There are two challenges specified below that must be considered:

(C1): Fairness is an imperative metric in Hadoop Fair Scheduler (HFS). We proclaim it as reasonable when all pools have been designated with the same amount of resource. In HFS, task slots are first allocated over the pools [9] and later then the slots are distributed to the jobs inside the pool. Also, a MapReduce job computation embodies two sections: map-phase task computation and reduce-phase task computation.

(C2): The resource requirement between the map slots and reduced slots are especially diverse. The purpose for this is the map tasks and reduced tasks regularly show totally different execution designs. Reduce task has a tendency to expend considerably more resources, for example, memory and system network speed. Basically permitting reduce tasks to utilize map slots configuring every map slots to take more resources, which will therefore lessen the powerful number of slots on every node, creating resources under-used amid runtime.

With a due appreciation towards (C1), we set forth a Dynamic Hadoop Slot Allocation (DHSA). It contains two choices, to be specific, pool- free DHSA(PI-DHSA)pool-Independent DHSA (PI-DHSA) HFS utilizes max-min fairness [20] to allocate slots crosswise over pools with least ensures at the map-phase and reduce-phase, individually.

Pool-Independent DHSA (PI-DHSA): Pool-Independent DHSA (PI-DHSA) extends the HFS by provision slots from the clusters of worldwide level and free of pools. The allocation procedure is comprised of two sections:

- Intra-phase dynamic slot allocation: Each pool is piece into two sub-pools, i.e., map phase pool and reduce phase pool. At every stage, every pool will search out its share of slots.
- Inter-phase dynamic slot allocation: Behind the intra-phase dynamic slot allocation for both the map phase and reduced phase, next we can perform the dynamic slot allocation crosswise more typed phase.

The intact dynamic slot allocation flow is that, at whatever point a pulse is gotten from a computing node, at first we process the amassed demand for map slots and reduce slots for the present MapReduce workload. At that point we focal point alertly the need to acquire map (or reduce) slots for reduce (or map) tasks in light of the interest for map and reduce slots, with revere to these four situations. The specific number of map (or reduce) slots to be obtained is based on the account of capacity of unused reduced (or map) slots and its map (or reduce) slots needed. To achieve the reluctance usefulness, we give two variables rate Of Borrowed Map Slots and rate Of- Borrowed Reduce Slots, defined as the rate of vacant map and reduced slots that can be obtained, separately.

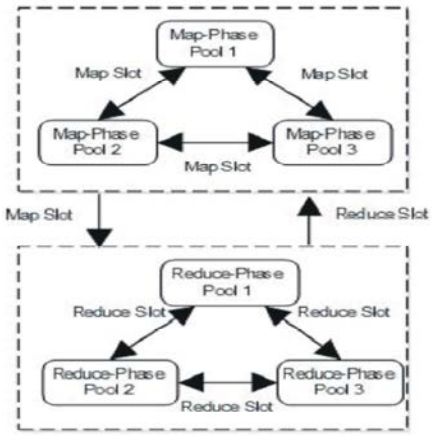

Fig. 2: Fairness-based slot allocation flow for PIDHSA

Thus, we can restrict the quantity of unused map and reduced slots that ought to be distributed for map and reduced tasks at every pulse of that task tracker. With these two parameters, clients can flexibly adjust the exchange off between the performance execution optimization and the starvation minimization.

In addition, Challenge (C2) makes us to review that we can't treat map and reduce slots as same and just obtain unused slots for map and reduce tasks. Rather, we should be mindful of shifted resource sizes of map and reduce slots. A slot weight- based methodology is therefore proposed to address the issue. We allot the map and reduce slots with distinctive weight values, regarding the asset configurations. Particular to the weights, we can alterably decide the amount of map and reduce a task which has to be generate in the length of runtime.

Algorithm 1: the Dynamic Task Assignment Policy for Task Tracker under PI-DHSA:

When a heartbeat is received from a compute node *n*:

1: compute its clusterUsedMapSlots, clusterUsedReduceSlots, mapSlotsDemand, reduceSlotsDemand, mapSlotsLoadFactor and reduceSlotsLoad-Factor.

2: */*Case 1: both map slots and reduce slots are sufficient.*/*

3: if (*mapSlotsLoadFactor*?? 1 and *reduceSlotsLoadFactor*?? 1) then

4: //No borrow operation is needed.

5: end if

6: */*Case 2: both map slots and reduce slots are not enough.*/*

7: if (*mapSlotsLoadFactor*_ 1 and *reduceSlotsLoadFactor*_ 1) then

8: //No borrow operation is needed.

9: end if

10: */*Case 3: map slots are enough, while reduce slots are insufficient. It calculates borrowed map slots for reduce tasks.*/*

11: if (*mapSlotsLoadFactor*_ 1 and *reduceSlotsLoadFactor*_ 1) then

12: *currentBorrowedMapSlots= clusterUsedMapSlots-clusterRunningMapTasks*;

13: *extraReduceSlotsDemand*= min*{*max*{* floor*{ clusterMapCapacity* percentageOfBorrowedMapSlots}* - currentBorrowedMapSlots, 0*}*, *reduceSlotsDemand-clusterReduceCapacity}*

14: *updatedMapSlotsLoadFactor*= (*mapSlotsDemand*+ *extraReduceSlotsDemand*) / *clusterMapCapacity*;

15: end if

16: */*Case 4: map slots are insufficient, while reduce slots are enough. It calculates borrowed reduce slots for map tasks.*/*
17: if (*mapSlotsLoadFactor*_ 1 and *reduceSlotsLoadFactor*_ 1) then
18: *currentBorrowedReduceSlots= clusterUsedReduceSlots- clusterRunningReduceTasks*;
19: *extraMapSlotsDemand*= min{max{ floor{ *clusterReduceCapacity * percentageOfBorrowedReduceSlots}* - currentBorrowedReduceSlots, 0*}mapSlotsDemand-clusterMapCapacity}*
20: *updatedReduceSlotsLoadFactor*= (*reduceSlotsDemand*+ *extraMap- SlotsDemand*) / *clusterReduceCapacity*;
21: end if
22: compute availableMapSlots and availableReduceSlots based on the updated map/reduce load factor and used slots.

Pool-Dependent DHSA (PD-DHSA):

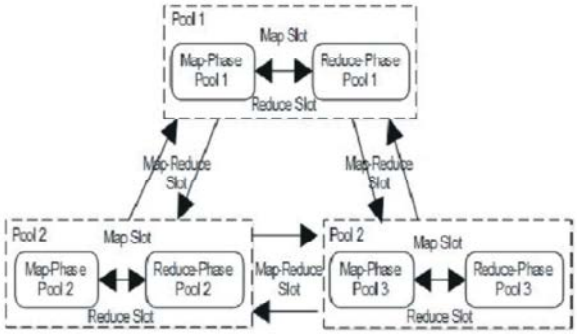

Fig. 3: Pool-Dependent DHSA.

As an opposite point on checking towards PI-DHSA Pool-Dependent DHSA (PD-DHSA) considers fairness for the dynamic slot allocation across pools. Accepting that every pool, includes two sections: Map phase pool and Dynamic Phase pool, is selfish. It is considered fair when aggregate quantities of map and reduce slots allocated across pools are the same with one another. PD-DHSA will be performed with the accompanying two courses of actions:

Intra-Pool Dynamic Slot Allocation: In an early process, each typed- phase pool will receive its share of typed-slots in an max-min fairness at all phase. It has four possible relationships cases for every pool regarding its demand (denoted as mapSlots Demand, reduce Slots Demand) and its workload (marked as mapShare, reduceShare) between two phases:

Case (a): mapSlotsDemand<reduceShare and reduceSlots-Demand >reduceShare. We can use of the unused map slots for its overloaded reduce tasks from its reduce-phase pool first before using other pools.

Case (b): mapSlotsDemand>mapShare and reduceSlots-Demand <reduceShare. we can use some unused reduce slots for its map tasks from its map-phase pool first before using pools.

Case (c): mapSlotsDemand<mapShare and reduceSlots-Demand <reduceShare. Both map slots and reduce slots are enough for its use. It can give some unused map slots and reduce slots to other pools.

Case (d): mapSlotsDemand>mapShare and reduceSlots-Demand >reduceShare. If both map slots and reduce slots of a pool have become insufficient. It may have to borrow some unused map or reduce slots from other pools through inter-Pool dynamic slot allocation is shown below.

Inter-Pool Dynamic Slot Allocation:

It is obvious that,

- If a pool, has mapSlotsDemand + reduce SlotsDemand<mapShare + reduceShare. The slots are enough for the pool and there is no need to get some map or reduce slots from other pools
- On the contrary, when mapSlotsDemand + reduceSlotsDemandmapShare + reduceShare, the slots are not enough even after Intra-pool dynamic slot allocation.

The overall slot allocation process for PD-DHSA is as sketched down below in figure. At first, it computes the maximum number of free slots that can be allocated at each round of heartbeat for the task tracker. Next it starts the slot allocation for pools. For every pool, there are four possible slot allocations as illustrated in Figure below.

Case (1): We try the map tasks allocation, if there are idle map slots for the task tracker and there are pending map tasks for the pool.

Case (2): If the attempt of Case(1) fails, the condition does not hold good and it level, we continue to try reduce tasks allocation when there are pending reduce tasks and idle reduce slots.

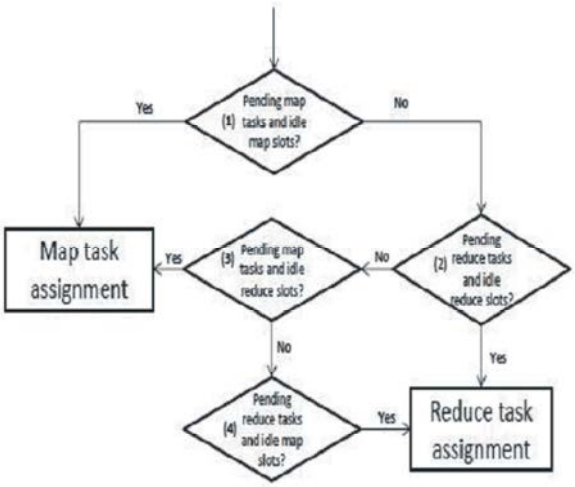

Fig. 4: The slot allocation flow for each pool under PD-DHSA.The numbers labeled in the graph corresponds to Case (1)-(4).

Case (3): If Case(2) fails due to the required conditions does not hold, we try for map task allocation again. If Case(1) fails then there might not have to be any idle map slots available. In contrast, if Case(2) fails then there are no pending reduce tasks. In this case, we can relay on reduce slots for map tasks of the pool.

Case (4): If Case(3) fails, we try for reduce task allocation once again. Case(1) and Case(3) fail might be because of no valid locality-level pending and map tasks available, but there are idle map slots. In contrast, Case(2) maight not have any idle reduce slots available. At such cases, we can allocate map slots for reduce tasks for the pool.

Algorithm 2: the Dynamic Task Assignment Policy for Tasktracker under PD-DHSA:

When a heartbeat is received from tasktracker*tts*:
1: Compute its totalSlotsDemand, totalSlotsCapacity, trackerSlotsCapacity, trackerRunningTasksNum and trackerCurrentSlotsCapacity.
2: /* Return when there are no idle slots. */
3: if trackerRunningTasksNum ?? trackerCurrentSlotsCapacity then
4: return NULL;
5: end if
6: for (i = 0; i _ trackerCurrentSlotsCapacity - trackerRunningTasksNum; i++) do
7: Sort pools by distance below min and fair share
8: for (Pool p : pools) do
9: */* Case (1): allocate map slots for map tasks from Pool p*/*
10: if (there are pending map tasks and idle map slots) then
11: attempt to allocate map slots for map tasks (considering data locality) and jump out of loop if allocation succeeded.
12: end if
13: */* Case (2): allocate reduce slots for reduce tasks from Pool p*/*
14: if (Case (1) failed and there are pending reduce tasks and idle reduce slots) then
15: attempt to allocate reduce slots for reduce tasks and jump out of loop if allocation succeeded.
16: end if
17: */* Case (3): allocate reduce slots for map tasks from Pool p*/*
18: if (Case (2) failed and there are pending map tasks) then
19: attempt to allocate reduce slots for map tasks (considering data locality) and jump out of loop if allocation succeeded.
20: end if
21: */* Case (4): allocate map slots for reduce tasks from Pool p*/*
22: if (Case (3) failed and there are pending reduce tasks) then
23: attempt to allocate map slots for reduce tasks and jump out of loop if allocation succeeded.
24: end if
25: end for
26: */* Case (5): schedule the non-local map tasks when its node-local tasks cannot be satisfied. */*
27: if (Case (1)-(4) failed) then
28: for (Pool p : pools) do
29: if (there are pending map tasks) then
30: attempt to allocate map/reduce slots to map tasks (not considering data locality) and jump out of loop if allocation succeeded.
31: end if
32: end for
33: end if

Furthermore, there is a special scenario that needs to be considered particularly. so, it is possible that all the above four possible slot allocation attempts fail for all pools, due to the data locality for map tasks.

CONCLUSION

This paper proposes a DynamicMR Technique can be use to improve the execution of MapReduce workloads

while charge up the fairness. It comprises of three methods, in exacting DHSA, SEPB and Slot PreScheduling, all of which consider on the slot use optimization for MapReduce assembly from alternate points of view. DHSA concentrates on the slot use extension by distributing map or reduce slots to map and reduce tasks alterably.

Especially, it doesn't have any presupposition or involve any earlier learning and can be utilized for any sorts of MapReduce jobs (e.g., autonomous or subordinate ones). Two sorts of DHSA are introduced, in particular, PI-DHSA and PD-DHSA, in view of distinctive levels of fairness. Client can choose both of them likewise. Rather than DHSA, SEPB and Slot PreScheduling with the effectiveness advancement for a specified slot usage. SEPB recognizes the slot unused issue of hypothetical execution.

It can adjust the execution trade off between a single job and a batch of job alterably. Slot PreScheduling enhances the proficiency of slot use by growing its data locality. By empowering the over three systems to work helpfully, the examining results demonstrate that our proposed DynamicMR can develop the execution of the Hadoop framework altogether.

In future, we plan to believe executing DynamicMR for distributed computing surroundings with more measurements (e.g., plan, due date) considered and distinctive stages.

REFERENCES

1. Hadoop. http://hadoop.apache.org.
2. Zaharia, M., D. Borthakur, J. Sarma, K. Elmeleegy, S. Schenker and I. Stoica, 2009. Job Scheduling for Multi-user Mapreduce Clusters. Technical Report EECS-2009-55, UC Berkeley Technical Report.
3. Dean, J. and S. Ghemawat, 2004. MapReduce: Simplified Data Processing on Large Clusters, In Proceedings of the 6th Symposiumon Operating Systems Design and Implementation (OSDI).
4. How Many Maps And Reduces. http://wiki.apache.org/hadoop/HowManyMapsAndReduces.
5. Max-Min Fairness (Wikipedia). http://en.wikipedia.org/wiki/Maxmin fairness.
6. Agrawal, P., D. Kifer and C. Olston, 2008. Scheduling Shared Scans of Large Data Files. In VLDB.
7. Nykiel, T., M. Potamias, C. Mishra, G. Kollios and N. Koudas, MRShare: Sharing Across Multiple Queries in MapReduce . Proc. of the 36th VLDB (PVLDB), Singapore, September 2010.
8. Condie, T., N. Conway, P. Alvaro, J.M. Hellerstein, 2010. *MapReduce online*. In Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation, pp: 21C21.
9. Moseley, B., A. Dasgupta, R. Kumar, T. Sarl, 2011. On scheduling in map-reduce and flow-shops. SPAA, pp: 289-298.
10. Verma, A., L. Cherkasova and R.H. Campbell, 2013. Orchestrating an Ensemble of MapReduce Jobs for Minimizing Their Makespan, IEEE Transaction Ondependency and Secure Computing.
11. Verma, A., L. Cherkasova, R. Campbell, 2012. Two Sides of a Coin: Optimizing the Schedule of MapReduce Jobs to Minimize Their Makespan and Improve Cluster Performance. MASCOTS.
12. Tang, S.J., B.S. Lee and B.S. He, 2013. MROrder: Flexible Job Ordering Optimization for Online MapReduce Workloads. in Euro-Par, pp: 291-304.
13. Tang, S.J., B.S. Lee, R. Fan and B.S. He, 2013. Performance Optimization for MapReduce Workloads, CORR (Technical Report).
14. O¢guz, C. and M.F. Ercan, 1997. Scheduling multiprocessor tasks in a two-stage flow-shop environment. Proceedings of the 21st international conferenceon Computers and Industrial Engineering, pp: 269-272.
15. Herodotou, H., H. Lim, G. Luo, N. Borisov, L. Dong, F.B. Cetin and S. Babu, 2011. Starfish: A Self-tuning System for Big Data Analytics. In CIDR, Pages 261C272.
16. Herodotou, H. and S. Babu, 2011. Profiling, What-if Analysis and Costbased Optimization of MapReduce Programs. in Proc. of the VLDB Endowment, 4(11).
17. Guo, Z.H., G. Fox, M. Zhou and Y. Ruan, 2012. Improving Resource Utilization in MapReduce. 2012 IEEE International Conference on Cluster Computing (CLUSTER), pp: 402-410.
18. Polo, J., C. Castillo and D. Carrera, *et al*., 2011. Resource-aware Adaptive Scheduling for MapReduce Clusters. Proceeding Middleware'11 Proceedings ofthe 12th ACM/IFIP/USENIX international conference on Middleware, pp: 187-207.
19. Shanjiang Tang, Bu-Sung Lee, Bingsheng He, 2014." DynamicMR: A Dynamic Slot Allocation Optimization" IEEE Transactions On Cloud Computing, 2(3).
20. Jiang, D.W., B.C. Ooi, L. Shi and S. Wu, 2010. The Performance of MapReduce:An Indepth Study, PVLDB, 3: 472-483.