Adaptive Information Passing For Early State Pruning in MapReduce Data Processing Workflows

Seokyong Hong, Padmashree Ravindra, and Kemafor Anyanwu
Department of Computer Science, North Carolina State University
{shong3, pravind2, kogan}@ncsu.edu

ABSTRACT

MapReduce data processing workflows often consist of multiple cycles where each cycle hosts the execution of some data processing operators e.g., join, defined in a program. A common situation is that many data items that are propagated along in a workflow, end up being "fruitless" i.e. they do not contribute to the final output. Given that the dominant costs associated with MapReduce processing (I/O, sorting and network transfer) are very sensitive to the size of intermediate states, such fruitless data items contribute unnecessarily to workflow costs. Consequently, it may be possible to improve the performance of MapReduce data processing workflows by eliminating fruitless data items as early as possible. Achieving this will require maintaining extra information about the state (output) of each operator, and then *passing* this information to descendant operators in the workflow. The descendant operators can use this state information to prune fruitless data items from their other inputs. However, this process is not without any overhead and in some cases, its costs may outweigh its benefits. Consequently, a technique for adaptively selecting *Information Passing* as part of an execution plan is needed. This adaptivity will need to be determined by a cost model that accounts for MapReduce's partitioned execution model as well as its restricted model of communication between operators. These nuances of MapReduce impose limitations on the applicability of information passing techniques developed for traditional database systems.

In this paper, we propose an approach for implementing *Adaptive Information Passing* for MapReduce platforms. Our proposal includes a benefit estimation model, and an approach for collecting data statistics needed for benefit estimation, which piggybacks on operator execution. Our approach has been integrated into Apache Hive and a comprehensive empirical evaluation is presented.

1. INTRODUCTION

A dominant trend for large scale data intensive processing is to use parallel processing over a cluster of commodity grade machines. The MapReduce [13] parallel processing model that was made popular a few years ago by Google has emerged as the de facto standard for processing data-intensive workloads. Data intensive tasks are captured in the MapReduce model as workflows made up of sequences of MapReduce cycles/jobs. Each MapReduce cycle consists of 2 phases - a *Map* and a *Reduce* phase. The Map phase executes the `map` function which takes a set of key-value pairs as input, and maps each pair to an intermediate key-value pair. Each phase can have multiple instances (mappers and reducers respectively) running concurrently on assigned data partitions i.e., partition parallelism. A popular open-source implementation of Google's MapReduce proposal is Apache Hadoop [1]. In order to implement a data processing task in this model, programmers have to figure out the best translation of their tasks into the MapReduce model. This process has been simplified with the introduction of extended MapReduce platforms such as Hive [22] and Pig [21], that provide high-level declarative languages with querying constructs ala SQL, and compilers for automatically compiling high-level programs into MapReduce execution workflows. The compilation process assigns query operators or constructs to specific MapReduce cycles. Fig.1 shows an abstract data processing MapReduce model which captures the structure of what each cycle with data processing operators looks like using Hive as an example. *Primary operators* (P_m, P_r) are operators that take input data from the Hadoop framework, process them, and feed them into other operators. Once the operators in each phase complete their processing, a *terminal operator* (T_m, T_r) collects the resulting output. Further, for non-trivial data processing tasks that require multiple MapReduce cycles, control and data dependencies are implied by the high-level program and represented as a workflow.

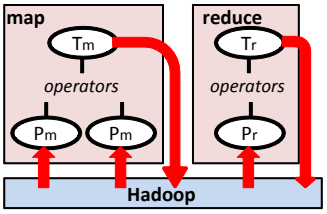


Figure 1: An Abstract MapReduce Job

An important thing to note about MapReduce data processing workflows is that they can be very costly. Table 1 shows the dominant costs (CPU costs of map and reduce function are ignored) in a MapReduce cycle. The scheduler schedules a set of slave nodes (*mappers*) to execute the Map phase, and assigns a split or partition of the input file to each mapper. The input data loading cost is represented as C_{Load} in Table 1. (Note that multiple splits can be assigned to a mapper, in which case multiple `map` function instances are executed on the node). Once all mappers are complete, the intermediate key-value pairs are materialized on the mappers' local disk (cost $C_{MapStore}$) in preparation for the Reduce phase. The scheduler then assigns nodes (*reducers*) to *reduce* a partition of map output values in a

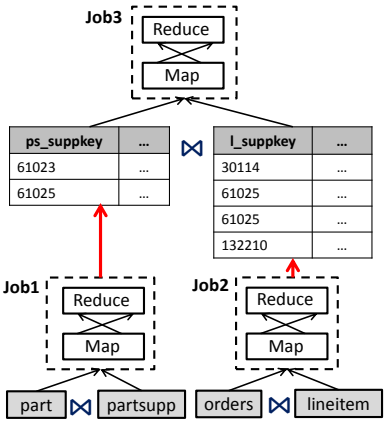


Figure 2: Unnecessary data movement of "fruitless" data items across a workflow

process that consists of three phases: *copy* — copying the sorted map output from mappers' disks to reducer nodes, resulting in a data transfer cost $C_{Shuffle}$; *merge* — merging the sorted output lists from different mappers based on the intermediate key with cost C_{Merge}, and *reduce* — `reduce()` is invoked once for each intermediate key, and applied to the associated group of values. The output key-value pairs generated by all reducer instances are merged and materialized into the underlying distributed file system e.g., Hadoop Distributed File System (HDFS) contributing to cost $C_{RedStore}$. The cost of a workflow is the aggregate cost of all cycles in the workflow. As can be observed, these costs are all sensitive to the size of data. Therefore, developing ways to keep the footprints of intermediate states small is crucial to performance of MapReduce workflows.

Table 1: Cost Factors in a MapReduce Job

Cost Factor	Description
C_{Load}	Input data loading cost
C_{Sort}	Map-side sorting cost
$C_{Shuffle}$	Data transfer cost
C_{Merge}	Reduce-side partition merging cost
$C_{MapStore}$	Map output data materialization cost
$C_{RedStore}$	Reduce output data materialization cost

1.1 The Problem

Due to the significant overhead associated with each MapReduce cycle, a key objective when optimizing MapReduce data processing workflows is minimizing the length of the workflow i.e., the number of MapReduce cycles in the workflow. Another very important objective (similar to relational databases) is minimizing the overall size of intermediate results or states produced during a data processing workflow (query). This is particularly crucial because of the multiple I/O, sorting, and network data transfer phases of intermediate results during a MapReduce data processing workflow. State-producing operators such as the `JOIN` operator, whose outputs may be larger than their inputs are an important consideration when thinking about minimizing the size of intermediate states produced. Indeed, in relational query optimization, the `JOIN` operator is a very fundamental operator for combining datasets, and its optimization is the focus of a lot of research. Typically, this is achieved by ordering operators in a way that minimizes inputs to each join. Cost models based on heuristics are used to estimate outputs of each operation so as to determine the best ordering of operations. In order to achieve this during compile-time, the parameters to such cost models will have to be precomputed, requiring preprocessing of data. This is natural for relational databases where data structures like indexes and statistical profiles of data are maintained as data is ingested into the system. However, MapReduce data processing platforms are typically not used to manage or host data in the long term but rather just for processing data-intensive workloads. Thus, features like statistics and indexing are immature or absent in MapReduce-based platforms like Hive and Pig. Further, any cost models used for relational optimization are not adequate for MapReduce data processing, because they do not capture key MapReduce-specific costs. Consequently, it is very important to consider runtime optimization techniques that can be applied as data is being ingested. Further, since these platforms are often used in batch processing mode, information gathered during earlier tasks in the batch workload can be used to inform the execution of latter tasks within the same batch workload.

A useful group of runtime optimization techniques in relational query optimization is called *Information Passing*(IP). Such techniques support passing information from earlier phases of data processing to later execution steps, particularly join operations, to help them prune their states more aggressively. If this is done early enough, we could avoid fruitless data from traveling along in the workflow only to be eventually pruned. As an example, Fig.2 shows an example job plan consisting of three MapReduce jobs executing equi-join (joining relations on equality condition on particular fields). The execution sequence is $Job1$, $Job2$, and $Job3$ (all joins are repartitioning joins). $Job3$ joins the two intermediate tables on *ps_suppkey* and *l_suppkey* after $Job1$ and $Job2$ complete. In this case, it is obvious that only records with suppkey 61025 will be joined and emitted to the final output (output of $Job3$) while remaining records will be discarded. The other records are essentially fruitless or irrelevant to final result. Unfortunately, these fruitless records affect C_{Load}, C_{Sort}, $C_{MapSort}$, $C_{Shuffle}$, and C_{Merge} costs for $Job3$. Further, they contribute to the costs of $Job1$ and $Job2$. If on the other hand, it is possible to pass information about the output context of $Job1$ to $Job3$, then it is possible to have $Job3$ prune its inputs (output of $Job2$ and $Job1$) while loading them, so that some savings in its C_{Sort}, $C_{MapStore}$, $C_{Shuffle}$, and C_{Merge} can be achieved. Better still, we may be able to pass this information to $Job2$ so that while it writes its output, it can prune records that are irrelevant to final results e.g., records 30114, 132210. In addition to savings in $Job3$'s costs, we can get additional savings with respect to $Job2$'s $C_{RedStore}$ and $Job3$'s C_{Load}. Such savings can be significant if pruning irrelevant tuples can be done much earlier in the workflow. For example, assume an execution plan such that these records are eventually pruned in $Jobk$ and not in $Job3$. Then, pruning these records from Job2's output will avoid carrying them along and processing them in some of jobs in $Job3$ to $Jobk-1$, before being eventually pruned out in $Jobk$.

In shared-memory or distributed environments where relational information passing techniques have been investigated, IP is usually achieved through some centralized shared memory structure which can be accessed by all operators. Shared nothing cluster environments and rigid communication models allowed by MapReduce make such an implementation strategy difficult. These issues will be elaborated in

Section 2.1. More significant is that the overhead of information passing may surpass its benefits. Therefore, techniques for runtime, adaptive selection of an information passing strategy as part of workflow execution need to be developed. Further, the decision for information passing selection has to be based on a cost model that is MapReduce-aware and informed by characteristics of data as determined during processing. The latter also suggests the need for a light-weight technique for collecting information about data at runtime by piggybacking on data processing.

1.2 Contributions

Specifically, we make the following contributions:

- We propose an architecture that supports adaptively enabling information passing (IP) in Hadoop-based data processing platforms based on its cost-benefits tradeoffs. The architecture provides support for *IP-aware* operators, IP-plan compilation and execution. We present a strategy for integrating these components into Apache Hive.
- We propose a MapReduce-based *benefit estimation* cost model for estimating the benefits of an IP-enabled execution plan. In addition, we present a light-weight data statistics collection technique that piggybacks on workflow execution, and collects necessary parameter values for the IP benefit estimation cost model.
- Finally, we present a comprehensive evaluation of the proposed framework using two datasets including a benchmark.

2. RELATED WORK

2.1 Sideways Information Passing

Relational database research has proposed a few variants of *Sideways Information Passing* (*SIP*) techniques [19, 17, 8, 9, 20] where information is passed between operators in an execution plan. *Magic-set rewriting* [19] ships summary information on examined values from a parent query to its subqueries or views so that they can discard unmatched records. *Semi-join* [9] processes join over remote sites. It projects and sends join columns from one site to another, and joins the projection with a remote relation. The resulting records are transferred to the original site and joined. In a sense, the task of passing information is integrated into the semantics of the operator itself rather than being an augmentation. In [8], an operator named *Eddy* routes records among relational operators based on dynamic runtime properties so as to maximize performance. [17, 20] produce filters from operators to other co-related operators to prune unnecessary records. Compile-time plans are augmented by run-time benefit estimation to reduce the information passing overhead. [17, 8, 20] introduced adaptive SIP approaches. Adaptivity can selectively add IP to an execution plan based on a cost model that relies on statistics of the underlying database system.

Discussion. While these SIP techniques have similar goals with our proposal, their implementations make assumptions that do not carry over to MapReduce execution platforms, making their adoption infeasible. Specifically, [19, 20] is designed for a centralized shared-memory environment where summaries can be exchanged using buffers or other message passing techniques [17, 8] that assume the existence of a centralized repository for exchanging tuples or summaries at will. On the other hand, MapReduce executes in a shared-nothing environment with a very restricted communication model between nodes. Further, operator execution in existing techniques is holistic i.e, one instance of operator processes all the input for that operator. This is in contrast to partitioned execution in MapReduce where multiple instances of an operation are executed, each processing one partition of the operator's input. The consequence is that information about the state of a MapReduce operator is fragmented across the different instances whereas in the traditional case, all summary information can be found in a single location. Finally, the adaptivity model proposed in [17] assumes the availability of data statistics. However, as was mentioned before, such features are lacking in existing MapReduce data processing platforms or require significant overhead [15] to compute. These assumptions by existing approaches make for a much simpler information passing problem than would be required in the MapReduce setting.

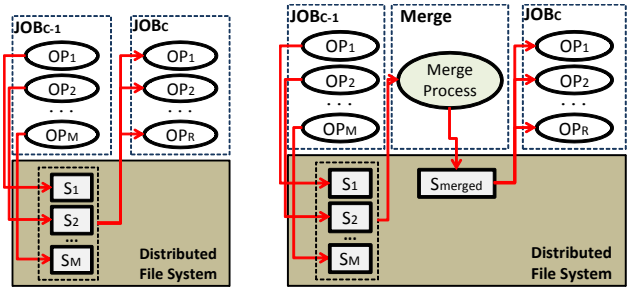


Figure 3: Summary Distribution

2.2 Information Passing Other Join Optimizations for MapReduce Platforms

Some recent efforts [10, 16] have focused on enabling information passing in MapReduce. A series of semi-join techniques in [10] resemble the traditional 2-way semi-join. However, each join operation is implemented using three MapReduce cycles which can be very expensive. In some earlier work [16], we proposed a basic information passing technique *Hadoop Information Passing (HIP)* that enables summary exchanges between multiple MapReduce jobs in a workflow. In this approach, fragmented summaries about the state of an operator are generated at the end of the cycle in which its executes. The summaries are stored into the Hadoop Distributed File System as compressed files of summary lists. If the total size of those summary fragments do not exceed a user-defined threshold, they are broadcast to nodes at the initiation of a subsequent job that takes as input, the earlier job's output. Other relevant work include efforts to reduce the length of a MapReduce workflow by clustering operations into few cycles as possible using multi-way join algorithms [18, 7, 23]. If the length of a workflow is reduced to just one cycle, then information passing becomes unnecessary. However, this is not always feasible for non-trivial data processing tasks because invariably, different operators will have conflicting key partitioning requirements forcing their execution to be assigned to different cycles. Also, some of the algorithms result in a large amount of replication which impedes performance. In the case of [23], a cost-based query optimization approach is proposed, where the multiplicity of

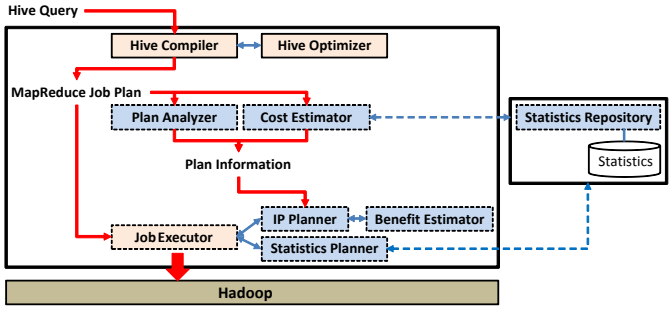


Figure 4: System Architecture (Solid-line: Hive component, Dotted-line: New or modified component)

replicated-join is decided by cost estimation. However, this approach requires additional MapReduce cycles per input table to calculate the necessary statistics.

3. ADAPTIVE INFORMATION PASSING IN HADOOP WORKFLOWS

In this section, we discuss our proposal for enabling information passing on Hadoop-based processing systems. Fig.4 shows our extended Hive framework with dotted lines denoting the new or extended components. Our system consists of three major components, (i) an *information passing framework* that enables IP-aware plans to generate and utilize summaries across jobs, (ii) a *benefit estimator* that estimates the costs for summary generation and propagation for each job, and (iii) a *statistics collector* that piggybacks on the MapReduce execution process to collect statistics required by the benefit estimator.

3.1 Information Passing Framework

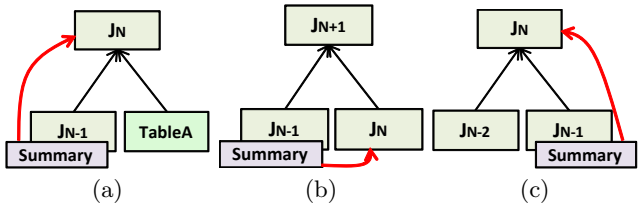
At compile-time, this framework decides whether information passing or statistics generation should be enabled by contacting several components. The default Hive job executor was modified to enable and execute IP-related decisions.

Compile-time preparation. A *Plan analyzer* analyzes a MapReduce job plan to produce *plan information*. Plan information contains job-specific information and relationships among the multiple jobs in a job plan. Information for each job is stored in a data structure called *JobDescriptor*, which maintains the job ID, a pointer to the Hive job descriptor, the type of reduce-side primitive operator P_R (e.g., join or group-by), paths for summary input and output, and so on. The relationships among jobs are represented in a *dataflow graph* and a *dependency graph*. A dataflow graph describes the input/output dataflow of jobs, and a dependency graph (reverse of the dataflow graph) describes their execution order. A *job executor* that submits each job in a MapReduce job plan to the Hadoop framework, references the plan information to make a decision on information passing. A *cost estimator* retrieves statistics about the job's input tables from the *statistics repository*, and calculates job costs. Job descriptors are augmented with such job cost information. Before submitting the current job J_C, the job executor invokes the *IP planner* to check whether the job should generate summary information (*SUMMARY_CREATION*) for any subsequent jobs, and/or load any summaries created by previous jobs (*SUMMARY_USAGE*).

Algorithm 1 corresponds to the decision-making process for summary generation, which consists of two steps. First, the IP planner probes the plan information and checks whether any subsequent job processing a join operation can leverage the summary information that J_C may produce (lines 3-4). For example, job J_{N-1} in Fig.5a is eligible for summary generation since job J_N can use the summary to prune unnecessary data while loading *TableA*. Next, the IP planner calls the *benefit estimator* to estimate the benefit that the summary can bring about (line 5). If there is considerable benefit (configurable via parameter $\beta > 0$), the planner makes a decision to enable *SUMMARY_CREATION* (line 6) so that J_C generates summary information during its execution.

Algorithm 1: Decision-making for State Creation

```
1 SUMMARY_CREATION ← false;
2 if IP is enabled in configuration file then
3     if J_C linked to any job J_N in dataflow graph then
4         if descriptor(J_N).ReducerType == JOIN then
5             if BenefitEstimator.estimate() > β then
6                 SUMMARY_CREATION ← true;
              end if
          end if
      end if
  end if
```



(a) (b) (c)

Figure 5: Possible IP Plans (a) Map-side pruning using *child-to-parent* IP, (b) Reduce-side pruning using *sibling-to-sibling* IP, (c) another example of *child-to-parent* IP using intermediate tables

The IP planner also examines J_C to determine usability of available summaries from a previously executed job J_P. Algorithm 2 shows the corresponding decision-making algorithm for summary utilization which considers two possible cases of information passing: *child-to-parent* and *sibling-to-sibling*. Fig.5 shows three job plans illustrating the two scenarios (the execution sequence is J_{N-2}, J_{N-1}, J_N, J_{N+1}, and current job $J_C = J_N$). First is the case of *child-to-parent* IP between a parent job J_C and a child job J_P (lines 3-7). If job J_C has a join operator in its reduce-phase (line 3), the planner looks up the dependency graph to find any jobs whose output is fed into J_C i.e., child jobs which have generated summaries that have not yet been used (lines 4-5). There are two possible cases of child-to-parent IP: a job may either join an intermediate table and a base table (Fig.5a), or it may join two intermediate tables (Fig.5c) such that the summary generated by the child was not used by its sibling. In both cases, the planner determines whether the size of the summary files P_i ($1 \leq i \leq r$, for r reducers) generated by the child job is less than a user-defined threshold α. This is done to avoid heap memory leakage issues caused by loading large summary information. If the summary size is less than the threshold, the planner invokes the benefit estimator to determine the benefit in summary usage. Based on the estimated benefit, the summary from the child job is used by J_C to prevent irrelevant data from being shuffled between

map and reduce phases. This map-side pruning helps to reduce C_{Sort}, $C_{Shuffle}$, and C_{Merge} costs in J_C. The second case is that of *sibling-to-sibling* IP where the summary information generated by a previously-executed sibling job J_S in the dataflow graph, can be used to prune the output of J_C (line 8). For example, J_N in Fig.5b has a sibling job (J_{N-1}) which generated a summary. J_N uses the summary to curtail C_{Store} costs in J_N, and C_{Load}, C_{Sort}, $C_{Shuffle}$, and C_{Merge} in J_{N+1}. Before the job executor submits each job to Hadoop, all decisions are encoded in the job so that corresponding operators generate and / or load summaries.

Algorithm 2: Decision-making for State Utilization

```
1  SUMMARY_USAGE ← false ;
2  if IP is enabled in configuration file then
       //Child-to-Parent IP
3      if descriptor(J_C).ReducerType == JOIN then
4          if J_C has a neighbor J_P in dependency graph then
5              if J_P generated summaries P_i (1 ≤ i ≤ r)
               such that Σ|P_i| ≤ α then
6                  if BenefitEstimator.estimate()> β then
7                      SUMMARY_USAGE ← true;
           //Sibling-to-Sibling IP
8          if J_C has a sibling job J_S in dataflow graph then
9              if J_S generated summaries P_i (1 ≤ i ≤ r) such
               that Σ|P_i| ≤ α then
10                 SUMMARY_USAGE ← true;
```

Run-time operation. In order to generate summary information and utilize it, operators in a job should be aware of decisions and operate accordingly. Hence, we designed *IP-aware* reduce-side T_r and map-side P_m operators. As described earlier, T_r is the final logical operator in the reduce phase that stores the reduce output to HDFS. If a job should generate summary information ($SUMMARY\_CREATION$ is true), decision-aware T_r operator generates bloom filters on the target column that is a join key in a subsequent job. Algorithm 3 describes the pseudo code for the IP-Aware *map* and *reduce* function skeletons. Note that the init (close) function calls each operator's init() (close()) method in the beginning (end) of each phase. At runtime, the IP-aware T_r operator calculates the hash value on the target column for each record, and puts the hash value in an in-memory buffer (lines 3-5). In the close() of the operator, the hash values are compacted using a bloom filter to minimize the size of the summary information (lines 8-10), and stored into the HDFS. Multiple instances of the T_r operator produce multiple partial summaries (one per reducer), and the job executor merges them into a single summary (merged bloom filter) as shown in Fig.3b.

The merged summary is broadcasted to computing nodes via *JobConf* before the subsequent job executes. JobConf is a data transport facility provided by the Hadoop framework to propagate system-wide and job-specific configurations to nodes. The JobConf is copied once to each node's local disk rather than to every mapper or reducer, and hence reduces the summary propagation costs. At the initialization phase of the subsequent job, the IP-aware map-side operator P_m loads the merged summaries into an in-memory buffer (lines 12-13). Whenever a record is read, the operator calculates the hash value for the target column (join key), probes the in-memory buffer, and prunes out irrelevant records (lines 15-17). Note that in the case of sibling-to-sibling IP, the summary loading and pruning happens in the reduce phase using the IP-aware T_r operator. If the T_r operator also needs to generate summary, the pruning precedes the summary generation phase. The next section describes the benefit estimation model that guides decisions in the information passing framework.

Algorithm 3: IP-Aware map()/reduce() Skeletons

```
   //Reduce() of current job J_n
   Reduce (key:grpKey, val:Corresponding list of tuples T)
   Init ();
   reduce ()
1  foreach tu ∈ T do
2      out_tup ← Normal reduce processing;
3      next_join_key ← extract join key for next job;
4      hashVal ← next_join_key.hash();
5      Add distinct hashVal to in-memory sorted set;
6      emit <null, out_tup>;
   Close ()
7  status ← close status of all reduce operators;
8  if status is Success then
9      summary ← bloom filter on hashVals;
10     Store summary to HDFS;

   //Merge summary from r reducers
11 mergedSummary ← merge(summary_1,...,summary_r);
   //Map() of a subsequent job J_{n+1}
   Map (key:null, val:Tuple tup from Input)
   Init ()
12 Load mergedSummary;
13 summarySet ← build in-memory sorted set from
   mergedSummary;
   map ()
14 join_key ← extract the join column from tup;
15 if tup ∈ baseRelation then
16     if join_key.hash() does not exist in summarySet
       then
17         Prune out tup;
18 out_tup ← Normal map processing;
19 emit <join_key, out_tup> ;
   Close ();
```

3.2 Benefit Estimation Model

Our benefit estimation model does not consider the case of sending summaries to sibling jobs (e.g., Fig.5b). Instead, it estimates the benefit from consuming summaries in parent jobs (e.g., Fig.5a and 5c). If a given summary is beneficial for a parent job, it is expected that a sibling job can achieve more benefit by using the summary in terms of C_{STORE} and C_{LOAD}. Hence, the proposed benefit estimation model aggregates partial benefits in C_{SORT}, $C_{SHUFFLE}$, and C_{MERGE} (B_{sort}, $B_{shuffle}$, and B_{merge} respectively), and differences the aggregated benefit and the cost for summary creation and propagation ($C_{summary}$):

$$B = B_{sort} + B_{shuffle} + B_{merge} - C_{summary} \quad (1)$$

Benefit in each step is estimated by subtracting the cost in the IP-enabled approach from the cost in the default approach. Hence, benefits in the three steps are:

$$B_{sort} = C_{sort-orig} - C_{sort-ip} \quad (2)$$

$$B_{shuffle} = C_{shuffle-orig} - C_{shuffle-ip} \quad (3)$$

$$B_{merge} = C_{merge-orig} - C_{merge-ip} \quad (4)$$

.

In sort-phase, MapReduce performs external merge sort

and the cost can be estimated as follows:

$$C_{sort-orig} = M(C_R + C_W)(\lceil \log_{B-1} \frac{M}{B} \rceil + 1) \quad (5)$$

when each mapper emits intermediate data of M pages on average, and the size of sorting buffer is B. C_R (C_W) is the unit cost to read (write) a page from (to) disk. Let Δ be the average size of unnecessary data that is expected to be pruned by a given summary information, then the cost in the IP approach is:

$$C_{sort-ip} = (M - \Delta)(C_R + C_W)(\lceil \log_{B-1} \frac{M - \Delta}{B} \rceil + 1) \quad (6)$$

.

When the intermediate data is shuffled into r reducers, each reducer receives, on average, $(M \times m)/r$ pages from m mappers. Hence, the estimated data shuffle cost of the original approach is:

$$C_{shuffle-orig} = \frac{M \times m}{r} C_T \quad (7)$$

where C_T is the unit cost to transfer a page. The estimated data shuffle cost in the IP approach is:

$$C_{shuffle-ip} = \frac{(M - \Delta)m}{r} C_T \quad (8)$$

In reduce-phase, each reducer receives partitions from all m mappers and merges them into one block. Hence, we estimate the merge cost in the default approach as equation 9, and that of the IP approach as equation 10.

$$C_{merge-orig} = R(C_R + C_W)(\lceil \log_{B-1} m \rceil) \quad (9)$$

$$C_{merge-ip} = (R - \Delta \cdot \frac{m}{r})(C_R + C_W)(\lceil \log_{B-1} m \rceil) \quad (10)$$

where R is the average input size per reducer. Since $M \cdot m = R \cdot r$, we substitute $(M \cdot m)/r$ for R producing the following equations:

$$C_{merge-orig} = M \cdot \frac{m}{r}(C_R + C_W)(\lceil \log_{B-1} m \rceil) \quad (11)$$

$$C_{merge-ip} = \frac{m}{r}(M - \Delta)(C_R + C_W)(\lceil \log_{B-1} m \rceil) \quad (12)$$

Cost $C_{summary}$ is caused by summary generation and propagation and consists of four costs:

$$C_{summary} = C_{ip-store} + C_{ip-merge} + C_{ip-copy} + C_{ip-load} \quad (13)$$

Each T_r operator stores a partial summary information in a bloom filter of size $|F|$ into HDFS. Hence, the cost to store a partial summary is:

$$C_{ip-store} = (U_T + U_W)|F| \quad (14)$$

where U_R, U_W, and U_T are unit costs to read, write, and transfer a single byte, respectively. Since partial summaries generated by r' reducers are combined into one bloom filter, the cost to merge partial summaries is:

$$C_{ip-merge} = (U_T + U_R)|F|r' + (U_T + U_W)|F| \quad (15)$$

The cost to copy a merged summary to nodes is:

$$C_{ip-copy} = N(U_R + U_W + U_T)|F| \quad (16)$$

The summary loading cost of each operator from local disk is:

$$C_{ip-load} = U_R|F| \quad (17)$$

Parameters such as C_R, C_W, C_T, U_R, U_W, and U_T are machine-specific and can be earned by performance measurement. B and N can be calculated with Hadoop configuration parameters, and $|F|$ is set in the Hive configuration file. At the time that benefit estimation is performed, the number of reducers (r') in a current job has already been set by the Hive compiler. However, the number of mappers (m) and reducers (r) in its parent job are unknown since they depend on the input table sizes of the parent job and it is complicated to estimate those parameters without input statistics. For the same reason, the estimation of M and Δ is complicated. In the next section, we describe our statistics collection approach that piggybacks on MapReduce job execution.

3.3 Piggybacking Statistics Collection on Operator Execution

Our approach for statistics collection piggybacks on the execution of MapReduce jobs. Generated statistics are used by the cost estimator to estimate statistics on intermediate data in a MapReduce job plan. These statistics are used by the benefit estimator to calculate necessary parameters such as m, r, M, and Δ, which are supplied to the benefit estimation model to make decisions about information passing. Required statistics include the size of tables, the number of records, the average sizes of columns, and the value distributions within a column. The general idea is that rather than requiring a separate pre-processing step for computing statistics, we choose to exploit the fact there is a good likelihood that users will want to execute multiple queries on their datasets. Therefore, for each query, we compute and store statistics on selected columns of the tables being processed. Then, for future queries that refer those columns, we use the already computed statistics. Note that the statistics planner does not estimate statistics on all columns in input tables. For example, while evaluating TPC-H queries, columns for comments (e.g., *ps_comment*, *l_comment*, *c_comment*, and *p_comment*) and auxiliary information (e.g., *c_address*, *s_address*, and *s_phone*) are rarely positioned in filter or join conditions, and hence are not important for cost estimation. Such columns can be excluded to reduce the overhead of statistics generation and transmission costs. Users can manually specify additional columns into the column set by updating the Hive configuration file.

At compile-time, the *statistics planner* creates a list of input tables from a MapReduce job plan, and extracts a set of meaningful columns whose statistics are required for cost estimation. The statistics planner probes the statistics repository using Java Remote Method Invocation (Java RMI) to check the availability of statistics related to the listed tables and columns. If the statistics are unavailable, the statistics planner enables statistics collection and embeds the decision into the job plan. Additionally, the statistics planner registers the input table sizes in the statistics repository if such information is missing.

At run-time, the map-side primary operators (P_m) in each job generate the statistics on the required columns. Once the operators finish execution, the generated statistics are registered with the statistics repository. The generated statistics are partial since the P_m operators process records from different input portions. In this environment, it is straightforward to calculate the number of records and the average sizes of columns from fragmented statistics. However, re-

constructing the value distributions from fragmented statistics is quite tricky. Sampling [11, 12] or histogram [23] can be considered but such approaches may incur high inaccuracy or huge data transmission overhead. In order to combat this issue, we used a recent logarithmic counting method called as *HyperLogLog* [14]. This algorithm supports set union operation which enables combination of multiple statistics fragments in a natural way. Therefore, partial value-distribution from multiple input partitions can be merged efficiently. Next, the memory footprint that the algorithm requires to store bit-vectors is relatively small ($O(\log\log D)$) with high accuracy. For example, the algorithm consumes 1.5KB memory with a 2% error rate for 10^9 cardinality values [14]. When a job completes its execution, the statistics repository unions partial bit-vectors that have been registered.

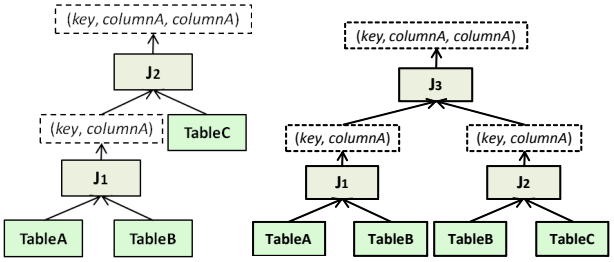


Figure 6: (a) Query1, (b) Query2

4. EXPERIMENTAL EVALUATION

The information passing approach has been implemented in Apache Hive 0.5 [2] on top of Apache Hadoop 0.20.0 [3]. This section presents an extensive study of the proposed approach by comparisons with original Hive, HIP [16], and semi-join [10]. For HIP, the threshold of total summary size was set to 512KB and 4MB. For the proposed IP approach, two sizes of bloom-filter were used for storing summary information (512KB and 4MB). First two steps of semi-join were implemented using vanilla MapReduce applications. However, the last step was implemented as a repartitioning join, instead of Hive's fragment-replication join named *map-side join*. This is because the map-side join failed for experiments in which the size of intermediate data from the first two steps was not small enough to fit into memory.

4.1 Experiment Setup

Experiments were conducted on a cluster on *NCSU VCL* [6] which consists of 21 blade servers (one master node and 20 slave nodes). Each node has a 3.0GHz dual-core Xeon processor, 4GB memory, and a 28GB SCSI disk, and runs Red Hat Enterprise Linux 5. Hadoop framework was configured with 512MB of block size, replication factor 1, no speculative execution, and 1024MB of heap size for mappers and reducers.

4.2 Workloads and Analysis

Two kinds of synthetic datasets, including one benchmark were used. The first synthetic dataset is generated based on user-supplied parameters such as number of records, column sizes, and range of join column values. This benchmark was chosen since it is complicated to manually set reference ratios among generated tables with existing benchmarks. Three tables were generated using this benchmark: *TableA*, *TableB*, and *TableC*. Each table is 20GB and includes three columns (*key*: 25B, *columnA*: 75B, *columnB*: 100B). The *key* column in each table was used as a join key. The join key density of *TableB* and *TableC* is fixed to one while the join key density in *TableA* varies across experiments. Hence, records in *TableA* match different numbers of records in other tables. Records in *TableB* and *TableC* are exactly same. In order to evaluate the effects of passing summary information to different MapReduce jobs, two benchmark queries were used as shown in Fig.6. The query in Fig.6(a) compiles into two MapReduce jobs. The first job joins *TableA* and *TableB*, producing intermediate records of (*TableB.key*, *TableA.columnA*). The second job joins the intermediate output from the first job with *TableC*, and generates (*TableB.key*, *TableA.columnA*, *TableC.columnA*). For HIP and IP, the execution time of *Job2* is summed up with any delay in execution time of *Job1*. Semi-join was computed on the intermediate records from *Job1* and *TableC*, and the execution times of three steps were summed up. Next, the query in Fig.6(b) is translated into three MapReduce jobs. The first one is similar to the first job in the previous query. The second job joins *TableB* and *TableC*, and produces same number of records. The intermediate records are of (*TableB.key* and *TableC.columnA*). The last job joins the two intermediate tables and generates records of the form (*TableB.key*, *TableA.columnA*, *TableC.columnA*). Execution times for *Job2* and *Job3* for different approaches were also compared. However, semi-join was not performed for this query since the first two steps of semi-join cannot remove any records deriving no benefit. Second set of experiments used the TPC-H benchmark [5] dataset (40GB). The Hive version of TPC-H queries [4], which are written in HiveQL were used. Among the TPC-H benchmark queries, a set of relevant queries with multiple join operations were chosen. Different query plans were evaluated.

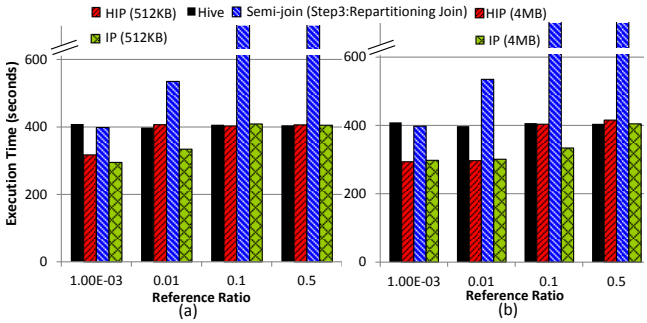


Figure 7: Execution times for Query1:Job2 with varying reference ratios (a) HIP: 512KB threshold, IP: 512KB bloom-filter, (b) HIP: 4MB threshold, IP: 4MB bloom-filter

4.3 Experimental Results

Performance improvement with varying reference ratios. The IP approach enhances query processing performance by pruning unnecessary records before being joins. Hence, the effectiveness of the approach depends on the reference ratio between joined tables. To check the relationship between performance and reference ratio, *TableA*'s reference ratio was varied such that its records match different numbers of records in the other tables. Fig.7 shows

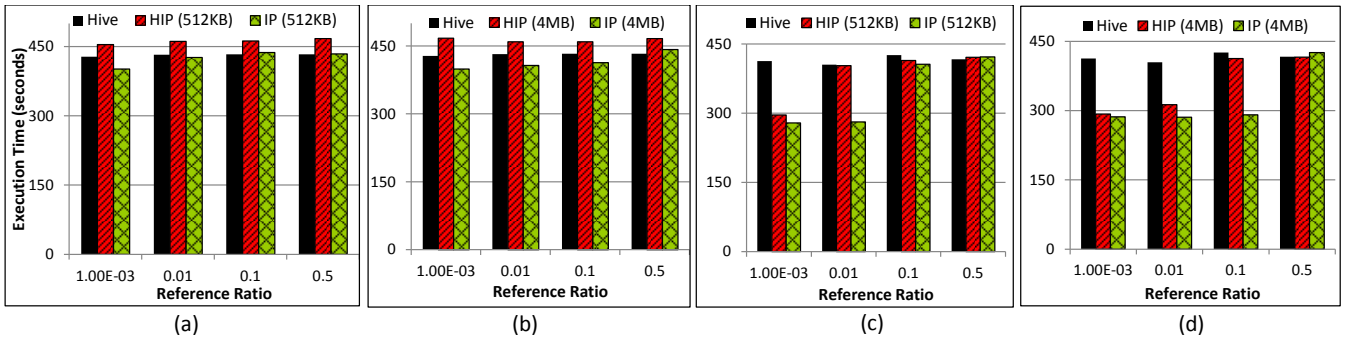


Figure 8: Query2 execution times with varying reference ratios, (a) Job2(HIP: 512KB threshold, IP: 512KB bloom-filter), (b) Job2(HIP: 4MB threshold, IP: 4MB bloom-filter), (c) Job3(HIP: 512KB threshold, IP: 512KB bloom-filter, (d) Job3(HIP: 4MB threshold, IP: 4MB bloom-filter))

the execution times of the query in Fig.6(a). First, semi-join failed for reference ratio 0.1 and 0.5 (denoted by broken lines on the y-axis to indicate that execution failed after a lengthy execution period) since the size of join key list exceeded available heap memory in its first step. For reference ratio 0.001 and 0.01, the semi-join approach did not show better performance than the original Hive approach. On the other hand, HIP and IP showed performance improvements as the reference ratio decreases. However, the IP approach was beneficial with relatively higher reference ratio than HIP (reference ratio = 0.01 in Fig.7(a) and reference ratio = 0.1 in Fig.7(b)). A problem with HIP approach is that it cannot calculate the size of summary information before partial summary fragments are merged and loaded into memory. Hence, the total size of summary fragments is compared with the user-defined threshold. As a consequence, even if the size of merged summary information is less than the threshold, it may disallow generated summary information to be loaded by jobs. On the other hand, IP approach decides about summary generation and utilization based on benefit estimation, and can maximize the benefit of information passing. In Fig.7(b), for example, HIP disabled the use of summary for reference ratio 0.1, while IP allowed summary to be used by *Job2* deriving benefit.

When HIP processes *Query2* in Fig.6(b), *Job1* and *Job2* generate summaries, and *Job3* utilizes those summaries. Hence, as shown in Fig.8(a-b), *Job2* execution times in HIP are longer than the original Hive approach due to summary generation overhead. On the other hand, the IP approach transports summary information from *Job1* to *Job2*, and prunes unnecessary records at the end of *Job2*. Hence, it could derive benefits in materialization steps when its benefit estimation enables summary information (e.g., when reference ratio = 0.001 and 0.01 in Fig.8(a) and reference ratio = 0.001, 0.01, and 0.1 in Fig.8(b)). The benefits were less than 8% performance improvement. In case of *Job3*, IP achieved more performance improvement than HIP since reduced intermediate data from *Job2* derived benefits in data loading phase in addition to shuffle-phase as shown in Fig.8(c-d).

Performance improvement with varying block sizes. This experiment evaluates the impact of varying HDFS block size on the different approaches. The reference ratio was fixed to 0.01, and HIP threshold and size of bloom-filter in IP were set to 4MB. Fig.9(a) shows the performance improvement rates of *Job2* in *Query1* relative to the original Hive approach. Semi-join was worse than Hive in all settings while the performance degradation rate decreased as block size increased since the execution time of semi-join increased slower than that of Hive. Both HIP and IP showed linear speedups with increasing block sizes. Each mapper that loads a block from *TableC* is assigned more data as the block size increases. Hence, given the summary information produced in the previous job, both approaches are able to prune out larger amount of "fruitless" data items before being shuffled, thus deriving higher cost saving in shuffle-phase.

Fig.9(b) shows the performance improvement rates of HIP and IP for processing *Job2* in *Query2*. In HIP, summary information is always generated by children jobs (e.g., *Job1* and *Job2* in Fig.6(b)), and transported to parent jobs (e.g., *Job3* in Fig.6(b)) in a dataflow graph. Hence, in the case of HIP, *Job2* is accompanied with an overhead to generate summary information causing less than 1% slowdown. On the other hand, the IP approach can prune unnecessary data by using summary information transported from a sibling job (e.g., *Job1* in Fig.6(b)) in a dataflow graph. Hence, IP derives benefit in the output materialization step of *Job2* as shown in Fig.9(b). However, the benefit is relatively small (less than 1% speedups) for all block sizes.

Fig.9(c) compares the performance improvement rates of HIP and IP for *Job3* in *Query2*. First of all, the IP approach outperforms HIP for all block sizes. HIP showed more than 20% performance improvement by preventing unnecessary data before data shuffle-phase. On the other hand, IP achieved relatively better performance enhancements than HIP since already reduced intermediate data could reduce the cost in data loading phase in addition to the data shuffle cost. In the second place, with 512MB to 1024MB block sizes, both approaches showed almost constant performance improvements. Hive and HIP scheduled same numbers of mappers (86 mappers) and reducers (26 reducers) with those block sizes where the amounts of input data to each mapper and each reducer were not changed even if the block size increased. Hence, the effect of summary information in shuffle-phase was almost same with those block sizes. In case of IP, the same numbers of mappers (86 mappers) and reducers (14 reducers) were scheduled from 512MB to 1024MB block sizes. Therefore, the effect of reduced input data was almost same in data-loading and shuffling steps with different block sizes.

Performance measurements with TPC-H benchmark. Experiments with TPC-H benchmark were performed to measure the effectiveness of the IP approach. In these experiments, the bloom-filter size of IP and the

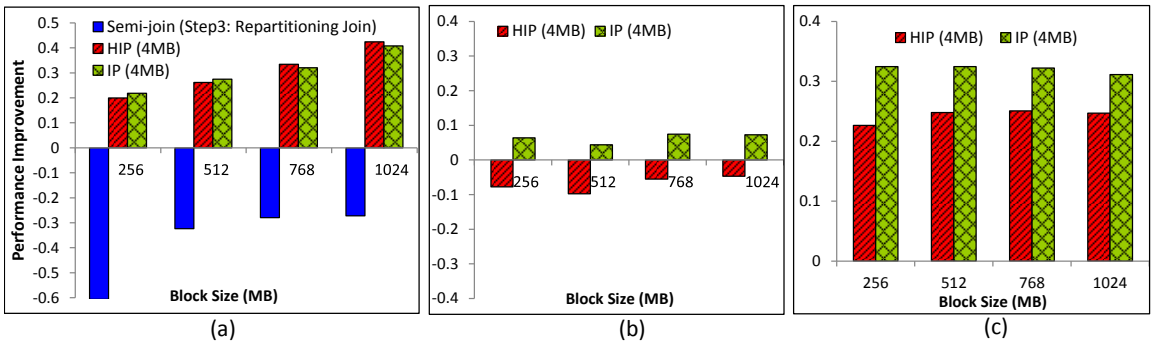


Figure 9: Performance improvement with varying block sizes (a) Query1:Job2 (b) Query2:Job2 (c) Query2:Job3

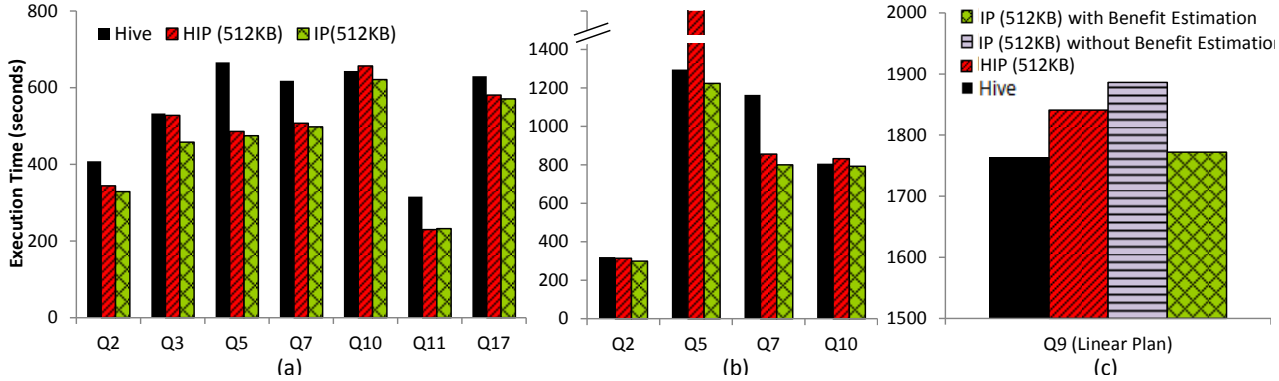


Figure 10: TPC-H queries with (a)linear and (b) non-linear plans, (c) Benefit estimation (TPC-H Q9 linear plan)

summary size limitation of HIP were configured to 512KB. Fig.10(a) shows the execution times of the different approaches for TPC-H queries which are translated into linear plans. HIP and IP approaches improve the query processing performance for most of the queries up to 27.0% and 28.6%, respectively. However, HIP does not improve the performance of *Q3*, and degraded the performance of *Q10* a little due to overhead of generating and transporting summaries. With non-linear plans as shown in Fig.10(b), the IP approach showed 5-6% performance improvements when processing Q2, Q5, and Q10, and improved the execution time of Q7 by about 31%. In HIP, the execution times of Q2 and Q7 were improved 2% and 26%, respectively while that of Q10 was degraded about 3% due to the overhead of summary generation. It is notable that HIP did not work for Q5 because one of the jobs in the plan did not have enough heap memory space for storing its output summary information. HIP stores a list of hash values on a join column in memory. Hence, if the size of the in-memory list exceeds available memory space, it drives reducers to fail their execution. On the other hand, IP stores such data in a more compact bloom filter whose size is configurable, thus avoiding such problems.

Benefit Estimation. HIP and IP approaches without benefit estimation, may worsen query processing performance for cases where the overhead of generating and transporting summary information exceeds the benefit achieved by using the summary information. For example, with TPC-H Q9, both approaches showed worse performance when they were enabled. Fig.10(c) shows the execution times of a Q9 linear plan in different approaches. We compared the original Hive, HIP, IP without benefit estimation, and IP with benefit estimation. In HIP, jobs always generate summary information as long as they have subsequent jobs to which such summary information can be transferred. In addition, it transports summary information to the next job if its size is less than a user-defined threshold, even if the summary may not be beneficial. Because of such summary generation and transmission overheads, HIP brought about a 80 second delay in the execution time of the Q9 linear plan. In IP without benefit estimation, all jobs in the query plan always generate summaries and transport them to next jobs. As a result, such imprudent use of summary information added a cost of about 125 seconds to the original execution time. On the other hand, the IP approach which leverages benefit estimation, could selectively allow summary generations and transmissions based on cost estimations. This allowed for maximizing the benefit of IP and preventing any large performance degradation in worst cases.

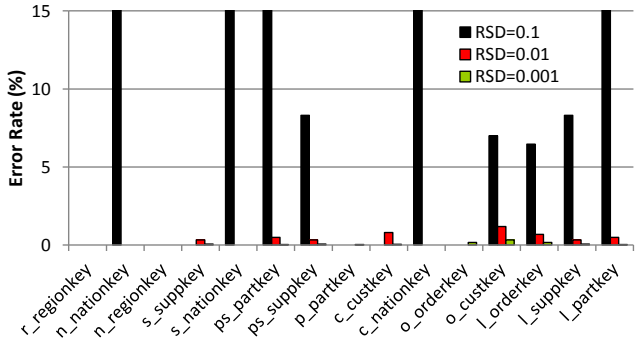


Figure 11: Accuracy for the number of distinct values with different RSDs

Piggyback Statistics Collection. Statistics collection which piggybacks query processing may impose penalties on its performance. This section evaluates the effect of piggy-

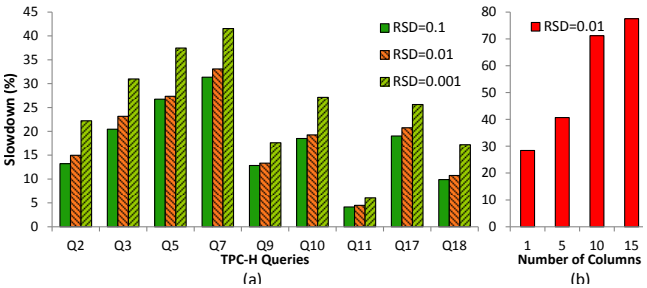


Figure 12: Piggyback statistics collection overhead (a) varying RSD values (b) varying number of columns

back statistics collection. While processing TPC-H queries, distinct value cardinalities were generated on columns that were either join keys or were involved in filter conditions. The slowdown on execution time was then measured. The slowdown for each query was calculated as following:

$$slowdown(\%) = \frac{T_P - T_O}{T_O} \times 100$$

where T_P is the execution time of a query with piggyback statistics collection, and T_O is the execution time of the original Hive approach. During the experiments, three relative standard deviations (RSDs) (0.1, 0.01, and 0.001) were used as a parameter to HyperLogLog. This was done to change the size of required memory space. As the RSD value increases, HyperLogLog requires less memory space while causing higher error rate. Fig.12(a) shows the slowdowns of the TPC-H queries. As RSD became smaller, the slowdown of query processing performance increased. When RSD = 0.001, slowdowns were between 6% and 41% while error rates in distinct value cardinalities were less than 0.4%. RSD = 0.01 caused 4-33% slowdowns with less than 1.6% error rates. With RSD = 0.1, slowdowns were between 4% and 31% with error rates less than 30.7%. Fig.11 shows error rates in distinct value cardinality estimation on a set of key columns. In the experiment, processing overhead to generate statistics was the main factor that affected the slowdowns of query execution times. For TPC-H Q7, the processing overhead for statistics generation affected about 97% of the slowdowns when RSD is 0.1 and 0.01, while registering partial statistics caused 3% of the slowdowns. However, when RSD = 0.001, the overhead to register partial statistics to the statistics repository caused about 19% of the slowdown. As a result, as the RSD value increases, partial statistics registration overhead can be one of the dominant factors that affect query performance.

Next, slowdowns were measured by manually changing the number of columns on which distinct value cardinalities were generated. This experiment used a query which loads *lineitem* table and performs a groupby operation on *l_linenumber*. RSD was fixed at 0.01. As shown in Fig.12, the increase in number of columns involving distinct value cardinality collection, resulted in more processing and statistics transmission overheads, thus increasing slowdown in query processing time. Hence, choosing minimal columns from which distinct value cardinalities are collected is necessary to decrease the performance degradation of piggyback statistics collection.

5. CONCLUSIONS

We present an adaptive information passing approach for early pruning of intermediate states in a MapReduce data processing workflow. The approach is based on a MapReduce-aware cost model for estimating potential benefits or loss of information passing in a particular workflow. We also present a light-weight approach for computing statistics by piggybacking on operator execution. Our approach for integrating the proposed information passing technique into a popular platform, Apache Hive, is presented. A comprehensive empirical evaluation using two datasets shows the benefits of our approach over existing techniques.

6. ACKNOWLEDGMENTS

This work was partially funded by NSF grant IIS-0915865 and IIS-1218277.

7. REFERENCES

[1] Apache hadoop. http://hadoop.apache.org.
[2] Apache hive. http://hive.apache.org.
[3] Hadoop 0.20 documentation. http://hadoop.apache.org/common/docs/r0.20.0/.
[4] Running the tpc-h benchmark on hive. https://issues.apache.org/jira/secure/attachment/12416257/TPC-H_on_Hive_2009-08-11.pdf.
[5] Tpc-h. http://www.tpc.org/tpch/.
[6] Virtual computing lab. http://vcl.ncsu.edu.
[7] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.
[8] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, 2000.
[9] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.
[10] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD Conference*, pages 975–986, 2010.
[11] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.
[12] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *SIGMOD Conference*, pages 287–298, 2004.
[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
[14] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA*, pages 127–146, 2007.
[15] R. Grover and M. J. Carey. Extending map-reduce for efficient predicate-based sampling. In *ICDE*, pages 486–497, 2012.
[16] S. Hong and K. Anyanwu. Hip: Information passing for optimizing join-intensive data processing workloads on hadoop. In *DEXA (2)*, pages 384–391, 2012.
[17] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE*, pages 774–783, 2008.
[18] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, pages 25–36, 2011.
[19] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In *SIGMOD Conference*, pages 103–114, 1994.
[20] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD Conference*, pages 627–640, 2009.
[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[23] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 12:1–12:13, 2011.