Parallel Computing 85 (2019) 66–78

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Multivariate modeling and two-level scheduling of analytic queries

Zhuo Liu[a], Amit Kumar Nath[b], Xiaoning Ding[c], Huansong Fu[b], Md. Muhib Khan[b], Weikuan Yu[b,*]

[a] Department of Computer Science and Software Engineering, Auburn University, AL, United States
[b] Department of Computer Science, Florida State University, FL, United States
[c] Department of Computer Science, New Jersey Institute of Technology, NJ, United States

ARTICLE INFO

Article history:
Received 20 November 2018
Accepted 21 January 2019
Available online 22 February 2019

Keywords:
MapReduce
Multivariate modeling
Query scheduling

ABSTRACT

Analytic queries are typically compiled into execution plans in the form of directed acyclic graphs (DAGs) of MapReduce jobs. Jobs in the DAGs are dispatched to the MapReduce processing engine as soon as their dependencies are satisfied. MapReduce adopts a job-level scheduling policy to strive for a balanced distribution of tasks and effective utilization of resources. However, such simplistic policy is unable to reconcile the dynamics of different jobs in complex analytic queries, resulting in unfair treatment of different queries, low utilization of system resources, prolonged execution time, and low query throughput. Therefore, we introduce a scheduling framework to address these problems systematically. Our framework includes two techniques: multivariate DAG modeling and two-level query scheduling. Cross-layer semantics percolation allows the flow of query semantics and job dependencies in the DAG to the MapReduce scheduler. With richer semantics information, we build a multivariate model that can accurately predict the execution time of individual MapReduce jobs and gauge the changing size of analytics datasets through selectivity approximation. Furthermore, we introduce two-level query scheduling that can maximize the intra-query job-level concurrency, and at the same time speed up the query-level completion time based on the accurate prediction and queuing of queries. At the job level, we focus on detecting query semantics, predicting the query completion time through an online multivariate linear regression model, thereby increasing job-level parallelism and maximizing data sharing across jobs. At the task level, we focus on balanced data distribution, maximal slot utilization, and optimal data locality of task scheduling. Our experimental results on a set of complex query benchmarks demonstrate that our scheduling framework can significantly improve both fairness and throughput of Hive queries. It can improve query response time by up to 43.9% and 72.8% on average, compared to the Hadoop Fair Scheduling and the Hadoop Capacity Scheduling, respectively. In addition, our two-level scheduler can achieve a query fairness that is, on average, 59.8% better than that of the Hadoop Fair Scheduler.

1. Introduction

According to IDC [1], the total digital data generated per year would reach 40,000 exabytes by 2020. Among such a gigantic amount of data, 33% can bring valuable information if analyzed. However, currently only 0.5% of total data can be analyzed due to limited analytic capabilities. MapReduce [2] and its open-source implementation Hadoop [3] have become powerful engines for processing *BigData* and extracting precious knowledge for various business and scientific applications. Analytics applications often impose diverse yet conflicting requirements on the performance of underlying MapReduce systems; such as high throughput, low latency, and fairness among jobs. For example, to support latency-sensitive applications from advertisements and real-time event log analysis, MapReduce must provide fast turnaround time.

Because of their declarative nature and ease of development, analytics applications are often created using high-level query languages. Each analytic query is transformed by a compiler (e.g., Hive) into an execution plan of multiple MapReduce jobs, which is often depicted as a directed acyclic graph (DAG). A job in a DAG can only be submitted to MapReduce when its dependencies are satisfied. A DAG query completes when its last job is finished. Thus, the execution of analytic queries is centered around dependencies among jobs in each DAG, as well as the completion of jobs along the critical path of the DAG. On the other hand, to support MapReduce jobs from various sources, the lower level MapReduce systems usually adopt a two-phase scheme that allocates computa-

* Corresponding author.
E-mail address: yuw@cs.fsu.edu (W. Yu).

https://doi.org/10.1016/j.parco.2019.01.006
0167-8191/

tion, communication, and I/O resources to two types of constituent tasks (map and reduce tasks) from concurrently active jobs. For example, the Hadoop Fair Scheduler (HFS) and Hadoop Capacity Scheduler (HCS) strive to allocate resources among map and reduce tasks to aim for good fairness among different jobs and high throughput of outstanding jobs. When a job finishes, a scheduler will immediately select tasks from another job for resource allocation and execution. However, these two jobs may belong to DAGs of two different queries. Such interleaved execution of queries can cause prolonged execution for different queries and delay the completion of all queries. Besides, the lack of query semantics and job relationships in these schedulers can also cause unfairness to queries of distinct structures, e.g., chained or tree-shaped queries.

Such problems occur due to the mismatch between system and application objectives. While the schedulers in the MapReduce processing engine focus on job-level fairness and throughput, analytic applications are mainly concerned with query-level performance objectives. This mismatch of objectives often leads to prolonged execution of user queries, resulting in poor user satisfaction. As Hive [4] and Pig Latin [5] have been used pervasively in data warehouses, such problems become a serious issue and must be timely addressed. More than 40% of Hadoop production jobs at Yahoo! run as Pig programs [6]. In Facebook, 95% Hadoop jobs are generated by Hive [7].

In this paper, we propose a *multivariate query modeling and two-level scheduling (TLS) framework* that can address these problems systematically. Two techniques are introduced including multivariate DAG modeling and two-level query scheduling. First, we model the changing size of datasets through DAG queries and then build a multivariate model that can accurately predict the execution time and resource usage of individual jobs and queries. Then, based on the multivariate modeling, we introduce two-level query scheduling that can maximize the intra-query job-level concurrency, speed up the query completion, and ensure query fairness. Furthermore, jobs in the same DAG may share their input data, but MapReduce schedulers have difficulties in recognizing the locality of data across jobs. Our two-level scheduler is designed to detect the existence of common data among jobs in the same DAG, and accordingly share the data across jobs.

Our experimental results on a set of complex workloads demonstrate that TLS can significantly improve both fairness and throughput of Hive queries. Compared to HCS and HFS, TLS improves average query response time by 43.9% and 27.4% for the Bing benchmark and 40.2% and 72.8% for the Facebook benchmark. Additionally, TLS achieves 59.8% better fairness than HFS on average. Our contributions from this research are listed as follows.

- We create a multivariate regression model that can leverage query semantics to accurately predict the execution time of jobs and queries.
- We design a two-level scheduling framework that can schedule analytics queries at two levels: the intra-query level for better job parallelism and the inter-query level for fast and fair query completion.
- Using an extensive set of queries and mixed workloads, we have evaluated TLS and demonstrated its benefits in improving system throughput and query fairness.

2. Motivation

In the current MapReduce-based query processing framework, a physical execution plan for each Hive query is generated as a DAG of MapReduce jobs after being processed by the parser and semantic analyzer. The jobs in DAGs are then submitted to Hadoop according to precedence constraints. Traditional Hadoop schedulers such as HCS and HFS are adapted to allocate resources to runnable

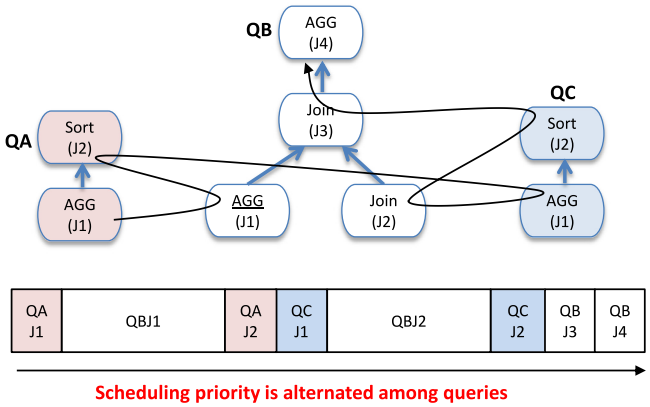


Fig. 1. Under HCS, scheduling priority is alternated among different queries due to semantic unawareness, causing resource thrashing.

MapReduce jobs, which may belong to different DAGs (meaning that the jobs stem from different queries). As we have mentioned in Section 1, the lack of query semantics awareness can result in the inefficient execution of analytic queries, which can be observed in both HCS and HFS. In HCS, it is possible for jobs belonging to different queries to have their executions interleaved. In HFS, resource allocation can be divided too thinly among jobs. Furthermore, the lack of query semantics and job relationship knowledge within schedulers can also cause unfairness to different types of queries. The remainder of this section investigates and proposes a solution for the impact of semantics-unawareness on query scheduling.

2.1. Inefficient interleaved execution of analytic queries

TPC-H [8] represents a popular online analytical workload. We have conducted a test using a mixture of three TPC-H queries which contains two instances of Q14 and one instance of Q17. Q14 evaluates the market response to a production promotion in one month. Q17 determines the loss of average yearly revenue if some orders are not properly fulfilled in time. For convenience, we denote the two instances of Q14 as QA and QC, and the only instance of Q17 as QB. Fig. 1 shows the DAGs for these three queries and their constituent jobs. Both QA and QC have 10 GB of input data and are composed of two jobs: Aggregate (AGG) and Sort. QB is a query composed of four jobs and with larger input data size of 100 GB.

In our experiment, we submit QA, QB, and QC one after another to our Hadoop cluster. Fig. 1 shows that the scheduling priority in this test is alternated among three query jobs under the HCS scheduler, which causes serious resource thrashing among different queries. Since J1 and J2 from QB arrive before QA-J2 and QC-J2, respectively, they are scheduled for execution. As a result, QA-J2 and QC-J2 are blocked from getting container resources and experience execution stalls due to the lack of available containers. Such stalls delay the execution of QA and QC (small queries) by 3 × more than when they are running alone.

When a job is satisfied in HCS, the scheduler immediately selects tasks from another job to receive resource allocation and begin execution. It is possible for these two jobs to exist in the DAGs of different queries. Such interleaved execution of jobs can increase the makespan of different queries, which results in a significant delay of the completion for all queries.

The HFS scheduler is known for its monopolizing behavior, causing different jobs to stalls [9,10]. Applying the same experimental setup as the HCS experiment described above to HFS, we have observed similar execution stalls of QA and QC. For succinct-

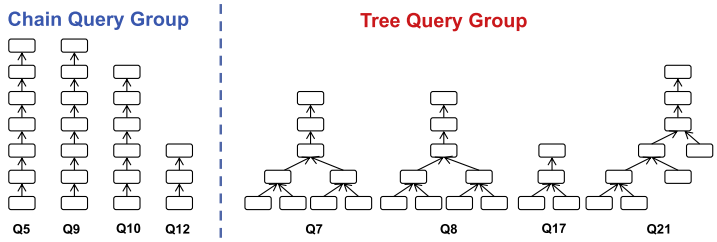


Fig. 2. Chain query group and tree query group.

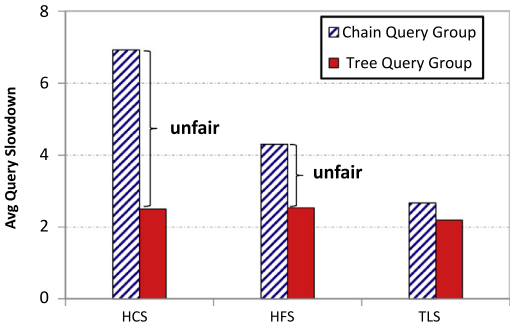


Fig. 3. Fairness to queries of different compositions.

ness, the results are not shown here. Suffice to say that, in HFS, the thinly divided resource allocation among all active jobs can result in suboptimal performance of queries.

Therefore, due to the lack of knowledge about query compositions and actual resource demand, Hadoop schedulers cause execution stalls and performance degradation of small queries. In a large-scale analytics system with many concurrent query requests, the issue of execution stalls caused by semantic-oblivious scheduling is exacerbated.

2.2. Unfairness to queries with different topologies

The lack of query semantics and job relationships at the schedulers can also cause unfairness to different types of queries. For example, some analytic queries possess significant parallelism and they are compiled into DAGs that exhibit a flat star or tree-shaped topology, with many branches such as TPC-H Q18 and Q21 [8]. Other queries do not have much parallelism, thus are often compiled into DAGs that have a chained linear topology, with very few branches. As depicted in Fig. 2, we build two groups of queries: Group 1 (*Chain*) composed of Q5, Q9, Q10, Q12 with a chain topology and Group 2 (*Tree*) composed of Q7, Q8, Q17, Q21 with a tree topology. Both groups of queries are from TPC-H and we submit them simultaneously to the system for execution.

Fig. 3 shows the average execution slowdown of two groups with different scheduling algorithms. Group 1 has an average slowdown much larger than Group 2, about 2.8 × and 1.7 × under HCS and HFS, respectively. This is because the Hadoop schedulers are oblivious to high-level query semantics, thus unable to cope with queries with complex, internal job dependencies. For example, according to HFS, a tree-shaped query that has more branches (more runnable jobs) will be allocated with more resources than a chain-shaped query. Such unfair treatment to queries of different compositions can incur unsatisfactory scheduling efficiency for users. A scheduler that is equipped with high-level semantics can eliminate such unfairness. As shown in Fig. 3, our two-level scheduler (TLS) can leverage the query semantics that is percolated to the scheduler, and complete queries of different DAG topologies under comparable slowdowns.

2.3. Proposed solution: multivariate query modeling and scheduling

Within the Hadoop MapReduce framework, Hive queries are first processed by the parser and the semantic analyzer, and then a physical execution plan is generated as a DAG of MapReduce jobs which are submitted to YARN side according to the precedence constraint. YARN is a new version of Hadoop computing platform, which consists of three major types of components: ResourceManager (RM), NodeManager (NM) and ApplicationMaster (AM). The RM takes charges of job-submissions from clients, monitoring NMs and resource allocation among all applications in units of *containers*. Each container is configured with certain memory and CPU usage limitations. An NM monitors the runtime information of its containers and reports to the RM. A per-application AM is initiated for each job which is responsible for resource negotiation, launching and monitoring tasks.

For all jobs in a DAG, Hive submits one job to the JobListener when that job's dependency has been satisfied. Jobs from various queries are linearly ordered at the Hadoop scheduler, which then assigns map and reduce slots according to either HCS or HFS. Clearly, all the query-level semantics are lost when YARN's ResourceManager receives a job from Hive. Thus the traditional Hadoop scheduler can only see the presence of individual jobs, not their parent queries. As a result, there is no coordination among jobs to ensure the best overall progress for the query. In addition, without global resource usage information of each query, it would be difficult for the scheduler to achieve efficient resource allocation among queries with diversified resource usage. To address these issues systematically, we propose a *Semantics-Aware Multivariate Query Modeling and Scheduling* framework.

In this framework, we extract the query semantics information during the query compilation and its execution plan construction, and then submit this information along with the jobs to the ResourceManager. The semantics information includes query attributes such as the DAG of jobs, dependencies among the jobs, the operators and predicates of the query, and the input tables.

The two main techniques, multivariate query prediction model and two-level query scheduling, are introduced in the framework, as shown in Fig. 4. Query selectivity prediction is designed to evaluate query predicates and estimate the changing size of data during the query execution. Accordingly, we formulate a multivariate regression model to predict the execution times of the jobs in the query. With the help of the model, two-level scheduling is designed to (1) manage queries for efficient resource utilization and ensure fair execution progress at a coarse-grained inter-query level; and (2) leverage our multivariate prediction model to gauge the progress of jobs at the intra-query level, prioritize jobs on the critical path of a DAG, thereby achieving fast overall execution of queries. Multivariate query prediction model and two-level query scheduling are described in detail in Sections 3 and 4, respectively.

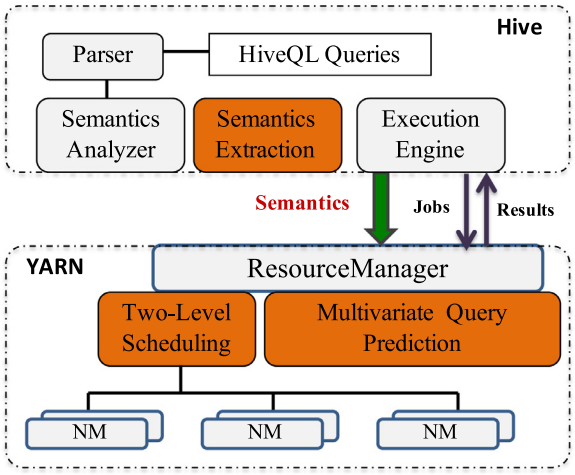


Fig. 4. Semantics-aware multivariate query modeling and scheduling.

3. Multivariate query modeling

Modeling of query execution time and resource usage is critical for efficient scheduling. A query usually consists of multiple MapReduce jobs, each with a separate scan of its input tables, materialization and de/serialization. Therefore, modeling of individual jobs can gain insights about their execution statistics and dynamic behaviors. In addition, there are multiple phases of data processing and movement during the execution of a job, and the data size changes inside a job and along the DAG of a query. A good model also needs to reflect such dynamics of data [11]. We first describe the selectivity estimation of different jobs in a query and then elaborate the integration of selectivity estimation in the job time model.

3.1. Selectivity estimation

For a MapReduce job in a query DAG, the output of its map tasks provides the input for its reduce tasks. One job's output is often taken as part of the input of its succeeding job in the same DAG. The input size of a job directly affects its resource usage (number of map and reduce tasks) during execution in MapReduce. In addition, the data size of map output (intermediate data) affects the execution time of reduce tasks and that of the downstream jobs in the query. Accurate modeling of execution for the job and/or the query requires a good estimation of the dynamic data size during execution. We divide this requirement into two metrics: Intermediate Selectivity and Final Selectivity. Let us denote the input of a job as D_{In}, D_{Med} as the intermediate data, and D_{Out} as the output. The Intermediate Selectivity (IS) is defined as the ratio between D_{Med} and D_{In}, and the final selectivity (FS) as the ratio between D_{Out} and D_{In}.

3.1.1. Intermediate selectivity

In general, a job's intermediate selectivity is determined by the semantics of its predicate and projection clauses on the input tables. For some jobs with a local combine step in its map tasks, e.g., groupby, the impact of combination needs to be taken into account by their intermediate selectivity. Let us denote $|In|$ and $|Med|$ as the numbers of tuples in the input table and the intermediate data, respectively.

The selectivity of a project clause, S_{proj} can be calculated as the ratio between the average width of the selected attributes and the average tuple size in a table. We rely on the table statistics information for estimating S_{proj}.

The selectivity of a predicate clause, S_{pred}, can be calculated as $S_{pred} = |Med|/|In|$ if there is no local combine operation in the map tasks. We build off-line histograms for the attributes of the input table for estimating S_{pred}. Assuming piece-wise uniform distribution of attribute values, equi-width histograms [12] are built on tables' attributes to be filtered through a MapReduce job and stored on HDFS.

When analytic queries are compiled into DAGs, there exist a variety of different operators. The operators for global shuffle/aggregation will be converted into separated MapReduce jobs, which are referred to as *major operators*, including groupby, orderby and join. Other operators will be carried out within the map phase of a job, which are referred to as *minor operators*, such as normal range and equality predicates and map-side join. We categorize jobs into three types with respect to their major operators: groupby, join and extract (including orderby and all other major operators). Next we elaborate further on our estimation of IS for these jobs.

Extract: An extract operation usually scans one input table and we calculate its intermediate selectivity as $IS = S_{pred} * S_{proj}$.

Groupby: In a groupby operation, its local combine can further reduce the intermediate data. Let S_{comb} denote the combination selectivity. We can calculate its intermediate selectivity as $IS = S_{comb} * S_{proj}$. The calculation of S_{comb} needs to be elaborated further with an example. Suppose a job performs a groupby operation on Table T's keys x and y. Let $T.d_{xy}$ denote the product from the numbers of distinct keys for x and y. If the groupby keys are all clustered in the table, S_{comb} is calculated as $S_{comb} = \min(1, \frac{T.d_{xy}}{|T| * S_{pred}}) * S_{pred} = \min(S_{pred}, \frac{T.d_{xy}}{|T|})$. Otherwise, if the groupby keys are randomly distributed, S_{comb} is then calculated as $S_{comb} = \min(S_{pred}, \frac{T.d_{xy}}{|T|/N_{maps}})$, where N_{maps} denotes the number of map tasks in the job. We omit the calculation for other minor cases.

Join: A join operation may select tuples from two or more input tables. We describe the calculation of IS for a simple join job of two tables. Let r_1 represent the percentage of one table's data in the total input of a job, $r_2 = (1 - r_1)$ for the other table. We can calculate IS for a join job with two input tables as: $IS = S_{pred\,1} * S_{proj\,1} * r_1 + S_{pred\,2} * S_{proj\,2} * (1 - r_1)$.

3.1.2. Final selectivity

Let us denote $|Out|$ as the number of tuples and W_{out} the average width of tuples in the output (Out), a job's final selectivity is calculated as $FS = |Out| * W_{Out}/D_{In}$. Specially, $FS = 1$ for map-only jobs. The key of calculating FS is to compute $|Out|$. Our discussion focuses on three common operations including extract, groupby and join.

Extract: In Hive, there are two common extract operations, "*limit k*" and "*order by*". For the former, $|Out| = \min(|In|, k)$; and for the latter, $|Out| = |In|$.

Groupby: A groupby operation may use one or more keys. The number of tuples in the output is determined by the cardinalities of the keys and the predicates on the keys. We show the calculation for an example operation with one key. For a table T, let us denote the cardinality of Key x as $T.d_x$. A groupby operation on key x will have $|Out| = \min(T.d_x, |T| * S_{pred})$.

Join: There are many variations of join operations. Multiple operations may hierarchically formulate as a join tree. We focus on a few common join operations with two or three tables to illustrate the calculation and the most important case: equi-join between a primary key and a foreign key (and tables should obey referential integrity).

An equi-join operation from these two tables T_1 and T_2 will have $|Out| = |T_1 \bowtie T_2| = |T_1| * |T_2| * \frac{1}{\max(T_1.d_x,\ T_2.d_x)}$ if join keys follow uniform distribution [13]. However, uniform distribution is

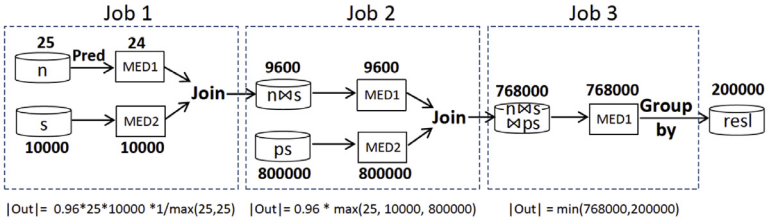


Fig. 5. An example of selectivity estimation.

rare in practical cases; in addition, this approach only applies for multiple joins that share a common join key. In our work, we assume piece-wise uniform distribution, wherein each equi-width bucket keys follow uniform distribution. Let T_{1i} denote the ith bucket in the equi-width histogram for a join operation on Key x. Then we can calculate the number of tuples in a join job's result set as:

$$|T_1 \bowtie T_2| = \sum_{i=1}^{n} |T_{1i}| * |T_{2i}| * \frac{1}{\max(T_{1i}.d_x,\ T_{2i}.d_x)} \quad (1)$$

Since $(T_{1i} \bowtie T_{2i}).d_x = \min(T_{1i}.d_x,\ T_{2i}.d_x)$, the equation above can be evolved to calculate the join selectivity for shared-key joins on three or more tables.

For chained joins with unshared keys, e.g., T_1 and T_2 joining on Key x and T_2 and T_3 joining on Key y, we leverage the techniques introduced in [14] by acquiring the updated piece-wise distribution of Key y after the first join for selectivity estimation of cascaded joins. For natural joins (each operator joins one table's primary key with another table's foreign key) with local predicates on each table, selectivities are accumulated along branches of the join tree, thus the number of tuples in the result set can be approximated as:

$$|T_1.pred_1 \bowtie T_2.pred_2 \bowtie \cdots \bowtie T_n.pred_n| = S_{pred\,1}S_{pred\,2}, \ldots, S_{pred\,n} \max(|T_1|, |T_2|, \ldots, |T_n|) \quad (2)$$

3.1.3. An example of selectivity estimation

We use a modified TPC-H [8] query Q11 as an example to demonstrate the estimation of selectivities. Fig. 5 shows the flow of selectivity estimation. This query is transformed into two join jobs and one groupby job. In Job 1, the predicate on the nation table has a predicate selectivity of 96% and it is relayed to the upcoming jobs along the query tree. Thus we can predict *IS* and *FS* for Job 1 and Job 2 according to the equations above. In Job 3, since the groupby key (partkey) has a cardinality of 200,000 that is much less than input tuples of this job, the output tuples of Job 3 is approximated as 200,000.

```
SELECT ps_partkey, sum(ps_supplycost*
ps_availqty)
FROM nation n JOIN supplier s ON
  s.s_nationkey=n.n_nationkey AND
n.n_name<>'CHINA'
JOIN partsupp ps ON
  ps.ps_suppkey=s.s_suppkey
GROUP BY ps_partkey;
```

3.2. A multivariate time prediction model

Based on the estimation of selectivities, we build a multivariate time prediction model for jobs. We focus on the three operations as we have discussed in Section 3.1: extract, groupby and join. As listed in Table 1, we rely on several input features to model the execution time. First, for simple jobs with the groupby or extract operator, we include three parameters D_{In}, D_{Out} and D_{Med} which can provide sufficient modeling accuracy. Second, different types of jobs display distinct selectivity characteristics. Thus we include the operator type as part of our multivariate model.

Table 1
Input features for the model.

Name	Description
O	The Operator Type: 1 for Join, 0 for others
D_{In}	The Size of Input Data
D_{avgMed}	Avg Intermediate Data Per Reduce Task
D_{Out}	The Size of Output Data
$P(1-P)D_{Med}$	The Data Growth of Join Operators

However, for a join job, these parameters are not enough to reflect the growth of data sizes because the number of tuples can be the Cartesian product of input tables. Let $|T_1|$ and $|T_2|$ denote the number of tuples for the two input tables of a join operator. We define P as the ratio between the number of tuples in the larger filtered table and that of the two filtered tables, i.e., $P = \frac{\max(|T_1|S_{pred1}, |T_2|S_{pred2})}{|T_1|S_{pred1}+|T_2|S_{pred2}}$, $0 < P < 1$. So $P(1-P)$ reflects the factor of a join operator, $P(1-P) \in (0, \frac{1}{4}]$. In our model, we include an additional parameter about the data growth for better prediction accuracy.

Based on these input features, we formulate a linear model with a set of coefficients $\vec{\theta} = [\theta_0, \theta_1, \ldots, \theta_m]$ to predict the job execution time (ET) as:

$$ET = \theta_0 + \theta_1 D_{In} + \theta_2 D_{avgMed} + \theta_3 D_{Out} + \theta_4 O * P(1-P) D_{Med}. \quad (3)$$

Note that $\vec{\theta}$ is trained separately for each of the three different operator types.

More features may lead to better prediction accuracy [15]. However, they can cause more monitoring overhead and are expensive to obtain in real-time. Thus the rationale behind our choice of features is to balance the need of accuracy with the complexity of extracting input features. Note that in the paper we concentrate on selectivity prediction for analytic queries, for other non-relational workloads such as User-Defined Functions (UDFs), there are some available solutions in recent work [16,17].

3.3. Model validation

We further validate the accuracy of our execution prediction for jobs and queries. To validate our model, we build up a training set using queries from TPC-H and TPC-DS benchmarks [8]. The data size ranges from 1 GB to 100 GB. Our validation test uses about 1000 queries, which are converted into 5647 MapReduce jobs. Among them, 3/4 queries are used as the training set while the rest are used as the part of the test set. In addition, we add 150 GB to 400 GB scale queries into the test set for assessing the model's scalability.

The prediction accuracy of our model on job execution time is shown in Fig. 6. The X-axis denotes the actual job execution time while the Y-axis denotes the predicted job execution time. The straight line demonstrates a perfect prediction. We can observe that our model can accurately predict the execution time of

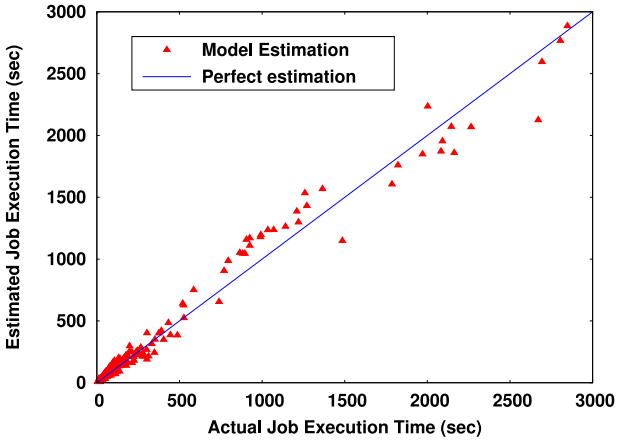


Fig. 6. Accuracy of job execution prediction.

Table 2
Accuracy statistics for job execution prediction.

Types	R-squared accuracy	Avg Error (%)
Extract	83.22%	10.14
Groupby	91.59%	10.48
Join	87.2%	15.67
TestSet	N/A	14.84

MapReduce jobs through a careful process of selectivity estimation based on a few input parameters.

Table 2 summarizes the R-squared accuracy and the average error rate of our model for jobs of each operator. R-squared accuracy shows how well data fit a statistical model and a value approaching 1 indicates a good fit. The average error rate for the test set of jobs is 14.84%.

Built on top of such job time prediction, we can further predict the query execution time with good accuracy. For example, we have applied our model to 22 TPC-H queries (each with 100 GB data) and achieved a low 8.3% error rate on average in modeling the execution time of these queries.

3.4. Validation for predicted query execution

The execution time of a query can be approximated as the sum of execution times of all jobs along the critical path of its DAG and other large jobs which are able to use up the system's resource. Such jobs may include some that are yet to be submitted and others that are actively running. We directly use our multivariate model to predict the execution time of jobs that have not been submitted. For the active jobs, we further improve the prediction accuracy of their execution time by taking into account runtime statistics, such as the execution time and intermediate data size of completed tasks.

We compare the actual execution time and estimated execution time of 100 GB TPC-H queries. As shown in Fig. 7, the average prediction error rate can be as low as 8.3%. Again, this error rate adequately validates the accuracy of our prediction model and its strength from selectivity estimation.

4. Two-level query scheduling

Based on the multivariate query model, our objective is to schedule the queries for better resource utilization, efficiency and fairness. We propose to schedule queries and their internal tasks and jobs at two levels.

Fig. 8(a) shows three Hive queries to a Hadoop system. In the default case, as shown by Fig. 8(b), jobs are admitted as active jobs when their dependencies in a DAG are satisfied, and arranged based on their arrival order. Hadoop then adopts job-based scheduling policies such as HCS and HFS to allocate low-level container resources for these active jobs. Thus Hadoop by default is oblivious to the relationship of jobs within or across high-level queries. In contrast, our proposed scheduling framework will schedule queries and their internal jobs at two levels as shown in Fig. 8(c).

At the coarse-grained query level, an inter-query scheduler selects queries for system efficiency and ensures fairness among concurrent queries. At the fine-grained job level, an intra-query scheduler increases parallelism and adopts input sharing within a query to reduce its processing time. Taken together, two-level scheduling is designed to (1) ensure fair execution progress at a coarse-grained inter-query level; and (2) improve resource utilization and minimize query makespans at the fine-grained intra-query level.

4.1. Inter-query scheduling

When analytic queries are first submitted to our scheduling framework, we admit the query and initialize the structure to keep track of its runtime information, according to our multivariate model. A query queue (L_{act}) is maintained for the active queries each of which contains a DAG of runnable or running jobs. We apply our selectivity estimation and multivariate model recursively from the largest depth of the query DAG to the smallest depth, i.e. the root node. To be specific, D_{In}, IS, D_{Med}, FS and D_{Out} are initialized based on the job type, predicate and projection selectivities as mentioned in Section IV. When one task gets completed, we will update the number of remaining tasks for this query. When one job is completed, we recursively update our estimation of input, output and execution time for the downstream jobs along the DAG.

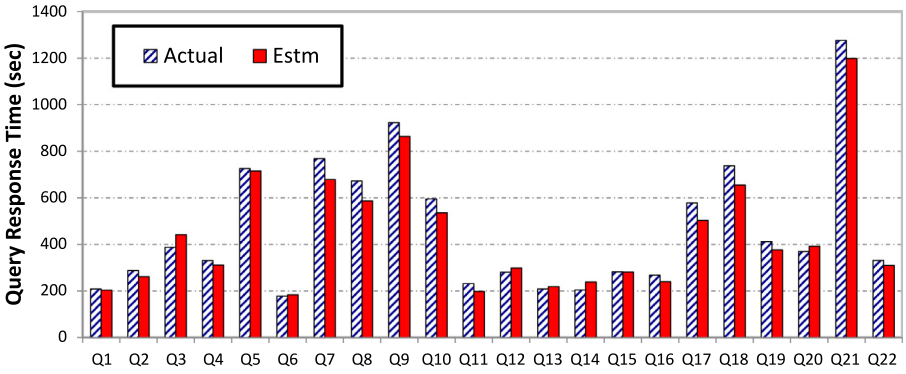
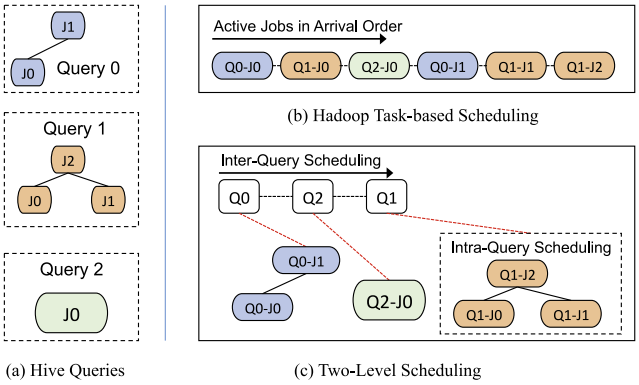


Fig. 7. Accuracy of query response time prediction.



(a) Hive Queries (b) Hadoop Task-based Scheduling (c) Two-Level Scheduling

Fig. 8. Diagram of a two-level scheduling framework.

Table 3
Accuracy for task execution prediction.

Types	Map task	Extract	Groupby	Join
R-squared accuracy	82%	95.94%	92.61%	80.87%

4.1.1. Selection metric – Weighted resource demand

We need to ensure that all queries would be fairly treated with comparable slowdown ratios so that small queries can turn around faster while big queries still get their fair share of time and resources for execution if delayed by a certain degree. However, in selecting queries that are organized as DAGs of jobs, we cannot solely rely on the temporal resource demand of a query, i.e. its remaining time that can be estimated from our multivariate model. A query and all of its jobs often employ a dynamic number of tasks during its execution. Each task may have its own execution time, CPU, memory and I/O resource demand. Therefore, we introduce a simple metric called **W**eighted **R**esource **D**emand (*WRD*) to quantify the resource demand of individual tasks in a query. *WRD* is intended as a metric to estimate the resource requirements of a query or a job. A query's *WRD* is calculated as:

$$WRD = \sum_{i=1}^{N} \alpha_i * MT_i * N_{Mi} + \sum_{j=1}^{K} \beta_j * RT_j * N_{Rj}, \quad (4)$$

where MT_i denotes the resource demand from the ith map task, N_{Mi} denotes the number of remaining map tasks for an arbitrary job i of the query. Similarly, RT_j denotes the resource demand for the jth reduce task and N_{Rj} the number of remaining reduce tasks. α_i and β_j are coefficients introduced to capture the linear relationship between the resource demand of a task and its execution time. For a job that is large enough to occupy all the available containers of the system, its execution time can be approximated as the job's *WRD* divided by the number of available containers plus scheduling overheads.

Even though we have a job-level multivariate linear model, the ranges of the parameter values for various jobs can sometimes go far beyond our training set, thereby causing underestimation of job execution time [18]. To deal with such an issue, we further empirically build prediction models for the average execution time of a job's map or reduce tasks based on the task type, the operator type, job scale, the per-task input size, and output size. Table 3 demonstrates the R-squared accuracy for map tasks and reduce tasks with three types of operators. Such a close estimation of task execution also allows us to determine the WRDs of all queries. We can then select the best query for execution.

4.1.2. Query scheduling for efficiency and fairness

We have introduced an inter-query scheduling algorithm for Query Efficiency and Fairness (QEF) management. It strives to reconcile efficiency and fairness among concurrent queries within the same queue (L_{act} in Algorithm 1). For optimal scheduling efficiency at the inter-query level, we adopt an **SWRD** policy that orders the queries with the *Smallest WRD* at the head of L_{act}. This heuristic algorithm is expected to achieve comparable query scheduling performance as SRPT (shortest remaining processing time) does in M/G/1 queue (see a brief proof in Section 4.1.3).

Algorithm 1 Query Efficiency and Fairness Management.

1: L_{act}: {a list of queries in the ascending order of WRD.}
2: L_{slow}:{a list of queries that have exceeded the slowdownthreshold in the ascending order of slowdown.}
3: **for all** $Q \in L_{act}$ **do**
4: **if** $(Q.slowdown > 2 \times D_{threshold})$ **then**
5: Schedule Q via Algorithm 2
6: Return
7: **end if**
8: **if** $(Q.slowdown > D_{threshold})$ **then**
9: L_{slow}.add(Q)
10: **end if**
11: **end for**
12: **if** $sizeof(L_{slow}) > Limit_{slow}$ **then**
13: $Q_{sched} \leftarrow$ {last query in L_{slow}}
14: **else**
15: $Q_{sched} \leftarrow$ {first query in L_{act}} //SWRD
16: **end if**
17: Schedule Q_{sched} via Algorithm 2

As shown in Algorithm 1, QEF includes the SWRD-based selection policy and a fairness guarantee policy, which addresses potential starvation and fairness issues among queries. In particular, all the queries are sorted within L_{act} according to their WRD requirement (Line 1). Our algorithm selects the query with the least WRD (Line 15). However, to ensure fairness, we look for the query that has been severely slowed down in L_{act} and prioritize it (Lines 4–7). Meanwhile, the query with slow progress is put into another list L_{slow} (Lines 8–10). QEF checks the size of L_{slow} and schedule the last query in L_{slow} if its size exceeds a configurable threshold $Limit_{slow}$ (Lines 12–13).

QEF computes the slowdown that each query experiences by considering each query's sojourn time $T_{sojourn}$ and estimated remaining execution time T_{rem}. The slowdown is defined as

$$slowdown = \frac{T_{sojourn} + T_{rem}}{T_{alone}}. \quad (5)$$

$T_{sojourn}$ is the amount of time the query has spent in the system since its submission. T_{alone} denotes the estimated execution time of the query when it runs alone in the system. Meanwhile, the threshold $D_{threshold}$ that determines whether a query has been unfairly treated is computed as $\frac{1}{1-\rho}$, where ρ is the accumulated load on the system. Such threshold exhibits expected slowdown with the Processor-Sharing (PS) policy as proven by M/G/1 model [19].

4.1.3. Proof for SWRD

According to the Little's Law [20], a schedule for minimizing average response time translates to a schedule for minimizing the average number of queries in a system. Similar to the proof introduced in [21], let $N(t)^{SWRD}$ and $N(t)^{\varphi}$ denote the number of queries residing in the system for the SWRD scheduling policy and for any other policy, respectively. For the J queries with the largest WRD and $J <= \min(N(t)^{SWRD}, N(t)^{\varphi})$, we have $\sum_{i=1}^{J} WRD_i^{SWRD} >= \sum_{i=1}^{J} WRD_i^{\varphi}$ because SWRD favors the queries with smallest WRD. With the assumption that the possible resource utilization difference caused by job phase independence and intra-query dependence is negligible, the remaining workloads (WRDs) at any

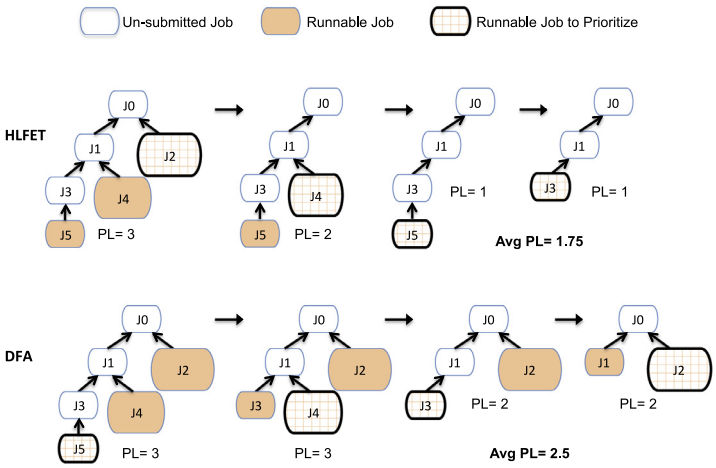


Fig. 9. Comparison between HLFET and DFA algorithms.

time should be the same for any scheduling algorithm, thus we have $\sum_{i=1}^{N(t)^{SWRD}} WRD_i^{SWRD} = \sum_{i=1}^{N(t)^{\varphi}} WRD_i^{\varphi}$. Therefore, $N(t)^{SWRD} <= N(t)^{\varphi}$.

4.2. Intra-query scheduling

At the intra-query level, our target is to minimize the makespan of a query that consists of a DAG of MapReduce jobs. This problem is analogous to the multiprocessor scheduling problem for a DAG of tasks. The HLFET algorithm [22] is able to achieve the best makespan for the scheduling of DAGs of parallel tasks. The *level* of a task is calculated as the total execution time of all constituent tasks along its longest path and the task at the highest level is prioritized in HLFET. However, the execution of DAGs for analytic queries on MapReduce systems is very different from the DAGs of parallel tasks on a multiprocessor environment because each job in the DAG of a query requires a collection of map and reduce tasks, i.e., causing rounds of resource allocation and task scheduling. Therefore, the HLFET algorithm is not a good fit to achieve the minimal makespan for DAGs of analytic queries. It can cause insufficient job parallelism and underutilization of system resources.

Depth-first algorithm: Based on the internal complexity of MapReduce jobs in the DAGs, we design an algorithm that would first prioritize the job with the largest depth. In addition, for jobs of the same depth, our algorithm prefers the job with a larger WRD of the path from this job to the root node. We refer to this algorithm as the Depth-First Algorithm. As shown in Fig. 9, a query is compiled into a DAG of six jobs. J2 and J4 are big jobs with highest levels in the HLFET algorithm. Thus HLFET schedules jobs J2, J4 and J5 in sequence according to their levels. In four steps of job scheduling, it can only achieve a job parallelism (PL) of 1.75 on average. System resources can be under-utilized with very low job parallelism. In contrast, DFA chooses the jobs with higher depths, whose results are needed by more downstream jobs. In the same number of steps, DFA achieves a job parallelism of 2.5 on average. When prioritizing one job with the biggest depth, the remaining system containers can be leveraged by other concurrent jobs for boosting the progress of the whole query. Thus DFA recognizes and leverages a query's DAG structure to achieve better job parallelism, thereby speeding up query execution.

Locality through input sharing: To further strengthen DFA, we exploit input-sharing opportunities for better memory locality. For example, as shown in Fig. 10, the TPC-H query Q21 contains two groupby (AGG) jobs and a join job that share the *lineitem* table as their input, an opportunity for intra-query table sharing. This input

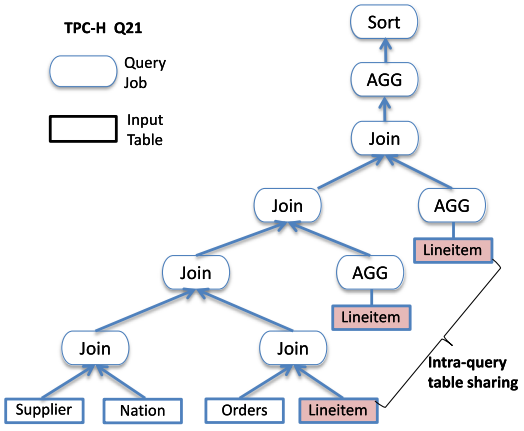


Fig. 10. An example of table sharing for a TPC-H query.

locality can be exploited to achieve better memory locality and reduce disk I/O. Note that input tables can be shared across different queries, e.g., between Q21 and Q17 (not shown for succinctness). Exploiting inter-query input sharing would complicate our design with diminishing returns. We focus on intra-query input sharing opportunities in this paper.

Combined algorithm: We propose a Locality-Based Depth-First Algorithm (LoDFA) to combine both ideas. As shown in Algorithm 2, LoDFA first initializes the depth, WRD and input tables for each job (Lines 5–6). In addition, for each input table, it creates a set that includes the jobs that share the table (Lines 7–11). Once a container is assigned to this query, it finds the sets of jobs that share the input table (e), and schedules a task from the job with the largest job WRD (Lines 16–21). This allows LoDFA to exploit the benefit of memory locality by opportunistically launching the batch of sharing-input jobs together similar to the fairshare manner [23]. If there are no runnable sharing-input jobs in this query, LoDFA then follows the DFA algorithm to select the job with the highest depth and then the largest accumulated WRD along the path from this job to the root node (Lines 22–25).

5. Evaluation

In this section, we carry out extensive experiments to evaluate the effectiveness of the framework with a diverse set of analytic query workloads.

Algorithm 2 Locality-Based Depth-First Algorithm.

Initialization:
$DAG(Q), Ready(Q), IT(Q) \leftarrow$ {Query Q's DAG, Runnable jobs, Input Tables}
$LA(e)$: {Jobs sharing Table e, first empty.}
for all $j \in DAG(Q)$ **do**
  $Depth_j, WRD_j, Input_j \leftarrow$ {Job j's depth, WRD, tables}
  Insert Job j into $Ready(Q)$ if its dependencies are ready.
  for all $e \in IT(Q)$ **do**
    if $e \in Input_j$ and $e.size > Input_j.size/2$ **then**
      Insert Job j into $LA(e)$ in descending WRD.
    end if
  end for
end for
Method:
A container is assigned to this query by Algorithm 1
$e \leftarrow$ the max table among the query's inputs whose $LA(e).size >= 2$
if $LA(e) <> null$ **then**
  Select a runnable Job k with the max job-WRD in $LA(e)$
  Allocate the container to Job k
  Check and update WRD_k, $LA(e)$ and $Ready(Q)$
  Return
end if
Select jobs with the highest depth from $Ready(Q)$ as L_{todo}
In L_{todo}, select Job k with the max WRD from the root
Allocate the container to Job k
Check and update WRD_k and $Ready(Q)$

5.1. Experimental settings

Testbed: We have implemented our prediction-based scheduling framework in Hive v0.12.0 and Hadoop v2.5.0 (YARN). Our experiments are conducted on a cluster of 17 nodes, one of which dedicatedly serves as both the ResourceManager of Hadoop MapReduce and the namenode of HDFS while each of the other nodes serves as both the NodeManager and datanode. Every node features two 2.67 GHz hex-core Intel Xeon X5650 CPUs, 24 GB memory and two 500 GB Western Digital SATA hard drives. The heap size for JVM is set as 2 GB and the HDFS block size as 256 MB. All other Hadoop parameters are same as the default configuration. We employ Hive with the default configuration, while allowing the submission of multiple jobs into Hadoop.

Benchmarks and workloads: We choose a wide spectrum of benchmarks to conduct the experiments. We first choose a few TPC-H queries with tree-structure execution plans to examine the efficacy of the intra-query scheduling. Then, we test the overall performance of the two-level scheduling under large-scale workloads with concurrent queries. For this purpose, with the TPC-H and TPC-DS queries, we first build two workloads based on the workload composition on Facebook and Bing production systems characterized in [24]. We name them *Facebook workload* and *Bing workload*, respectively. Though our framework is mainly targeted for Hive queries, we also test the feasibility of the framework on scheduling mixed workloads consisting of singleton MapReduce jobs and Hive queries. For these experiments, we build the *QMix workload*, which mixes TPC queries with non-Hive benchmarks (Terasort, WordCount and Grep, processed as single-job queries).

Table 4 summarizes the composition of the Bing, Facebook and QMix workloads. Each workload has 100 queries with different input sizes and these queries are divided into 5 bins based on their input sizes. We carefully tune the scales of the data sets and select queries, such that the numbers of queries in each bin follow a similar distribution as that described in [24]. While Facebook workload has a dominant portion of queries with small input sizes, queries of the Bing workload are more uniformly distributed in the bins. The QMix workload is built by replacing 20 Hive queries in the Facebook workload with 20 non-Hive MapReduce jobs such that the QMix workload follows a long-tailed distribution similar to the Facebook workload. The queries are submitted into the system following a random Poisson distribution of inter-arrival times.

Table 4
Workload characteristics.

Bin	Input Size	Number of Queries		
		Bing	Facebook	QMix
1	1–10 GB	44	85	85
2	20 GB	8	4	4
3	50 GB	24	8	8
4	100 GB	22	2	2
5	> 100 GB	2	1	1

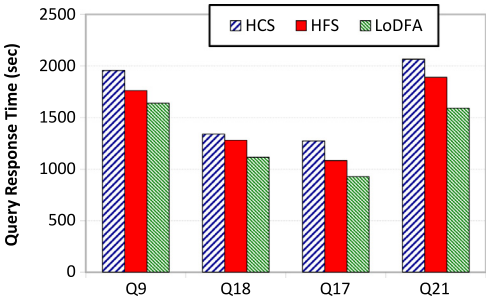


Fig. 11. Query response times of Q9, Q18, Q17, and Q21 when they use system resources alone.

5.2. Intra-query scheduling evaluation

To test the effectiveness of the intra-query scheduling algorithm LoDFA, we collect the response time of each query when it uses the whole system dedicatedly. While queries with chain-structured execution plans usually have similar response times under LoDFA as they do under conventional Hadoop schedulers HCS and HFS, we found that LoDFA can effectively reduce the response times for queries with tree-structured execution plans. Fig. 11 illustrates the response times of a few representative TPC-H queries of 200 GB scale under LoDFA and conventional Hadoop schedulers.

Compared to HCS, LoDFA reduces the response times of Q9 (tree-structured version), Q18, Q17, and Q21 by 16.3%, 16.7%, 27.0% and 23.0%, respectively. Compared to HFS, LoDFA reduces the response times of Q9, Q18, Q17 and Q21 by 7.1%, 12.9%, 14.2% and 15.8%, respectively. LoDFA improves the performance of Q9 and Q18 mainly because it preferentially schedules jobs such that it increases the number of concurrent jobs in these queries to fully utilize resources. To be specific, our DFA favors jobs with large depths and WRDs which are on critical paths. LoDFA improves the performance of Q17 and Q21 mainly since it is aware of the data sharing between the jobs in each query and schedules the jobs in a way that can exploit memory locality for efficient execution.

Since in Q17 and Q21 the jobs on the leaf nodes of their execution plans share the same big table (lineitem), the sharing-aware scheduling in LoDFA can consecutively execute the tasks of jobs sharing the same data sets and improve data accesses' temporal locality for these jobs. Such strategy reduces the amount of data to be read from disks and accelerates the execution of map and reduce tasks.

When queries run concurrently and contend for resources, the intra-query scheduling algorithm - LoDFA becomes more effective in reducing query response times, especially for small queries. At

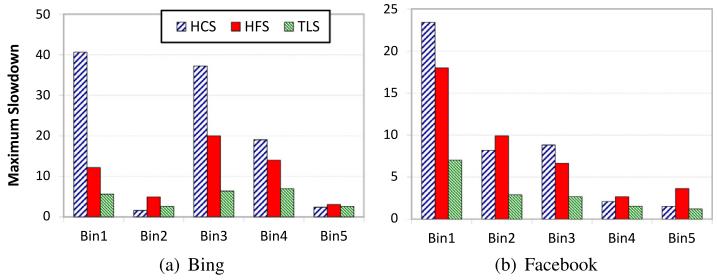


Fig. 12. Maximum slowdown.

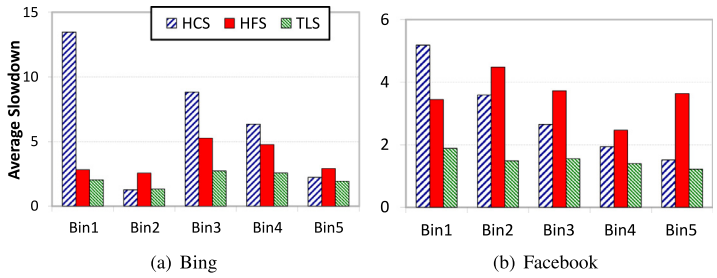


Fig. 13. Average slowdown.

the same time, it improves the data temporal locality among jobs that share data and hence increases system throughput.

5.3. Improving scheduling fairness

For a system shared by queries from multiple users, it is desirable to schedule the queries fairly. To measure the fairness, we run the concurrent query workloads (Bing and Facebook) and collect the slowdown of each query, which is calculated as the ratio of the response time of a query to its response performance when executing alone. In an ideal system, we expect the queries to be slowed down by similar percentages.

Fig. 12 shows the maximum slowdown of the queries for each bin in Bing and Facebook workloads. A smaller value of the maximum slowdowns indicates that the queries in the corresponding bin are more fairly scheduled, since it is the upper bound of the slowdown difference between any two queries in the bin. In Fig. 13, we show the average slowdown of the queries in each bin. By comparing the average slowdowns of different bins, we can estimate how fairly the queries across these bins are scheduled.

Not surprisingly, for both Bing and Facebook workloads, when they run with the HCS or HFS scheduler, the maximum slowdowns of the bins with large queries are usually lower than those of bins with small queries. For example, as shown in Fig. 12, the maximum slowdowns of their first bins are higher than those of any other bins. This indicates that with HCS or HFS the performance of small queries are more subject to unfair scheduling than large queries. Fig. 12 also shows that though replacing HCS with HFS generally helps improve the fairness for the bins to some extent, HFS still cannot render satisfactory fairness among different query bins. The main reason is that HFS strives to achieve job-level fairness but lacks the semantic information at query-level. Therefore, HFS cannot achieve good fairness among queries with different input sizes and DAG structures (e.g., tree-shaped and chain-shaped topologies described in Fig. 2). However, TLS achieves much lower maximum slowdowns for all the bins than HCS and HFS. To be specific, for Bing and Facebook workloads, TLS reduces the means of maximum slowdowns by 75.9% and 65.2% compared to HCS; compared to HFS, the improvement percentages are 55.2% and 62.5%, respectively. This confirms that TLS can improve fairness significantly and consistently, irrespective of the query input sizes.

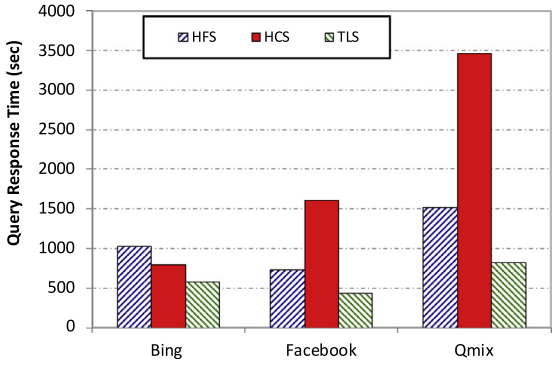


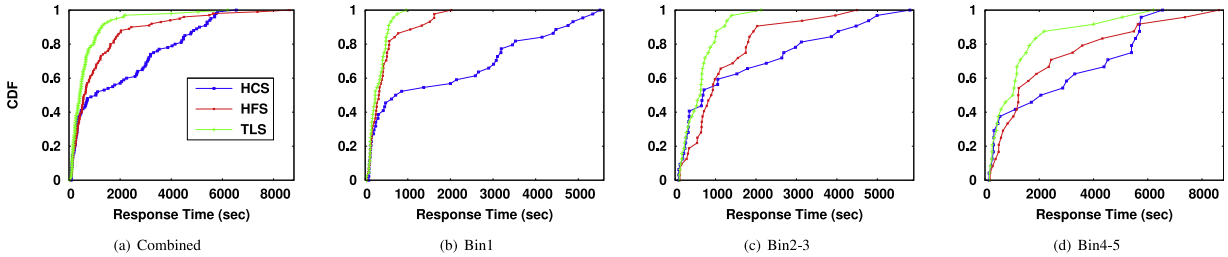
Fig. 14. Average query response times.

As shown in Fig. 13, with HCS, small queries are biased by the scheduler, and they are usually slowed down by higher percentages than large queries. For example, for the Bing workload, HCS incurs 13.5 × average slowdown for the queries in Bin1 but 2.23 × for Bin5. Though HFS may improve the fairness, the improvement comes at the cost of the performance of large queries. For example, compared to HCS, HFS significantly increases the average slowdowns of Bin4 and Bin5 in the Facebook workload. However, TLS can improve fairness more effectively than HFS. With TLS, the average slowdowns of different bins show much smaller variations than those with HFS. More importantly, the better fairness comes without degrading the performance. For all the bins, the average slowdowns are much lower than those with either HFS or HCS.

5.4. Overall benefits to query response time

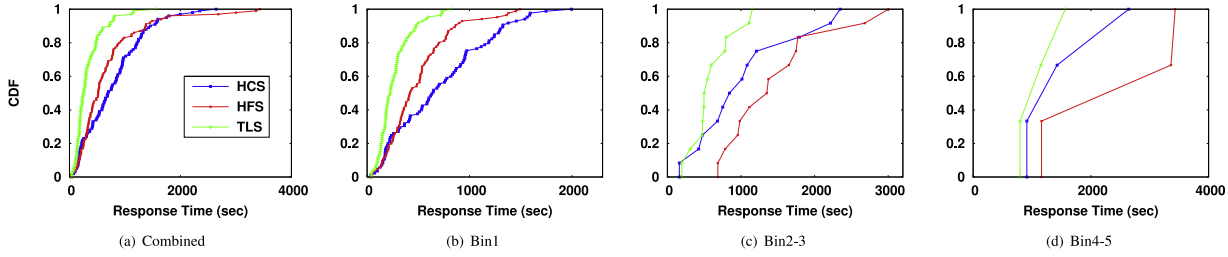
We also use Bing, Facebook and Qmix workloads to evaluate the overall performance of our framework.

Fig. 14 summarizes the average query execution times of Bing, Facebook, and Qmix workloads under three scheduling policies. Among all scheduling policies, TLS achieves the best performance by leveraging the semantics information that ensures all queries



(a) Combined (b) Bin1 (c) Bin2-3 (d) Bin4-5

Fig. 15. CDF of query response time in bing workload.



(a) Combined (b) Bin1 (c) Bin2-3 (d) Bin4-5

Fig. 16. CDF of query response time in Facebook workload.

are fairly treated with negligible slowdowns. HCS has reversed performance compared to HFS, because of the characteristics of HCS that maximizes the utilization of the cluster. As the resources always have the higher priority in HCS, the semantics are broken severely under HCS than HFS. However, HFS neglects the resources utilization. TLS considers the resource demands and the time, which avoids breaking the semantics and delaying priorities. Compared to HFS, TLS reduces the average query response times by 40.2% and 43.9%, respectively. Compared to HCS, the average query response times of TLS are decreased by 72.8% and 27.4%, respectively. For QMix workload, TLS reduces the average query response time by 45.96% and 76.28% respectively, relative to HFS and HCS. This clearly demonstrates the capability of TLS to handle query workloads with the help of semantics-aware query prediction.

An analysis on response times: Besides an evaluation on the overall performance benefits, we have analyzed the distribution of response times with semantics-aware scheduling. Figs. 15 and 16 plot the CDFs of the query response times for these scheduling schemes under Bing and Facebook workloads. As we can observe, TLS consistently reduces the response times for almost all the queries.

Allocating more resources to small jobs sacrifices the performance of large jobs. This can be confirmed with the HFS curves in Figs. 15(c) and (d). Thus, the Bing workload, which has more queries with large inputs, exhibits larger average response time with HFS than it does with HCS.

In the Facebook workload, most queries have small input sizes and thus have small jobs. With the HCS scheduler, the executions of the small jobs are significantly delayed due to the interleaving of the execution of large queries. The HFS scheduler reduces the response times for small queries by allocating a fair amount of resources to small jobs (Fig. 16(b)). This is why the average query response time is smaller with HFS than that with HCS. TLS outperforms both HFS and HCS due to help from query-based semantic-aware scheduling and accurate query prediction.

5.5. Comparison against the built-in schedulers of YARN

System response time: When queries run concurrently and contend for resources, TLS becomes effective in reducing query response time. This is especially true for small queries. To test how TLS performs compared to HCS and HFS, we submit 5 queries with

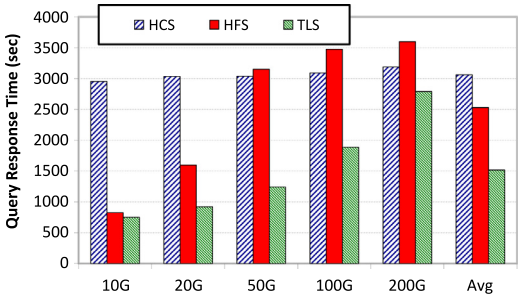


Fig. 17. Query response times for Q21 in TPC-H.

the same query number over different inputs (with sizes ranging from 10 GB to 200 GB) simultaneously to the system. We collect and compare the response times of each query under HCS, HFS, and TLS.

Fig. 17 shows the response times of each query, as well as the average response time of Q21 in TPC-H. Overall, TLS is substantially more responsive than HCS and HFS, especially for the small queries. With HCS, five queries have similar response times. This is because HCS groups jobs into multiple queues and adopts the FIFO scheduling policy for each queue. This means that HCS is imposing the bias that FIFO holds towards small queries, leading to the execution priority turning around among each of the five queries. Compared to HCS, TLS reduces the response times of small queries to a large extent (up to 74.6% for the query with 10 GB input). Concerning HFS, it splits resources among jobs so that each job receives its fair share, which allows small queries turn around faster, while large queries progress slowly. Throughput indicates the number of queries that a MapReduce cluster handles successfully during an interval and is considered as an primary unit of measurement in scheduling. Fig. 17 also shows that TLS reduces the response time for the query that finishes last, indicating the increase of the system throughput compared to HCS and HFS. To be specific, TLS reduces the response time by 36.1% and increases system throughput by 22.5% on average compared with HFS. The reason of these performance improvements is that the semantic information of TLS enables it to run queries in sequence while HFS runs them in parallel. By running queries sequentially, TLS allocates as much resource as a query demands so that the query can finish quickly

without penalizing other queries, leading to increased overall performance. At the same time, the LoDFA at the intra-query level improves the number of concurrent running jobs and the data locality among jobs that share a common input within a query. It is important to note that TLS not only caters to the interactivity requirement of small interactive queries, but also to the performance requirements of large queries.

6. Related work

In this section, we review recent work on query modeling and scheduling in the MapReduce environment.

6.1. Query modeling

Morton et al. [16] proposed the Paratimer model to estimate the progress of Pig queries, which are translated into DAGs of MapReduce jobs. Their model incorporated dataset cardinality and unit processing time. Verma et al. [17,25] provided different "work/speed" time models for the relationship of execution time and available resources. Both time models require pre-execution profiling or debug runs of the same job in order to acquire the necessary phase information to estimate the job's execution time. Our work builds on top of its precursor [26] and exploits selectivity estimation and does not require such pre-execution or debug runs of MapReduce queries.

In [27], Mullin proposed a partial bloom filter based join selectivity estimation algorithm. In [14], Bell et. al presented a piecewise uniform approach for estimating the frequency distribution of join attributes by equal-width histograms. Our paper can utilize their method for join selectivity estimation without the uniform and independent distribution assumption. In [28], Dell'aquila exploited the canonical coefficient parameters of estimating selectivity factors for relational operations through approximating both the multivariate data distribution and distinct values of attributes. Swami and Schiefer [13] attempted selectivity prediction for multi-join operations with a common key and uniform key distribution. Our work extends this prior study to provide selectivity prediction for multi-join operations on different keys with piece-wise uniform distribution in the MapReduce environment.

Wu et al. [15] proposed AQUA as a comprehensive cost model to estimate CPU, network and I/O costs of database operations and MapReduce jobs. Our work is different from AQUA as a time based model and includes selectivity estimation for different types of query jobs. Li et al. [18] estimated resource demands of queries using statistical models for individual operators of database queries. Ganapathi et al. [29,30] developed a KCCA (Kernel Canonical Correlation Analysis) model based prediction system to solve the database query time estimation. Compared to these studies, we design a multivariate model for queries in MapReduce-based data warehouse systems.

6.2. MapReduce job scheduling

Algorithms for scheduling jobs and/or DAGs of jobs have been studied based on general models. For example, the Johnson's algorithm [31] was proposed to solve two and three-stage Job-shop problems. HLFET was proposed by Adam et al. [22] as a scheduling algorithm for DAGs of jobs in a multiprocessor environment. Similarly, Hu [32] proposed a polynomial schedule algorithm for in-tree structured DAGs with unit computation cost. These algorithms cannot be directly applied to schedule analytic Hive queries due to the special features of Hive queries and frameworks. For example, Johnson's algorithm is not applicable for query scheduling due to the precedence constraint of jobs. In addition, parallelism level in MapReduce environment is flexible and job nodes in the query DAG are of non-uniform costs, which are different from the scenarios in [22,32].

Targeting the scheduling of individual MapReduce jobs, various algorithms were proposed. For example, Zaharia et al. [33] proposed delay scheduling to promote data locality in the scheduling of MapReduce jobs. Wolf et al. proposed a malleable scheduler *Flex* for optimizing the minisum and minimax metrics of response time, stretch, deadlines, etc [34]. They are not aware of the relationship of jobs in a DAG. Luo et al. [35] identified and transmitted critical semantic information from DBMS down to the hybrid storage layer, enabling the adoption of different QoS policies for different queries. Shenker et al. [36] implemented Shark as a scalable SQL and Rich Analytics on top of Spark. Yu et al. [37] designed the DryadLINQ model for users to conduct declarative operations on distributed datasets. A DryadLINQ program is translated into an execution plan graph where each vertex is to execute as a Dryad job on the cluster-computing infrastructure. Ke et al. [38] provided a framework called Optimus for dynamically rewriting EPG (Execution Plan Graph) at runtime in DryadLINQ. Compared to these studies, our work leverages semantic information from DAG queries for modeling of job execution time and resource usage and then employs the model for developing an efficient query scheduler. Unlike the aforementioned studies, our work advances query processing capability of MapReduce, and enables support for query level scheduling.

6.3. Mapreduce query scheduling

MapReduce job scheduling has already received much attention and has been studied extensively. Beyond job scheduling mechanisms, there are several interesting research efforts for scheduling queries like our paper. Oozie [39] was proposed by Yahoo! as a scalable workflow scheduler built on top of Hadoop. The Oozie server accepts textually specified workflow DAGs submitted by multiple Oozie clients, splits these workflows into sub-tasks, and dispatches the sub-tasks to a Hadoop cluster for processing. However, a user must specify some parameters for the job chronology when submitting a workflow job. Zhang et al. [40] offers a performance modeling environment that automatically profiles jobs from past runs and estimates the required resources for completing a Pig program within a given deadline so that it can meet SLO. MRShare [41] and CoScan [42] offer a similar automatic scheduling framework that merges the execution of MapReduce jobs with common data inputs in such a way that this data is only scanned once and the entire workflow completion time is reduced.

All of the above approaches study the Apache Pig framework. In contrast, we demonstrate our proposed scheduler using the more widely adopted framework, Hive, whose success in large-scale analysis greatly inspired our work. A performance comparison is conducted in [43] which shows that Hive is more efficient than Pig.

7. Conclusion

Many popular data warehouse systems are deployed on top of MapReduce. Complex analytic queries are usually compiled into directed acyclic graphs (DAGs). However, the simplistic job-level scheduling policy in MapReduce is unable to balance resource distribution and reconcile the dynamic needs of different jobs in DAGs. To address such issues systematically, we first develop a semantic-aware query prediction framework which includes three main techniques: semantic percolation, selectivity estimation and multivariate time prediction for analytic queries. Our framework is able to bridge the semantic gap between Hadoop and Hive. In addition, the multivariate query prediction can accurately predict the changing data sizes and the execution time of jobs in DAG queries.

In addition, a two-level scheduling scheme is designed to allocate resources and schedule tasks at both inter-query and intra-query levels for efficient and fair execution of concurrent queries. The query prediction framework offers important inputs for scheduling decisions in Hadoop scheduler. Experimental results demonstrate that our semantic-aware framework can achieve accurate query prediction and enable more efficient query scheduling than traditional Hadoop schedulers. Furthermore, our two-level scheduler significantly enhances query fairness.

Acknowledgments

We are very thankful for the helpful discussions and comments from many graduate students from the CASTL group, particularly Mr. Fang Zhou and Ms. Lizhen Shi. This work is funded in part by the U.S. National Science Foundation awards 1561041 and 1564647.

References

[1] J. Gantz, D. Reinsel, The digital universe in 2020: big data, bigger digital shadows, and biggest growth in the far east, IDC iView: IDC Analyze the Future (2012).
[2] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in: Proceedings of the OSDI'04, USENIX, Berkeley, CA, USA, 2004.
[3] Apache Hadoop Project, http://hadoop.apache.org/.
[4] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, R. Murthy, Hive-a petabyte scale data warehouse using hadoop, in: Proceedings of the ICDE, IEEE, 2010, pp. 996–1005.
[5] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: Proceedings of the SIGMOD, ACM, New York, NY, USA, 2008, pp. 1099–1110.
[6] A.F. Gates, O. Natkovich, S. Chopra, P. Kamath, S.M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, U. Srivastava, Building a high-level dataflow system on top of map-reduce: the pig experience, in: Proceedings of the VLDB Endowment, 2, 2009, pp. 1414–1425.
[7] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, X. Zhang, YSmart: yet another SQL–to-MapReduce translator, in: Proceedings of the Thirty-First International Conference on Distributed Computing Systems (ICDCS), IEEE, 2011, pp. 25–36.
[8] TPC, http://www.tpc.org/.
[9] Y. Wang, J. Tan, W. Yu, L. Zhang, X. Meng, X. Li, Preemptive reducetask scheduling for fair and fast job completion, in: Proceedings of the ICAC, 2013, pp. 279–289.
[10] J. Tan, X. Meng, L. Zhang, Delay tails in MapReduce scheduling, in: Proceedings of the Twelfth ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, ACM, New York, NY, USA, 2012, pp. 5–16, doi:10.1145/2254756.2254761.
[11] J. Duggan, U. Cetintemel, O. Papaemmanouil, E. Upfal, Performance prediction for concurrent database workloads, in: Proceedings of the SIGMOD, ACM, 2011, pp. 337–348.
[12] G. Piatetsky-Shapiro, C. Connell, Accurate estimation of the number of tuples satisfying a condition, in: Proceedings of the SIGMOD, ACM, 1984.
[13] A. Swami, K. Schiefer, On the estimation of join result sizes, Technical Report, IBM, 1993.
[14] D.A. Bell, D. Link, S. McClean, Pragmatic estimation of join sizes and attribute correlations, in: Proceedings of the ICDE, IEEE, 1989, pp. 76–84.
[15] S. Wu, F. Li, S. Mehrotra, B.C. Ooi, Query optimization for massively parallel data processing, in: Proceedings of the SOCC, ACM, 2011, p. 12.
[16] K. Morton, M. Balazinska, D. Grossman, Paratimer: a progress indicator for MapReduce dags, in: Proceedings of the SIDMOD, ACM, 2010, pp. 507–518.
[17] A. Verma, L. Cherkasova, R.H. Campbell, Aria: automatic resource inference and allocation for MapReduce environments, in: Proceedings of the ICAC, ACM, 2011, pp. 235–244.
[18] J. Li, A.C. König, V. Narasayya, S. Chaudhuri, Robust estimation of resource consumption for SQL queries using statistical techniques 5 (11) (2012) 1555–1566.
[19] R.W. Wolff, Stochastic Modeling and the Theory of Queues, 14, Prentice Hall, Englewood Cliffs, NJ, 1989.
[20] Little's Law, https://en.wikipedia.org/wiki/Little.
[21] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, Elsevier, 2012.
[22] T.L. Adam, K.M. Chandy, J. Dickson, A comparison of list schedules for parallel processing systems, Commun. ACM 17 (12) (1974) 685–690.
[23] S. Keshav, An Engineering Approach to Computer Networking: ATM Networks, The Internet, and the Telephone Network, 11997, 1 edition, Addison-Wesley Professional, Reading MA, 1997.
[24] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, I. Stoica, PACMan: coordinated memory caching for parallel jobs, in: Proceedings of the USENIX NSDI, 2012.
[25] A. Verma, L. Cherkasova, V.S. Kumar, R.H. Campbell, Deadline-based workload management for MapReduce environments: pieces of the performance puzzle, in: Proceedings of the ProNetwork Operations and Management Symposium (NOMS), IEEE, 2012, pp. 900–905.
[26] W. Yu, Z. Liu, X. Ding, Semantics-aware prediction for analytic queries in MapReduce environment, in: Proceedings of the Forty-Seventh International Conference on Parallel Processing Companion, ACM, New York, NY, USA, 2018, pp. 27:1–27:9, doi:10.1145/3229710.3229713.
[27] J.K. Mullin, Estimating the size of a relational join, Inf. Syst. 18 (3) (1993) 189–196.
[28] C. Dellaquila, E. Lefons, F. Tangorra, Analytic-based estimation of query result sizes, in: Proceedings of the Fourth WSEAS International Conference on Artificial Intelligence, Knowledge Engineering Data Bases, WSEAS, 2005, p. 24.
[29] A. Ganapathi, H. Kuno, U. Dayal, J.L. Wiener, A. Fox, M. Jordan, D. Patterson, Predicting multiple metrics for queries: Better decisions enabled by machine learning, in: Proceedings of the ICDE, IEEE, 2009, pp. 592–603.
[30] A. Ganapathi, Y. Chen, A. Fox, R. Katz, D. Patterson, Statistics-driven workload modeling for the cloud, in: Proceedings of the ICDEW, IEEE, 2010.
[31] S.M. Johnson, Optimal two-and three-stage production schedules with setup times included, in: Naval Research Logistics Quarterly, 1, Wiley Online Library, 1954, pp. 61–68.
[32] T.C. Hu, Parallel sequencing and assembly line problems, Oper. Res. 9 (6) (1961) 841–848.
[33] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling, in: Proceedings of the Fifth European Conference on Computer Systems, ACM, 2010, pp. 265–278.
[34] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, A. Balmin, FLEX: a slot allocation scheduling optimizer for MapReduce workloads, in: Middleware 2010, Springer, 2010, pp. 1–20.
[35] T. Luo, R. Lee, M. Mesnier, F. Chen, X. Zhang, hStorage-DB: heterogeneity-aware data management to exploit the full capability of hybrid storage systems, in: Proceedings of the VLDB Endowment, 5, 2012, pp. 1076–1087.
[36] S. Shenker, I. Stoica, M. Zaharia, R. Xin, J. Rosen, M.J. Franklin, Shark: SQL and rich analytics at scale, in: Proceedings of the ACM SIGMOD, 2013.
[37] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P.K. Gunda, J. Currey, DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language., in: Proceedings of the OSDI, 8, 2008, pp. 1–14.
[38] Q. Ke, M. Isard, Y. Yu, Optimus: a dynamic rewriting framework for data-parallel execution plans, in: Proceedings of the Eighth ACM European Conference on Computer Systems, ACM, 2013, pp. 15–28.
[39] M. Islam, A.K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, A. Abdelnur, Oozie: towards a scalable workflow management system for Hadoop, in: Proceedings of the First ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, ACM, 2012, p. 4.
[40] Z. Zhang, L. Cherkasova, A. Verma, B.T. Loo, Automated profiling and resource management of pig programs for meeting service level objectives, in: Proceedings of the Ninth International Conference on Autonomic Computing, ACM, 2012, pp. 53–62.
[41] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, N. Koudas, MRShare: sharing across multiple queries in MapReduce, in: Proceedings of the VLDB Endowment, 3, 2010, pp. 494–505.
[42] X. Wang, C. Olston, A.D. Sarma, R. Burns, CoScan: cooperative scan sharing in the cloud, in: Proceedings of the Second ACM Symposium on Cloud Computing, ACM, 2011, p. 11.
[43] R.J. Stewart, Performance and Programmability Comparison of MapReduce Query Languages: Pig, Hive, Jaql & Java (Ph.D. thesis), Heriot Watt University, Edinburgh, United Kingdom, 2010.