INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X
Impact Factor: 6.078
(Volume 9, Issue 2 - V9I2-1439)
Available online at: https://www.ijariit.com

Advanced Autoscaling and Team Tenancy for Hadoop Job Clusters

Rajkamal Mahamuni Natarajan
rajkamalmn@gmail.com
Indeed Inc, Austin, TX

Raj Pravinbhai Manvar
rajmanvarchamp@gmail.com
Indeed Inc, Austin, TX

Thai Bui
blquythai@gmail.com
Stealth Startup Austin, TX

ABSTRACT

Hadoop has a powerful framework to process large distributed data across distributed nodes. Hadoop Data Lake empowers users of different streams like engineers, data scientists, analysts in an organization. Slicing and dicing of data, complex computations can be expressed and built with Query Language and doesn't need the expertise or knowledge of Map-Reduce or Spark jobs. Once the complex logic is expressed as a query and the query is scheduled to run in regular cadence, the cluster can be auto scaled based on applications running at a given time rather than running a static big cluster. In a multi tenant cluster, where autoscaling is performed, resources should be isolated and dedicated to achieve multi tenancy. Dedicated resources will drive attributing cost to a specific team/tag. Dedicated resources also solve the noisy neighbor problem. This paper details architecture, algorithm and framework to allow multi-tenant job clusters to achieve team tenancy and auto-scale the cluster seamlessly.

Keywords *Data Engineering, Autoscaling, Team Tenancy, Cost Attribution, Big Data.*

I. INTRODUCTION

Hadoop brings the power of distributed systems to crunch, slice and dice and process voluminous data with ease. Constructing a Data Lake powered by Hadoop cluster brings the power of Query Language to distributed systems and empowers any user with querying language skill to express logic in query language to process Data. Hive is used as backbone to architect a data warehouse in Hadoop Cluster and process different datasets in an organization. The Hadoop cluster can be brought up with bare minimum software HDFS – to support Hadoop file system, YARN – to facilitate resource management, Oozie – to support any scheduled jobs in the cluster, Map Reduce – parallel process data, Zookeeper – coordination of Hadoop services and Hive – Data warehouse software.

Multiple teams producing multiple datasets can be written to a cloud based storage(Amazon S3, Google Cloud Storage). The datasets can either be incremental(every new dataset has old dataset + some changes) or non-incremental(every dataset has unique records). Also the datasets may be produced in different cadence – hourly, daily, weekly, monthly.

Each datasets would be represented in Hive as a table. For incremental datasets, a table suffixed with frequency parameters(say year, month and day) can be suffixed to the table and a view can be maintained which always points to the most recent table. For non-incremental datasets, a partitioned table can be created in hive with each partition for the frequency(say year, month, day) referring to the exact location of the dataset in cloud storage. With all datasets living in the cloud, the tables will be existing in Hive. When a user queries Hive for a specific dataset, then Hive would actually fetch the data.

Cost of processing the data would involve fetching the data from Cloud Storage and performing computation on the data using machines. Regardless of the data remaining local or remote the cost spent for compute will almost remain the same. To optimize the fetch Hot, Warm and Cold Storage can be performed.

In such an environment, different streams of users can schedule queries to generate another dataset or generate user friendly consumable files(csv, xlsx) or write data to another datastore. Each scheduled query will be run on the cluster as a Hadoop Job. In an organization with multiple users, running a single cluster with multiple tenants will bring huge benefits like reducing redundant infrastructure, HDFS node costs. In a multi-tenant Hadoop jobs cluster, instead of running a fat static cluster without scaling, running a lean cluster with advanced autoscaling brings benefits like cost attribution, team tenancy and solving noisy neighbor problem. This paper details advanced autoscaling and supports team tenancy on a multi-tenant cluster.

II. RELATED WORK

Prior approaches involve performing autoscaling on the cluster and ignoring the problem of noisy neighbor or a smart algorithm that can attribute the cost of computation to a specific user. Paper [1] explores dynamic autoscaling with data distribution but ignores the fact that with exploding data and preserving all data in the cluster will have more cost than having the data on the cloud. With more optimal fetching of data using smart partitioning or bucketing the cost of fetching data can be reduced rather than scaling the cluster and redistributing data. Paper [2] explores using ML on autoscaling but the training time can greatly be reduced with autoscaling on need basis in a system which uses scheduled queries.

Existing works focus on different solutions that suits experimental and usage of cutting edge features to autoscaling. However the papers ignore the industry and enterprise use cases. In an organization that uses Big Data based jobs scheduled to run on regular cadence, the jobs need stability, team tenancy, cost effectiveness and cost attribution. The existing papers address the autoscaling problem from a research oriented view and not from an enterprise view.

III. OUR APPROACH

In this paper, we present an advanced autoscaling algorithm and team tenancy for Hadoop clusters, which addressed Enterprise use cases. In an enterprise, a common cluster is shared by multiple teams. Multiple teams schedule queries/jobs in regular cadence for multiple use cases. Teams expect the jobs to complete within stipulated service level agreement and wouldn't wait for immediate results. Most jobs are scheduled to run overnight. A stable job would be one which always runs within a reasonable service level agreement time and always spends the same cost(with a reasonable variance) for every run. The algorithm ensures that resources are allocated for each team once a job has commenced and removes or transfers resources once the job has completed. By ensuring the resources are team tagged and jobs specific for the team are run in dedicated resources, the algorithm ensures a) cost is attributed to the team, b) jobs do not wait for resources and c) noisy neighbor(one application consuming all the resources while other resources are starving for resources) doesn't consume all resources.

IV. SYSTEM MODEL

Considering an organization with petabytes of data and the data growing exponentially every day, not all the petabytes of the data will be used for computation every day. Naturally, Data can be classified as Hot, warm and cold data. Hot data will be accessed frequently, Warm data will be accessed less frequently and cold data will either not be used or will be used rarely. Hence having all data in the Hadoop cluster would just not be costing more but would also not improve any performance. Hence having hot data alone in the Hadoop File System would be sufficient. A better model would be to have hot data in the cluster and all other data in Cloud based file system or all files in a cloud based file system(sacrificing some time to fetch the data). Users of the data warehouse can be broadly classified into two – 1) On Demand Queries, 2) Scheduled queries and 3) Advanced Queries. 1) On Demand queries are the scenarios when users of data warehouse run an arbitrary query to compute a metric or investigate an issue or explore data. 2) Scheduled Queries are the scenarios when users have a query which generates another dataset or generates a report and the queries should be run in regular cadence. A Hadoop job which runs with hive-action as spine would help power the scheduled queries in the Hadoop cluster. 3) Advanced queries are users performing Machine Learning or performing anomaly detection in the datasets. A Zeppelin Notebook or Jupyter Notebook that can connect to Hive or run Spark on Hive would best serve the needs of advanced users. The paper will be focusing on a Hadoop jobs cluster with scheduled queries.

V. PROBLEM STATEMENT

On a Hadoop cluster, with multiple teams scheduling jobs, a) auto scale the cluster to provide resources for all jobs, b) attribute the cost to teams, c) ensure resources allocated for the teams are used by the specific team and d) fat jobs don't cause noisy neighbor problem.

VI. SOLUTION

For a cluster in which queries are scheduled as Hadoop jobs, the users wouldn't be waiting for immediate completion of jobs. In this case, upscaling can be triggered by an application start and downscaling can be triggered by application success or failure. This use case will act as the use case to implement advanced autoscaling algorithm which has Team tenancy and cost attribution features.

Team tenancy denotes isolating each team's jobs to dedicated resources and dedicated queue. Multi-tenant cluster is a cluster shared by multiple teams or individuals. In a multi-tenant cluster each job will have different workloads and characteristics. When different jobs compete for resources, one job might be occupying all the resources and the rest of the jobs will have to wait for the resource. This problem is also described as a noisy-neighbor problem. Using Hadoop YARN's Fair resource sharing without pre-emption, a fat job that started first will occupy all resource and a lean job would be waiting for the fat job to complete. Even though both jobs bring up their fair share of nodes, the fat job would have to complete for the lean job to run. If Preemption is enabled, then the fat job tasks would be preempted for the lean job to run. Also the tasks cannot be isolated to their own dedicated resources. To alleviate, YARN Node labels and queues are used. Each team is assigned a label and YARN queues are created with the label. Resources are isolated by assigning YARN node labels. Each YARN queue will use the resources that are labeled with a specific node label. When a team submits a job, the team's job is restricted to run in the team specific YARN queue. This restriction will ensure that the team can only use the resource dedicated for the team.

Once a team's job is isolated to the team specific resources, costs are directly attributed to the team. Memory Optimized and performance optimized jobs would run faster and consume less cost. Understanding the cost involved for each job would drive more optimization. Hence Cost attribution will promote more optimization of jobs and less consumption of resources.

In this Hadoop job cluster, a master is brought up to run all important service (Hadoop name node, Oozie, Yarn resource manager) and 5 slaves to run Hadoop file system storage and workers to run YARN, Map Reduce, Spark or Hive Clients. These workers perform the actual compute. Once the compute is complete, data will either be flushed to the Hadoop File System or to cloud storage. Going forward the memory available in worker nodes will be referred to as resource.

Autoscaling is a seamless process of adding or removing resources to the cluster. Autoscaling has two components – Upscale and Downscale. Upscale is the process of adding resources to the cluster. Downscale is the process of removing resources from the cluster.

There are two interesting actors that affect autoscaling and suffer from two characteristics. A1) Resources and A2) Jobs. Actor resource can be defined as a compute power or in simple terms total MBs or GBs or TBs available to perform computation. Actor Job can be defined as the compute performed on the data. Since Data doesn't affect autoscaling, the factor is omitted from this section. Characteristics of these actors are C1) Idle Time and C2) Wait Time. Idle Time is the time for which a resource remains idle in the cluster. Wait Time is the time for which a job waits for the availability of a resource.

An efficient algorithm would be the one which provides the best result for all the three actors. Resources should be effectively used. Both Idle and Wait Time should be minimized. Jobs should get resources to perform the compute and jobs shouldn't be competing for resources and hence increase wait time. This competition for resources is solved with Team tenancy described above. For the sake of simplicity, its assumed at this point of time that all jobs need an equal number of resources say r.

A Self throttling autoscaling algorithm is designed to perform resource allocation to jobs. An internal application keeps track of all the state of jobs in a database. And the application runs the algorithm to perform auto-scaling. For any job there are two status maintained. 1) Job Status and 2) Autoscaling Status. For simplicity considering only job status which are either running/succeeded/killed/failed. Any Job which is in running status will be in need of resource. Any job which is in succeeded/killed/failed will be ready to release the resources acquired. Any job which needs a resource would be in NEEDS_UPSCALE(0) autoscaling status. Any job which has been allocated a resource would be in UPSCALED(1) autoscaling status. Any job from which resources have been removed will be in DOWNSCALED(2) autoscaling status. At the end of this algorithm, will be explaining why Job Status and autoscaling status should be considered for autoscaling rather than just using a single autoscaling status.

For any job that has started running, the application will add an entry in job status – running, NEEDS_UPSCALE status. If a job has succeeded, job status will be updated to succeeded. If a job has failed, job status will be updated to failed. If a job has been killed, job status will be updated to killed. Jobs use Oozie workflow notification functionality to notify the change in status and the application will update the entry in database.

At any given point in time, only one action – upscaling or downscaling can be performed in a Hadoop cluster and hence autoscaling is sequential in a cluster. The process of creation of resource and adding the resource to the Hadoop cluster can have a worst case scenario of about 30 minutes. The process of removing the resource from the cluster and deleting the resource would take about 10 minutes.

The algorithm can be documented step by step as follows

1) Compute total nodes needed – Nodes needed for all jobs in job status - running and autoscaling status - NEEDS_UPSCALE status. Add the nodes to cluster and update each job's autoscaling status to UPSCALED.

2) For any job in succeeded/killed/failed status and autoscaling status – UPSCALED and current time - job completion time > 15 minutes, remove nodes from the cluster and update the autoscaling status to DOWNSCALED.

3) As an optimization, Before step 1, transfer surplus nodes from a succeeded/killed/failed job in UPSCALED autoscaling status to jobs in running and NEEDS_UPSCALE autoscaling status. Update UPSCALED -> DOWNSCALED and NEEDS_UPSCALE to UPSCALED.

Omitted the complexity of different jobs requiring different number of nodes for the sake of simplicity.

Since jobs belonging to same team are ran under same YARN queue, consider a job performing a small task(takes about a minute to complete) gets to running job state and NEEDS_UPSCALE autoscaling status. If the autoscaling thread was sleeping or was upscaling the cluster for another job, and if there are enough free resources in the queue(assuming a running job did not consume all compute memory or a job just went to succeeded/failed/killed when autoscaling was busy) then the small job would complete with the resource before even the autoscaling thread can allocate resource. Then the job would move to succeeded state. And hence autoscaling thread will not allocate resource, since the job is in succeeded status and NEEDS_UPSCALE autoscaling status. If only autoscaling status was considered then autoscaling algorithm would have allocated nodes unnecessarily. Hence two status – job status and autoscaling status are considered to allocate resource.

The application would have an autoscaling thread that would wake up every 5 minutes to collect all jobs that are in running status and NEEDS_UPSCALE autoscaling status. Consider this as the first run and hence there are no idle nodes. A case of presence of idle nodes will be covered at a later point in time.

When a job begins, using workflow notification, the application will be notified. The application will create an entry for the job in running job status and NEEDS_UPSCALE autoscaling status. Considering the actors here A1) Resource – There is resource in the cluster and hence the resource is not wasted. A2) Job – Job is in a waiting state in need of resources. Considering the characteristics C1) Idle Time – The resources are not having any idle time at this given point of time. C2) Wait Time – The job is waiting for resources. This wait time worst case can be worst case thread wake up time 5 minutes + worst case upscaling time – 30 minutes.

Autoscaling thread would consolidate the number of nodes needed for each job and would add the nodes to the cluster. Once the nodes are added to the cluster, team specific tags are assigned to the nodes. This ensures each job gets a necessary node. And then the job would be updated to UPSCALED status. During the UPSCALED state, each job would have got the node needed and would be performing the compute. Considering the actors now A1) Resources – Resources are created and are fully used as soon as the compute memory is available in the cluster. A2) Job is in running state. Characteristics - C1) Idle Time – Resources at this moment do not have any idle time. C2) Wait Time – Job does not have a wait time now.

Once the job has completed, the job would be moved to succeeded/killed/failed job status and still the job would be in UPSCALED autoscaling status. At this point of time, considering that the worst case scenario Wait Time for a job to get a node would be about 35 minutes, the resources are not immediately removed from the cluster. Downscaling process if postponed to about 15 minutes of cool down period from the time the job was updated to SUCCEDED/killed/failed job status. Considering the actors - A1) Resource – Resource is in idle state A2)Job – Job is in completed state. Characteristics – C1) Idle Time – Resource can have a worst case idle time of 5 minutes for autoscaling thread wake up + 15 minutes for cool down time. C2) Wait Time – Wait Time is zero.

Autoscaling algorithm introduces resource transfer at this point of time. Resource transfer would be the transfer of resources from a team which has idle nodes(surplus) to a team which is in need of the node(needy). Considering Job States, if a job is in succeeded/failed/killed job status and UPSCALED autoscaling status, and there is another job in running job status and NEEDS_UPSCALE autoscaling status, then the nodes from UPSCALED can be transferred to NEEDS_UPSCALE state. Assume two jobs j1 and j2 where j1 is in succeeded/failed/killed job status and UPSCALED autoscaling status and j2 is in running job status and NEEDS_UPSCALE autoscaling status. Then nodes from j1 are transferred to j2 and j1 will be updated to DOWNSCALED and j2 will be updated to UPSCALED. Consider that j1 and j2 entered into their state within the time frame of the autoscaling thread in sleep mode(5 minutes) + when the resources are in Cool down time(15 minutes).

If one or more jobs are in succeeded/failed/killed job status and NEEDS_DOWNSCALE status and has exceeded the cooldown period and there is no other job in running job status and NEEDS_UPSCALE status, then remove the nodes from the cluster. And update the autoscaling status of jobs to DOWNSCALED. Considering the actors A1) Resource – Resources have been removed and hence will not be affected. A2) Job – Job has already completed and is not affected. Considering the characteristics C1) Idle Time – None of the resources are in cluster and hence idle time is zero. C2) Wait Time – None of the jobs are waiting and hence wait time is zero.

VII. ILLUSTRATIONS

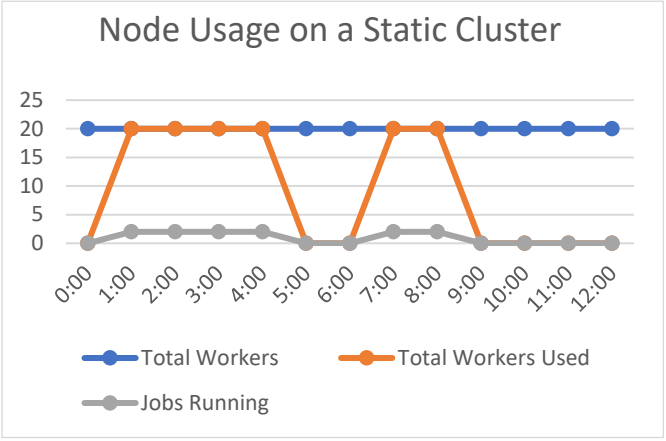

FIGURE 1.Worker node usage illustration for 12 hours on a static cluster. Nodes are always up and running and the nodes are used only for a brief period when jobs are running.

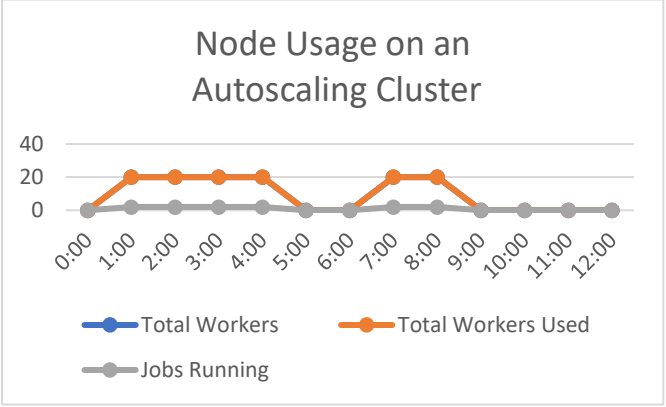

FIGURE 2.Worker node usage illustration for 12 hours on an autoscaling cluster. For simplicity, made upscaling and downscaling time to negligible minutes. Cool down period has been set to 0.

VII. CONCLUSION

Our algorithm is efficient in allocating resources for jobs and the resources are available only for the duration of the job. With a multi-tenant cluster with multiple jobs, the algorithm can a) add more resources on demand, b) transfer unused resources based on demand or c) remove unused resources if there is no demand. Resource cost is attributed to teams and multiple teams running multiple jobs at an enterprise level achieve accountability. By transferring resources, the autoscaling algorithm improves re-usability and negates time taken to add nodes. The algorithm itself is capable of computing the cost of running jobs and can be used to report cost. Cost of running the cluster is cheaper than running a fat cluster. Both actors' jobs and resources achieve best case scenario with jobs utilizing all the resources and running within stipulated time and resources utilized fully for the time up and running.

FUTURE WORK

Future work is to add predictive upscaling features to the algorithm. Based on the schedule information of the job, the next run time of the job can be computed. A smart predictive extension to the algorithm will upscale the cluster just ahead of the job's start time. This will eliminate the wait time for resources to be available. With predictive autoscaling, cooldown period to remove a resource, can be reduced closer to zero. Combining predictive upscaling and immediate termination of un-used resources, jobs will run as soon as they start and resources utilization will be equivalent to uptime.

REFERENCES

[1] A Step towards Hadoop Dynamic Scaling Anshul Gandhi ; Sidhartha Thota ; Parijat Dube ; Andrzej Kochut ; Li Zhang
[2] Using Machine Learning for Black-Box Autoscaling Muhammad Wajahat ; Anshul Gandhi ; Alexei Karve ; Andrzej Kochut
[3] https://hadoop.apache.org/
[4] https://zeppelin.apache.org/
[5] https://jupyter.org/
[6] https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/NodeLabel.html
[7] https://hive.apache.org/
[8] https://ambari.apache.org/

BIOGRAPHIES

Rajkamal Mahamuni Natarajan

Principal Software Engineer

Rajkamal Mahamuni Natarajan received a Master of Science in Birla Institute of Technology and Science, Pilani, India in 2009. He is a Principal Software Engineer at Indeed. At Indeed, He leads Imhotep and Auto join teams in the Data Infrastructure group. He has designed, led and Co-Implemented features like Balancing tasks dynamically in a Distributed Query Engine, inflight Sanitization, Change Data Capture Pipeline, Change Data Capture Sanitization Pipeline, Athena Connector, Benchmarking various open sources like Hudi, Delta and Iceberg.

Raj Pravinbhai Manvar

Senior Software Engineer

Raj Pravinbhai Manvar received Bachelors in Computer Science from International Institute of Information Technology in 2018. He is a Senior Software Engineer at Indeed Inc. At Indeed he led a major change to the client/server communication protocol in Indeed's primary large-scale data analytics platform "Imhotep". Deployed open source workflow orchestration tool "Airflow" and onboarded 35 teams running 1000+ workflows daily. Currently he is working on various initiatives on migrating Indeed's primary data warehouse storage from Imhotep to Hive, primary data transformation tool to Apache Spark, migrating Indeed's data infrastructure to AWS

Thai Bui

Staff Software Engineer

Thai Bui is a Staff software engineer at Bazaarvoice in the Reporting & Analytics team. He enjoys building and tinkering with scalable and distributed systems. Recently, he has been deep diving into making big data and its eco-systems easier and faster to use for engineers and customers of Bazaarvoice, specifically related to accelerating the performance of SQL-on-Hadoop and the speed of collaboration in data engineering.

