1-1-2012

Processing Data-Intensive Workflows in the Cloud

Zhuoyao Zhang
University of Pennsylvania, zhuoyao@seas.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-12-08.

Processing Data-Intensive Workflows in the Cloud

Abstract

In the recent years, large-scale data analysis has become critical to the success of modern enterprise. Meanwhile, with the emergence of cloud computing, companies are attracted to move their data analytics tasks to the cloud due to its exible, on demand resources usage and pay-as-you-go pricing model. MapReduce has been widely recognized as an important tool for performing large-scale data analysis in the cloud. It provides a simple and fault-tolerance framework for users to process data-intensive analytics tasks in parallel across dierent physical machines. In this report, we survey alternative implementations of MapReduce, contrasting batched-oriented and pipelined execution models and study how these models impact response times, completion time and robustness. Next, we present three optimization strategies for MapReduce-style work- ows, including (1) scan sharing across MapReduce programs, (2) work- ow optimizations aimed at reducing intermediate data, and (3) schedul- ing policies that map work ow tasks to dierent machines in order to minimize completion times and monetary costs. We conclude with a brief comparison across these optimization strate- gies, and discuss their pros/cons as well as performance implications of using more than one optimization strategy at a time.University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-12-07.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-12-08.

Processing Data-intensive Workflows in the Cloud

WPE-II Written Report

Zhuoyao Zhang

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
zhuoyao@seas.upenn.edu

April 19, 2012

Abstract

In the recent years, large-scale data analysis has become critical to the success of modern enterprise. Meanwhile, with the emergence of cloud computing, companies are attracted to move their data analytics tasks to the cloud due to its flexible, on demand resources usage and pay-as-you-go pricing model. MapReduce has been widely recognized as an important tool for performing large-scale data analysis in the cloud. It provides a simple and fault-tolerance framework for users to process data-intensive analytics tasks in parallel across different physical machines.

In this report, we survey alternative implementations of MapReduce, contrasting batched-oriented and pipelined execution models and study how these models impact response times, completion time and robustness. Next, we present three optimization strategies for MapReduce-style workflows, including (1) scan sharing across MapReduce programs, (2) workflow optimizations aimed at reducing intermediate data, and (3) scheduling policies that map workflow tasks to different machines in order to minimize completion times and monetary costs.

We conclude with a brief comparison across these optimization strategies, and discuss their pros/cons as well as performance implications of using more than one optimization strategy at a time.

Contents

1 Introduction

In the recent years, data-intensive analytic applications have become core to the functions of modern enterprise. Large companies like Google, facebook, and LinkedIn are processing and analyzing large amount data everyday. These data analytic tasks range from business intelligent analytics to social network connection analysis, to more advanced scientific data analysis and machine learning applications and the amount of data produced daily is exploding. For example, the data processed by Facebook everyday grows from 5-6TB to 10-15TB in 6 months [4].

Meanwhile, with the advance of cloud computing, companies are attracted to move their data analytics tasks to the cloud platform due to its large amount of resources which can be required and returned on demand and its flexible pay-as-you-go pricing model. Nowadays, there are several large companies providing cloud infrastructure services including Google, Amazon and Microsoft. Hundreds of companies are using these cloud infrastructures to provide service and perform data analytic tasks like Dropbox, IMDB, Yelp etc., with more and more companies moving towards it.

In such background, MapReduce has been widely recognized as an important tool for large-scale data analysis in the context of cloud computing. It provides a simple and fault-tolerance framework for users to process data-intensive analytic tasks with the distributed resources in the cloud. There are also platforms based on MapReduce such as Pig [7] and Hive [16] which provide even high-level abstraction on top of the MapReduce engines, so that user can specify complex data-analysis tasks easily with declarative SQL-like language such as Pig-Latin [14] and Hive-SQL. The underlying runtime will compile the user-specified declarative programs into MapReduce workflows (represented as a directed acyclic graph (DAG)) which will then be launched in the clouds.

Providing good performance for processing data analytic workflows is an important issue for these MapReduce based platforms, and the system performance could directly map to the monetary cost in the cloud environment since the cloud resources are charged based on a pay-as-you-go mode. However, the existing MapReduce based data processing frameworks fall short of efficient performance optimization strategies. Until now, optimizations are usually performed manually by users which is difficult and inefficient.

To address this problem, there are great amount of works focusing on the performance optimization for processing workflows on MapReduce based framework in the cloud. These works can be generally classified into two categories. The first category includes optimizations for workflow specification, e.g. optimizing the execution plan of different operators within a workflow to minimize the potential I/O costs. The second category include optimizations for workflow execution, e.g. determining the mapping between operators and the physical machines when launching the workflow in the cloud.

In this report, we will first present alternative implementations for the MapReduce framework, contrasting batched-oriented and pipelined execution models and study how these models impact response times, completion time and robustness. Next, we present three optimization strategies for MapReduce-style workflows, including (1) scan sharing across MapReduce programs, (2) workflow optimizations aimed at reducing intermediate data, and (3) scheduling policies that map workflow tasks to different machines in order to minimize

completion times and monetary costs. Besides the ones described in this report, there are also other optimization strategies such as more efficient operator implementation [10], better workload balancing for parallel data processing [13], intermediate data storage optimization [11] scheduling policy to handle stragglers [20], incremental computation [3] and system configuration turnings [8]). The goal of this report is not to describe all the possible optimization strategies but to show potential optimization opportunities in processing data analytic workflows in the cloud.

The rest of this report is organized as follows. We start with a detail problem statement in section 2 by defining the performance metrics and identifying the factors affecting these metrics. Section 3 introduces the MapReduce architecture with two alternative implementations. We describe scan sharing techniques in section 4.1, workflow optimization in section 4.2 and scheduling policy in section 5. Finally, section 6 reviews all the strategies and discuss the challenges.

2 Problem Statement

To scope our survey paper, we begin first with a problem statement that provides a concise definition of the performance metrics we will focus for data-intensive workflows in the cloud and the factors that affect these metrics.

2.1 Overview of Data Processing Workflow on The Cloud

The data processing workflow is often used to represent the execution of complex data-intensive analytics. It typically consists of a sequence of inter dependent data processing operators e.g. data loading, information filtering, data transformation & analysis and other user-defined data manipulation. Such workflow can be represented as a directed acyclic graphs (DAG) where the operators are represented as nodes and the data dependency between the operators are represented as edges in the graph. The left part in Figure 1 gives an example of such DAG, where the cylinder represent the input/output data and the circles represent the operators. There could be hierarchical structure of the data operators as well, for example, in a MapReduce based workflow, each operator represents a MapReduce job, while each job could contain a number map and reduce tasks.

To execute the workflow, the system will assign each data operator to a physical computing node as shown in the right part in Figure 1. In the cloud environment, the computing resource is provided as an instance that capsulates a fixed amount of resource including the CPU capability, the memory and disk storage and the network bandwidth. Different kind of instances contains different amount of resources. For example, the Amazon EC2 provides 12 different instances from micro instance with the minimal capacity to the extreme large instance with the most powerful capability. Cloud users can claim and return these instances on demand and will be charged for the use of the instance based on a per quantum pricing scheme, e.g, 1 hour. The unit pricing for each instance depends on its resource capacity. Take EC2 for example, the pricing ranges from $0.03 per hour for micro instance to $2.5 for extreme large instance.

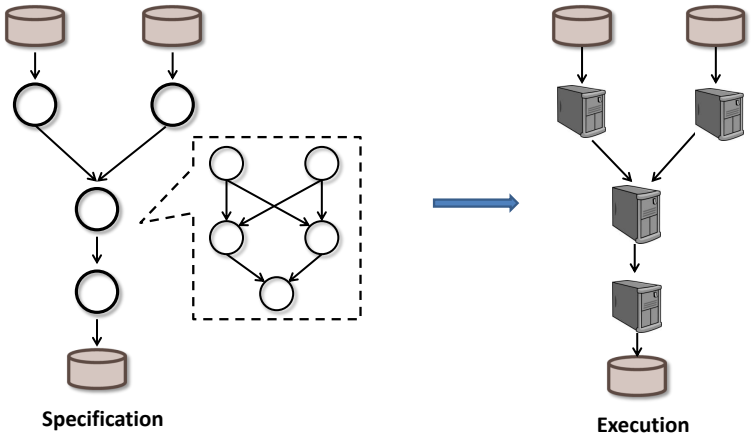

Figure 1: Example of Data Processing Workflow

2.2 Performance Metrics

Depending on the requirements of different applications, there are different performance metrics in optimizing the MapReduce framework such as completion time, monetary costs, throughputs, responsibility, accuracy, throughputs, reliability, fairness etc. There are tradeoffs between these metrics. For example, interactive applications prefer quick response and can sacrifice accuracy to some extend, batch applications are not very time sensitive and would like to minimize the monetary cost as long as it can finish within certain time period. In this report, we are particularly interested in the following performance metrics.

- **Completion time:** Completion time is the time elapses from the start of the workflows to the time when all the results are generated. It is the most important cost metric that represents the total time it takes to complete a workflow.
- **Monetary cost:** The monetary cost to complete the entire workflow based on cloud platforms (e.g, Amazon EC2) is also an important metric as the resources are claimed on demand and will be charged as long as you are using it. The monetary cost is closely related to the completion time. However, they not always correlated due to the pricing scheme in the cloud.
- **Response time:** Response time here represents the time elapses from the start of the workflows to the time when the system response. It is different from completion time as the response may contain only partial results. Short response time is especially important for interactive applications since the clients prefer short response time and are usually tolerant with incomplete results.

2.3 Factors Affecting Performance Metrics

Towards the goal of optimizing the workflow processing according to the above performance metrics, we first identify the critical factors that affect these metrics which is listed as follows.

1. **Initial data scan costs.** It represents the time spent in loading data before the processing. Such initial data scanning requires reading and transferring data from the underlying distributed file system to the local place of the processing instance and recent study [17] shows that initial scanning could take large fraction of the total completion time.

2. **Intermediate data costs.** Intermediate data represents the data generated from the previous operator and will be consumed by the next operator in the workflow. For ease of fault tolerance, most parallel dataflow programming framework materialize these intermediate data when they are generated and delete them after the completion of the workflow. The I/O costs to materialize such data could significantly affect the completion time.

3. **Data transmission costs.** The transmission of intermediate data between operators consists a considerable amount in the total I/O costs. Different data transmission model (e.g. pipelined and blocked) could also affect the completion time.

4. **Scheduling affect.** The parallel execution of a data processing workflow needs to schedule the operator to different cluster nodes. (i.e, mapping bestween operators and the instances). Different scheduling strategies here can be applied which will lead to different completion time or monetary costs.

3 MapReduce: Workflow Processing Framework on the Cloud

MapReduce [6] has been widely recognized as an important tool for large-scale data analysis in the context of cloud computing. It is now widely used in the industry including companies like Google, Facebook, Yahoo! The research communities are also interested in using it for scientific related data process including biological information analysis, scientific simulation and graph data processing.

We will focus on MapReduce in this report because its popularity and well-understood execution model that provide a useful starting point for further exploration. In addition, although the optimization techniques proposed are based on the MapReduce framework, we believe that the optimization strategies described are general and can be applied to other workflow processing framework such as Dryad [9].

3.1 Architecture

In the MapReduce model, computation is abstracted as two functions: map and reduce. Once the user specified the data analysis logic as the map and reduce functions, the underlying MapReduce execution engine will automatically parallelize, distribute, and execute the work on a cluster of commodity machines.

Figure 2 shows the architecture of the MapReduce framework. It contains two basic stages: map and reduce stage. In the map stage, the map task reads

the input data (which is typically stored in a distributed file system) and parses them into input key/value pairs. Then it applies the user-defined map function on each pair to generate an intermediate set of key/value pairs. The map tasks sort and partition these data for different reduce tasks according to a partition function.

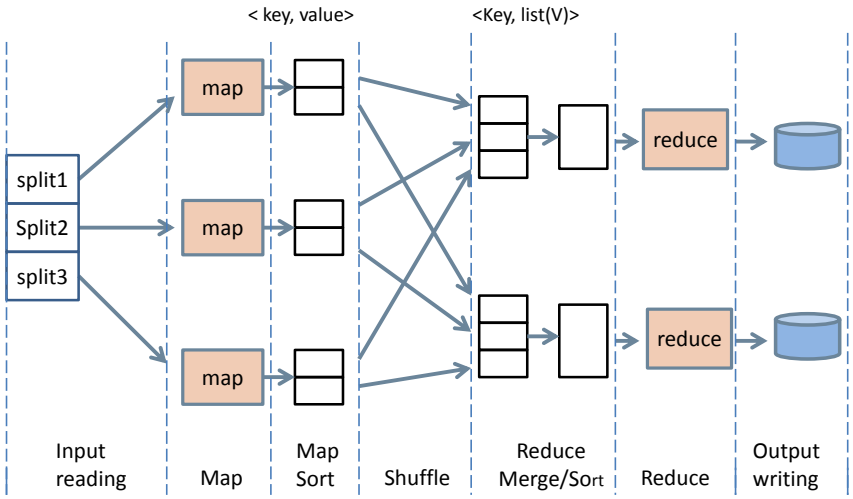

Figure 2: Architecture of MapReduce framework

In the reduce stage, the reduce task fetches its partition of intermediate key/value pairs from all the map tasks (called the shuffle phase). Then it merges these intermediate values with the same key and applies the user-defined reduce function to the value list with the same key to produce aggregated results (called the reduce phase). These results consists the output of the MapReduce job which will be typically stored back to a distributed file system.

Optionally, a job could define a combiner function that aggregates the map outputs. It takes a key and a subset of associated values and produces a single value. The combiner function is useful when it efficiently reduces the amount of data that need to be transferred to the reduce tasks.

The execution of a MapReduce job is controlled by the master node which is called the JobTracker. There is one centralized JobTracker for the entire cluster and the rest instances are workers. The workers periodically connect to the JobTracker to report its current status and the available resources. The JobTracker is responsible for using the reported information for scheduling task to workers with available resources, monitoring the status of the task execution and handling failures and speculative executions.

MapReduce Based Data Processing Framework To enable programmers to specify complex MapReduce based data analytics tasks in an easier way , several projects, such as Pig [7] and Hive [16], provide high-level SQL-like abstractions on top of MapReduce engines. These frameworks allow complex analytics tasks to be expressed with high-level, declarative language and the execution engine will compile the program into directed acyclic graphs (DAGs) of MapReduce jobs.

The following specification shows a simple example of a Pig program. It describes a task that operates over a table *URLs* that stores data with the three attributes: `(url, category, pagerank)`. This program identifies for each `category` the `url` with the highest `pagerank` in that `category`.

URLs = ***load*** *'dataset' as (url, category, pagerank);*
groups = ***group*** *URLs by category;*
result = ***foreach*** *groups generate group, max(URLs.pagerank);*
store *result into 'myOutput'*

The example Pig program is compiled into a single MapReduce job. However, typical Pig programs are more complex, and can be compiled into an execution plan consisting of several stages of MapReduce jobs. Figure 3 shows a possible DAG of five MapReduce jobs $\{j_1, j_2, j_3, j_4, j_5\}$, where each node represents a MapReduce job, and the edges between the nodes represent the data dependencies between jobs.

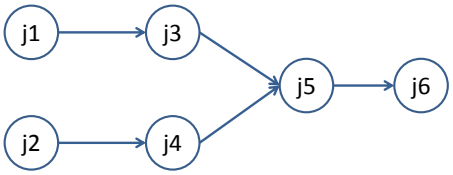

Figure 3: Example of a Pig program execution plan

3.2 Batch Oriented Data Analytics Processing

We present in this section the widely used open-source MapReduce implementation: Hadoop MapReduce [18] that contributed mostly by Yahoo! which aims at batch data processing.

"Pull" based data transfer model The Hadoop MapReduce implementation adopts a "Pull" based data transfer model with more details as follows.

In the map stage, each map task applies the map function to generate intermediate key/value pairs. It sorts and buffers these intermediate data and spill them into the local disk when the buffer is full. After all the input data are processed, the map tasks perform a merge sort of all the spilled file to generate a single final file.

In the reduce stage, the reduce tasks will fetch the intermediate data by copying them from the corresponding local files from all the completed map tasks. After all the data are fetched, it merges and sorts them into a single file and applies the reduce function to the merged data to generate the final output.

3.2.1 Fault tolerance

Since the MapReduce framework is typically deployed in the cloud environment which is based on large number of commodity machines, failures become quite common in that case. As a result, fault tolerance becomes an important function of the MapReduce framework. With the pull based data communication model, the Hadoop MapReduce provides simple and automatic failure recovery strategy.

Task failure The JobTracker receives periodical report from each worker which contains the statue for all the tasks assigned to that worker. Once a task failure is reported, the JobTracker simply re-executes the failed task on a different worker. For map tasks, when the failed task re-executes and generates new map outputs, the reduce tasks will be informed to fetch the new data from that task. For reduce task, as all map outputs are materialized in the local disk, the reduce task can fetch again the corresponding data from all the completed map tasks and re-execute the reduce function on them.

Worker failure As the JobTracker receives periodical report from each worker, if no response is received from a worker in a certain amount of time, the JobTracker marks the worker as failed. All the map tasks assigned to that worker (including those which are already completed) need to be re-executed since the map outputs are stored in the local machine and is lost when the worker failed. The JobTracker also re-executes all the incomplete reduce task on the failed worker.

Master failure According to the design of the MapReduce framework, there is only a single JobTracker, its failure is unlikely and no particular recovery strategy is proposed in the Hadoop implementation. However, it is easy to make the JobTracker write periodic checkpoints of the master data structures. If the JobTracker dies, a new copy can be started from the last checkpointed state.

3.3 Continuous Data Analytics Processing

In this section, we describe Hadoop Online Prototype (HOP) [5]: an alternative implementation of the MapReduce framework that provides directions to support interactive and continuous applications.

"Push" based data flow model Instead of the Hadoop MapReduce which materializes all the map outputs to the local disk before the start of reduce phase, HOP tries to "push" data directly from map to reduce tasks. Specifically, in the aggressive push model, when a client submits a new job, the JobTracker assigns the map and reduce tasks associated with the job to the available TaskTracker slots. To simplify the discussion, we assume that there are enough free slots to assign all the reduce tasks for each job[1]. Each reduce task contacts every map task upon initiation of the job, and opens a TCP socket which will be used to pipeline the output of the map function. When an intermediate key/value pair is produced, the map task determines which partition (reduce task) the pair belongs, and immediately sends it via the appropriate socket.

The problem with such data transmission model is that first it can not make use of the combiner function which could significantly increase the amount of data transferring between map and reduce stage. A related problem is that as there is no sort work in the map side, it pushes more work on the reduce side which may lead to longer reduce phase. It will also bring problems for fault

[1]For cases when these are not enough slots to hold all the reduce tasks, the map tasks store the key / value data in local disk for those reduce tasked which is not scheduled and transmit the stored data in the traditional way when they are scheduled.

tolerance since no intermediate data is materialized, whenever a reduce task fails, it will bring

To address these problems, HOP proposes a refined data transmission strategy. Instead of sending the output to reducers directly, it buffers the map outputs until it grows to a threshold size. The map task then applies the combiner function, sorts the output and spills the buffer contents to disk. If the reduce tasks have enough resources to process the outputs, the spill file will then be sent to the reduce tasks. However, if the reduce tasks fall to keep up with the production of the map tasks, the map task accumulates multiple spill files until the reduce tasks catch up the speed. Then it merges the accumulated spill files into a single file, applies the combiner function on it and then and sent the combined results to the reducers.

HOP also supports data pipelining between jobs with a workflow. Similar to the pipelining between the map and reduce stage, the reduce tasks of the previous job can pipeline their output directly to the map tasks of the next job, sidestepping the need for waiting for the completion of the reduce stage of previous jobs

3.3.1 Fault tolerance

In HOP, the map outputs are sent to the reduce tasks in a pipelined fashion, such implementation makes it more difficult to recover from failures and new techniques need to be introduced to achieve fault tolerance.

Map task failure To recover from map task failures, the reduce tasks keep the record which map task produces each pipelined spill file. The reducer treats the output of a pipelined map task as "tentative" until the map task has committed successfully. The reducer only merge together spill files generated by the same uncommitted mapper. Thus, if a map task fails and re-executes, each reduce task simply ignore any tentative spill files produced by the failed map task and starts to receive data generated by the re-executed map task.

Reduce task failure If a reduce task fails and a new copy of the task is started, the new reduce instance must collect all the data that was sent to the failed reduce attempt. To enable reduce failure recovery, map tasks retain their output data on the local disk for the complete job duration. This allows the map outputs to be resent if any reduce tasks fail. Note given such implementation, the I/O costs of HOP is the same as the Hadoop MapReduce. However, the key advantage of HOP is that the start of reduce tasks is not blocked waiting for the completion of the map tasks.

3.4 Summary

In this section, we present alternative implementations of the MapReduce framework: The Hadoop MapReduce and Hadoop Online Protocol as well as the MapReduce based workflow processing frameworks.

Hadoop MapReduce focuses on batch oriented data processing. Its simple map / reduce interface and the automatic parallelization and failure handling

makes it popular for large-scale batch data processing in the cloud environment. However, it also introduces high system overheads e.g. materialize all the intermediate data within a workflow.

Hadoop Online Protocol implements a pipelined data transmission model during MapReduce execution. It breaks the barrel between the map and reduce stage and allows the system to achieve short response time by sacrificing the accuracy (i.e, with partial results). Two typical applications which will benefit from such architecture: 1) online aggregation that provides quick and possible incomplete results with refined results returned later. 2) continuous MapReduce applications which analyze continuous arriving data and return results in real-time such as event monitoring. Besides, by pipelined data transmission, it provides better overlapping of the map and reduce execution which increases the system utilization.

However, such pipelined data transmission makes HOP more complex for failure handling. For example, all the reduce tasks need to keep tracking the source of each input key / value pair, i.e, which map task generates the data so that when a map task fails, it can discard all the intermediate data received from that task. In addition, though HOP can achieve much shorter response time, it may hurt the completion time in some cases. One reason is that the I/O costs in HOP is not reduced compared with the original Hadoop implementation as for fault tolerance it still need to materialize all the intermediate data (as a background process with the pipelined data transmission). Besides, as the pipelined data transmission pushes more sort work to the reduce side, it may cause performance degeneration in cases when reduce task execution time is the bottleneck.

4 Optimization for Workflow Specification

Given a MapReduce based workflow, several techniques can be applied to optimized the workflow specification to minimize the potential cost incurred during the execution of the workflow. The following sections describe two representative strategies in this category. The first tries to reduce the initial scanning cost by sharing the data scan process for jobs reading from the same input file. The second aims to reducing the potential intermediate data generated during the workflow execution.

4.1 Optimization to Reduce The Initial Scan Costs

During the execution of a MapReduce job, the first step is to load the input data to the processing instances. For large scale data analytic tasks, such input data could be of terabyte size or even larger and data loading is expensive since it involves intensive I/O operations and network data transmission. Recent study [17] shows that initial scanning could take 80% of the total completion time according to the experiments based on the Pig performance benchmark PigMix [2].

On the other hand, evidences show that there could be equivalent or similar work within the same workflow and across different workflows. The most common case for such similar work is that different jobs may read from the same input file. Based on these observations, [12] propose a sharing framework called

MRShare that aims to share similar works among different jobs to reduce the overall completion time.

4.1.1 Overview

The general idea of scan sharing is to transform a batch of MapReduce jobs reading from the same input file into a new batch that will be executed more efficiently, by merging jobs into groups and transform each group into a single job. Figure 4 shows an example of such merge transformation. where J_{11} and J_{12} require the same input and are merged as a single job.

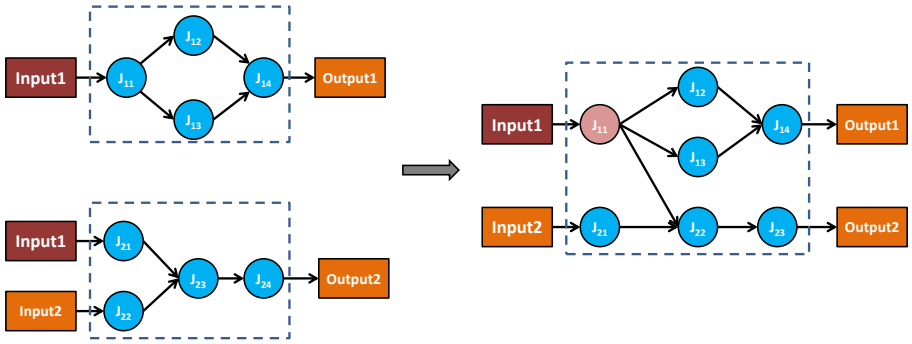

Figure 4: Merge workflow to reduce initial input scan

However, greedily grouping all jobs reading from the same input is not always the optimal choice since the merged job could add extra sorting costs as more map outputs are generated compared with each individual job. Specifically, during the final merge sort in the map side, there will be more I/O costs as there are more spilled files. It brings negative effect to the overall completion time if such extra I/O costs surpass the work reduction from eliminating duplicate input scan.

As a result, the optimization problem is formulated as follows: Given a set of jobs $J = \{J_1, ..., J_n\}$ that read from the same input file, group the jobs into S non-overlapping groups $G_1, G_2, ..., G_S$, such that the total savings of the I/O costs $\sum_{1 \leq i \leq S} Gain(G_i)$, is maximized.

4.1.2 Optimization algorithm

In [12], the authors propose a dynamic programming to solve the about optimization problem[2]. The optimization algorithm is based on the observation that if we sort all the candidate jobs into a list according to the jobs map-output-ratios, the optimal solution will consist of consecutive jobs in the list. Thus, the optimization problem is transferred to split the sorted list of jobs into sublists, so that the overall savings are maximized.

[2]The dynamic programming is proposed to solve a relaxed version of the optimization problem when the extra sorting overheads relies only on the job that contains the highest map-output-ratio in a group as the original problem is proved to be NP-Complete.

For the transferred problem. A dynamic programming algorithm is proposed. The general idea is suppose in the optimal solution, the last sublist starts from job J_i, than the problem is reduced to find out the optimal splitting strategy for the previous $i-1$ jobs. The algorithm will search all the possible is to find out the optimal solution. The pseudo code of the algorithm is shown in Algorithm 1, where c_l represents the total saving of the sublists.

Algorithm 1 SplitJobs

Input:
A set of N jobs sorted according to map-output-ratios $J = \{J_1, J_2, ...J_N\}$
Output:
The maximum saving c and the corresponding grouping s

```
1: for 1 ≤ l ≤ N do
2:   for 1 ≤ i ≤ l do
3:     Compute cost saving Gain(i, l) by grouping job j_i...j_l
4:     c(l) = max_{1≤i≤l}{c(i − 1) + Gain(i, l)}.
5:     Set s(l) as the corresponding grouping for c(l)
6:   end for
7: end for
```

Algorithm refinement After applying Algorithm 1 to the input set of jobs reading the same input, the return results could contain groups with single jobs. A refinement of the algorithm is to separate these single jobs to form a new job list and apply again Algorithm 1 on it to explore more sharing opportunities. These process can be applied repeatedly until no more jobs can be merged.

4.1.3 Summary

The MRShare framework provides automatic scan-sharing optimization across multiple MapReduce jobs which leads to less scan costs so as the reduced overall completion time.

However, as the goal of MRShare is to minimize the overall I/O costs of a set of jobs, it may sacrifice the performance for each individual jobs as the completion time for the merged job is probably longer than executing any of the constitute jobs individually. As a result, the system designer should be careful when apply this technique in a multiuser environment when fairness among jobs is an important concern.

Besides sharing the initial input scan, [12] also talks about more sharing opportunities including sharing the map outputs and even part of the map functions. However, the benefit analysis and cost model for such advanced sharing is still in a preliminary stage which requires further investment.

4.2 Optimization to Reduce Intermediate Data

The intermediate data generated during a workflow represents the output of the previous data operator which that will be consumed as the input of the next operator. For MapReduce based data processing workflows, the intermediate data include two parts: the intermediate date between the MapReduce jobs and

the intermediate data within the MapReduce job (i,e, the map outputs which needs to be shuffled to reduce tasks). In the current MapReduce based workflow processing framework, the system usually materializes these intermediate data for fault tolerance and will delete them after the completion of the entire workflow. Such data materialization incurs considerate overheads especially for complex workflows contain large number of jobs.

4.2.1 Overview

To reduce the intermediate data generated during the execution of a workflow, [19] presents an optimization framework called AQUA for MapReduce based relational data process. The general idea of AQUA includes two parts. The first is to reduce the data operator within a workflow by implementing each operator with more complex functions. The second is to optimize the workflow specification by using better execution plan to minimize the intermediate data.

4.2.2 Reducing the data operators within a workflow

To reduce the data operators within a workflow, [19] focus on the join operation in relational data processing. The default join strategy in Pig and Hive is distributed symmetric hash job[3]. and each join operation will be compiled into a MapReduce job based on the default strategy. For complex data analytics tasks which require join of multiple tables, the default join strategy will lead to e a sequence of MapReduce jobs. Figure 5 shows an example workflow specification to join 3 tables T_1, T_2, T_3.

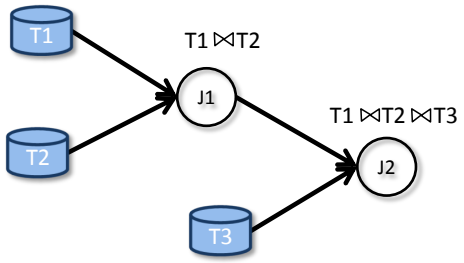

Figure 5: Workflow to join three tables

The problem with the default join strategy is that it lead to more intermediate data as many data operators will be generated for perforating a complex data analytics task. To address the problem, [1] proposes a replicated join algorithm which is able to process join of multiple tables with a single MapReduce job. The general idea of replicated join is to send each intermediate key / value pair generated by the map task to multiple reduce task so that all the input records which can be joined together (i.e, records with the same value of the join keys will be sent to the same reduce task).

[3]Another join implementation is map side join which is applied when one of the input size is small enough to be fully cached in memory or both input tables are co-partitioned by the join key. In the first case, the mappers fully load the small table into memory and scan the other table to perform in-memory hash join. In the second case, each mapper loads a co-partition from both table and performs a local symmetric hash join.

We show the details of the replicated join with a simple example shown in Figure 6 to process query $T_1 \bowtie_{T_1.k_1=T_2.k_1} T_2 \bowtie_{T_2.k_2=T_3.k_2} T_3$. There are 2 join attributes, k_1 and k_2 and suppose we have 3 map tasks and 4 reduce tasks (2 for each join attribute). For a tuple t of T_1, we generate two <key/value> pairs, the value is t in both pairs and the keys are $< hash(t.k_1)\%2, 0 >$ and $< hash(t.k_1)\%2, 1 >$ respectively. Similarly, we also generate two pairs for a tuple t' of T_3 with keys $< 0, hash(t'.k_2)\%2 >$ and $< 1, hash(t'.k_2)\%2 >$. However, only one pair with key ($< hash(t''.k_1), hash(t''.k_2) >$) is created for a tuple t'' of T_2, as T_2 contains both join attributes. In this way, each tuple of T_1 or T_3 will be shuffled to two reduce tasks, and all reduce tasks can process their local joins individually.

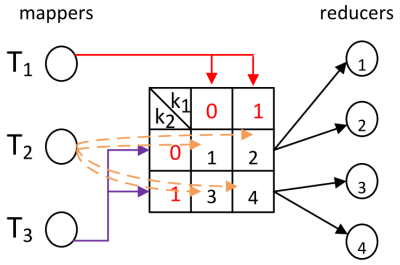

Figure 6: Replicated join of three tables

Compared to the default hash join algorithm, replicated join reduces number of MapReduce jobs (i.e, the data operators) within a workflow which leads to less intermediate data between the jobs. However, it incurs more shuffling costs by forwarding a tuple to multiple reducers(i.e, more intermediate data within a job). As a result, it is not always optimal to perform a replicated join on all the participant tables. To address the problem, AQUA proposes an adaptive algorithm to select the optimal join strategy for a data analytic task involving joining of N tables which is described as follows.

Joining graph Given a data analytic task which requires to join N tables. The joining graph is defined as a undirected graph $G(V, E)$ where the nodes in G represent the participating tables and the edge between the nodes represents the joining relation between these two tables.

Covering set A possible join strategy can be represented as a covering set of the joining graph G, which is defined as a set of disjoint sub-graph S that satisfies 1) The nodes of G is covered by the union of of the nodes in all sub-graphs, i.e. $G.V = \bigcup_{\forall G_i \in S} G_i.V$ and 2) all the subgraphs are well connected. i.e. $\forall n_x and n_y \in G_i$, there exists path in G_i that connects n_x and n_y.

With the covering set on joining graph G, AQUA compiles each sub-graph G_i into a single MapReduce job. Specifically, if $|G_i.V| = 2$, it uses the default symmetric hash join and if $|G_i.V| > 2$, then the replicated join is used. The cost saving of applying replicated join is defined as

$$Gain(G_i) = C_{rjoin}(G_i) - C_{hjoin}(G_i) \quad (1)$$

where $C_{rjoin}(G_i)$ denotes the cost of replicated join for G_i and $C_{hjoin}(G_i)$ is the estimated costs of the best plan using symmetric hash join to process G_i[4].

AQUA tries all the possible covering set (i.e.join strategy) to find out the one with maximal cost savings. The iteration is implemented by adaptively linking the sub-graphs with more details in [19]. The complexity of the searching process is $2^C - 1$ where C represents the number of edges in the joining graph. This approach works as C usually is a small value.

4.2.3 Reducing the intermediate data generated

The motivation of the optimization strategy is that given a data analytics task, we can generate different workflow specification as long as the final results are correct. Depending on the data property (e.g data distribution), different specification could lead to significantly different size of intermediate data. For example, Figure 7 shows two different ways to join 4 tables S, N, T, O, in the left one, a MapReduce job is used to perform $S \bowtie N$, and the results are written back to underlying file system after the job is done. Then, a second job is initiated to join the results of the first job with T . After the second job is done, the results of $S \bowtie N \bowtie T$ will be join with T to get the final results. While in the right one, two MapReduce jobs will be launched for processing $S \bowtie N$ and $T \bowtie O$ respectively. The results of these two jobs will both be joined together by another MapReduce job to get the final results.

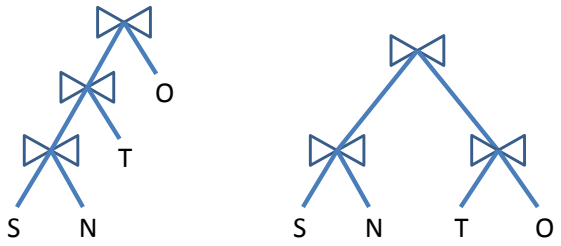

Figure 7: Different join plans

The intermediate data generated from the left plan contains $S \bowtie N$ and $S \bowtie N \bowtie T$. On the other hand, the intermediate data generated from the right plan contains $S \bowtie N$ and $T \bowtie O$. With certain information (e.g, join selectivity), we can easily select a better plan from these two by comparing the estimated size of their intermediate data.

For a data analytic workflow which contains join N tables. A possible solution is to recursively compare all the possible plans to find out the one with the minimal intermediate data. However, the searching space is extremely large in that case ($O(N^{2^N})$, even if only the left-deep plans are considered [15]) and the overhead to exploit all the plans is unacceptable. Therefore, a heuristic approach is employed in AQUA to prune the search space. Specifically, AQUA will iterate all the possible join plans, however, it adopts heuristics to identify inefficient plans as early as possible and only estimate and compares interme-

[4]The best plan here means the the best execution order of these join operators which leads to the minimal intermediate data.

diate data size for the plans that survive the pruning process. The pruning heuristics are based on the following ideas.

- It prunes equivalent sub-plans. i.e, it will not consider the current plan if there is already equivalent one contained in the candidate set.
- It prunes obvious inefficient plans. Specifically, For example, if $R_1 \bowtie R_2$ generates significantly more results than $R_2 \bowtie R_3$, it will not consider any plans contain $R_1 \bowtie R_2$ operator.

4.2.4 Summary

The AQUA optimizes the workflow specification by reducing the potential intermediate data. It first adopts the replicated join to reduce the data operators within a workflow and second, it optimizes the workflow specification which leads to less intermediate data between the MapReduce jobs.

However, this optimization is limited to SQL-like relational data analytic task involving typical data operator like selection, join and aggregation. Specifically, most of the optimization strategies applied only on the join operator. The optimizer is not general enough to optimize for more general data processing workflow on the MapReduce platform, such as sorting and word counting.

4.3 I/O based cost model

In this section, we describe the cost model used in section 4.1 and section 4.2 in determining a better workflow specification. This cost model is I/O based assuming that the job execution time is dominated by I/O operations. Table 4.3 shows the parameters that will be used in the model.

Table 1: Parameters

Parameter	Definition
C_r	cost ratio to read/write data remotely
C_l	cost ratio to read/write data locally
C_t	cost ratio for network transmission
$\|Input\|$	input size for a MapReduce job
$\|Output\|$	output size for a MapReduce job
B	the size of buffer used during sorting
D	the size of map outputs generated by a MapReduce job

Typically, the cost of a MapReduce job contains 4 parts: costs to read the input data T_{read}, costs to write the output data T_{write}, costs to sort the map outputs T_{sort}, and costs to transmit the intermediate data T_{trans}.

$$T(J) = T_{read}(J) + T_{write}(J) + T_{sort}(J) + T_{trans}(J) \tag{2}$$

For a given job j with m map tasks and r reduce tasks. T_{read}, T_{write} and T_{sort} can be estimated as follows.

$$T_{read}(J) = C_r \times |Input| \tag{3}$$
$$T_{write}(J) = C_r \times |Output| \tag{4}$$
$$T_{trans}(J) = C_t \times |D| \tag{5}$$

To estimate the sort cost, two parts should be considered. The first is the sort costs for the map tasks as each map task will sort all the map outputs before writing to its local disk. Suppose each map task generates on average $|M|$ size of map outputs, and each reduce task will receive on average R size of map outputs. Note that $|D| = m \times |M| = r \times |R|$ the sort costs in the map side can be estimated approximately:

$$\begin{aligned} T_{sort\_map}(J) &= C_l \times (m \times |M|(2 + 2(\lceil log_B \frac{|M|}{B+1} \rceil))) \\ &= C_l \times (|D|(2 + 2(\lceil log_B \frac{|D|}{(B+1) \times m} \rceil))) \\ &\approx C_l \times (|D| \lceil 2(log_B |D| - log_B m) \rceil) \end{aligned} \quad (6)$$

At the reduce side, each reduce task need to get the output from m map tasks and merges and sort these m inputs. Therefore, the sort cost at the reduce side is

$$\begin{aligned} T_{sort\_reduce}(J) &= C_l \times (r \times |R| \lceil 2 log_B m \rceil) \\ &= C_l \times (|D| \lceil 2 log_B m \rceil) \end{aligned} \quad (7)$$

As a result, the total cost for the sort phase is

$$\begin{aligned} T_{sort}(J) &= T_{sort\_reduce}(J) + T_{sort\_reduce}(J) \\ &= C_l \times (|D| 2(\lceil log_B |D| - log_B m \rceil + \lceil log_B m \rceil)) \end{aligned} \quad (8)$$

4.3.1 Estimate the cost saving with scan sharing

In section 4.1.2, we describes the strategy to share the input scan for jobs reading the same input file. Here, we give an estimation of the potential benefits by applying the sharing strategy which is used in algorithm 1 to determine the best sharing strategy.

Give a set of N jobs $J = J_1, ...J_N$ reading from the same input file F. The total cost to execute them individually is estimated as

$$T(J) = \Sigma_{1 \le i \le N} T(J_i) \quad (9)$$

where $T(J_i)$ is estimated based on equation 2[5].

On the other hand, the cost of merge these N jobs as a single job J_G and executed once is

$$\begin{aligned} T_{read}(J_G) &= C_r \times |F| \\ T_{write}(J_G) &= C_r \times \Sigma_{1 \le i \le N} |Output_i| \\ T_{sort}(J_G) &= C_l \times |D_G| \times 2(\lceil log_B |D_G| - log_B m \rceil + \lceil log_B m \rceil)) \\ T_{trans}(J_G) &= C_t \times |D_G| \\ T(J_G) &= T_{read}(J_G) + T_{write}(J_G) + T_{sort_G}(J) + T_{trans}(J_G) \end{aligned}$$

[5]We assume that all the jobs has the same number of map tasks m and same number of reduce tasks r. It is reasonable as the number of map tasks are determined by the input size and the number of reduce tasks are specified by the user.

where D_G represents the intermediate data generated by the merged job and is estimated as $D_G = \Sigma_{1 \le i \le N} D_i$

The cost saving by sharing the same input for the set of N jobs is

$$Gain(J) = T(J_G) - T(J) \tag{10}$$

4.3.2 Estimate the cost saving with replicated join

In section 4.2.2, we describe the implementation of replicated join which performs join of multiple tables with a single MapReduce job. Such strategy reduces the intermediate data but it also incurs more shuffling costs by forwarding a tuple to multiple reducers. Here, we give an estimation of the potential benefits by applying the replicated join which is used in section 4.2.3 to determine the best join strategy.

Given a request with $T_1 \bowtie_{T_1.k_1=T_2.k_1} T_2 \bowtie_{T_2.k_2=T_3.k_2} T_3$. With symmetric hash join, suppose the best join plan is to first launch a MapReduce job J_1 for $T_1 \bowtie_{T_1.k_1=T_2.k_1} T_2$ to get a temp result T_{12} and then another MapReduce job J_2 for $T_{12} \bowtie_{T_{12}.k_2=T_3.k_2} T_3$ to get the final results T_{123}.

Suppose the size of T_i is $|T_i|$ and the map outputs generated by J_x are $|D_x|$. The total cost of the workflow is estimated as

$$T_h = T(J_1) + T(J_2) \tag{11}$$

where $T(J_i)$ is estimated based on equation 2 and the input size of J_i is approximated as the total sizes of the input tables, the output size of J_1 can be estimated with the join selectivity, and the size of map outputs is approximately equals to the input size as in the hash join, the map tasks simply emits the input records[6].

On the other hand, with replicated join, we can get the final results for joining the three tables with a single MapReduce job J_r. The cost of J_r is estimated as

$$\begin{aligned}
T_{read}(J_r) &= C_r \times |T_1| + |T_2| + |T_3| \\
T_{write}(J_r) &= C_r \times |T_1 \bowtie T_2 \bowtie T_3| \\
T_{sort}(J_r) &= C_l \times |D_r| \times 2(\lceil log_B |D_r| - log_B m \rceil + \lceil log_B m \rceil)) \\
T_{trans}(J_r) &= C_t \times |D_r| \\
T(J_r) &= T_{read}(J_r) + T_{write}(J_r) + T_{sort}(J_r) + T_{trans}(J_r)
\end{aligned}$$

where $|D_r|$ represents the map outputs generated by J_r. Since in replicated join, each input tuple will be sent to multiple reduce tasks. The map outputs D_r increases. Suppose the number of reduce tasks used to process each join attributes a_i is c_i. Then, each tuple from table T_1 needs to be replicated to c_2 reduce tasks and each tuple from table T_2 needs to be replicated to c_1 reduce tasks. As a result, the map outputs for J_r is estimated as

$$|D_r| = |T_1| \times c_2 + |T_2| \times c_1 \tag{12}$$

More general, Given N tables join together, the number of replicas for each tuple from table T_i can be estimated as

$$r_i = \prod_{\forall a_x \notin A_i \ and \ a_x \in A} c_x \tag{13}$$

[6]For analysis simplification, we ignore the key size here.

where A represents the set of all join attributes and A_i represents the set of join attributes contained in T_i. D_i is then estimated as

$$|D_i| = \sum_{i=1}^{N} |T_i| \times r_i \tag{14}$$

The cost saving by applying replicated join is estimated as

$$Gain(J) = T(J_h) - T(J_r) \tag{15}$$

5 Optimization for Workflow Execution

To execute a given workflow, each operator in the workflow will be mapped to a physical instance for execution. (For MapReduce based workflow, each MapReduce job is consists of a set of map and reduce tasks and each task will be assigned to a physical instance for execution). Such mapping is controlled by the scheduling policy and different scheduling policy will greatly affect the execution performance. As a result, the scheduling policy is critical for the workflow execution and great attention has been paid for scheduling optimizations.

5.1 Overview

There are different goals for a particular scheduler. The completion time and monetary cost are among the most important ones. The monetary cost is related to the completion time. i.e, shorter complete time leads to less monetary. However, they are not always correlated in the cloud environment as the resources in the cloud are charged in a pay-as-you-go model based on a per quantum pricing scheme, e.g. one hour.

Figure 8 shows an example workflow to illustrate the affect of different scheduling on the its completion time and monetary cost. Assume that the execution time of data operator A and C is 1 hour and the execution time of each data operator B is 10 minutes. Assuming that the data transfer time is negligible.

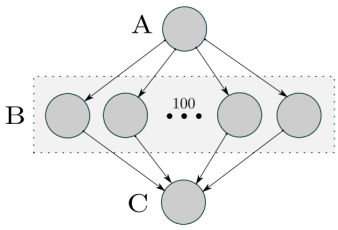

Figure 8: Example workflow with 3 different kind of operators

Consider two possible schedules for this workflow: Schedule 1: execute all operators with only one instance. The time required for the dataflow to complete is: $60 + 10 \times 100 + 60 = 1120$ minutes or 18.6 hours. Since there is only one host involved, the cost for this schedule corresponds to 19 hours of instance usage. Schedule 2: execute each operator in a different instance (i.e, 102 instance is needed in this case). the completion time is 60+10+60 = 130 minutes or

2.17 hours. However, the cost for this schedule is 102 hours of instance usage. Schedule 2 will run about 9 times faster than Schedule 1, but will cost 5 times more money.

[10] proposes a general scheduling algorithm which considers both the completion time and monetary cost. There are two types of optimization problems targets: constrained and skyline. Typical constrained problems are minimize completion time given a fixed budget, or minimize monetary cost given a deadline. Typical skyline problem is to find trade-offs between completion time and monetary cost.

5.2 Optimizing Mapping Between Operators and Instances

Algorithm 2 shows a generic nested loop optimizer proposed in [10] that could solve both the constrained or skyline problems, depending on the values of its parameters.

Algorithm 2 Scheduling

Input:
G: A workflow DAG
CONST: Solution constrains
FILTER: Solution space filter
LIMIT: Contain limit sequence generator
STOP: Stopping Condition
Scheduling optimizer
Output:
Space: the space of solutions

```
1: space ← ∅
2: while LIMIT.hasNext() and STOP.continue() do
3:    limit ← LIMIT.getNext()
4:    next ← OPT(G, limit, CONST)
5:    space ← FILTER(space ⋃ {next} )
6:    STOP.addFeedback(next)
7:    LIMIT.addFeedback(next)
8: end while
9: return space
```

- CONST represents the constraints a schedule should satisfy. e.g the budget or the deadline.
- FILTER prunes inefficient schedule according to performance goal. For example,for the skyline problem, the Filter function will keep the schedule along the skyline and remove the other ones.
- LIMIT is a generator of instance limits. It return the maximum number of instances the scheduler can use.
- STOP determines whether or not to stop the exploration based on certain criteria. For example, we can stop the loop when the last 5 schedules do not differ significantly with respect to completion time.

- OPT is a single-objective optimizer that tries to optimally assign the operators to instances, minimizing either the time or the money related to the schedule.

In this optimizer, the most important module is the OPT function which returns a optimized mapping between the operator and physical instances depending on the optimizing goal. In [10], the authors propose two different ways to find such optimal solution: the Greedy scheduling algorithm and the Local Search scheduling algorithm.

Greedy Scheduling Algorithm It greedily selects one operator from all the ready operators and assigns it to the "best" instance among all the available ones. Depending on the optimization goal, [10] provides several schedulers that have different criteria in selecting the next operator and the corresponding instances. For example, G-MPT tries to minimize the workflow completion time by assigning the next operator with the maximum execution time to the container that minimizes completion time. other schedulers focus on balancing instance utilization or minimizing the monetary cost.

Local Search Scheduling Algorithm It starts with an *init* schedule of the operators, and moves to a *neighbor* scheduling if the *cost* of the neighbor scheduling is less than the original one. The neighbor scheduling is generated by assigning a randomly chosen operator from the current scheduling to a different random-chosen instance. The authors also tried several different scheduler which differ in the init assignment and the cost criteria. For example, SA-MPT begins with a random assignment and the cost criteria is the completion time.

5.3 Cost Model

In this section, we describe the cost model used in section 5.2 in determining the best scheduling according to the completion time or monetary cost.

5.3.1 Completion time modeling

In [10], an operator in a workflow is modeled as op(time, cpu, memory, behavior) where time is the execution time of the operator, cpu is its average CPU utilization measured as a percentage of the host CPU power when executed in isolation, memory is the maximum memory required for the effective execution of the operator, and behavior represents the data transfer mode. There are two different behaviors: pipeline (PL) or store-and-forward (S&F). If behavior is equal to S&F, all inputs to the operator must be available before execution; if it is equal to PL, execution can start as soon as some input is available. It assumes that each operator has a uniform resource consumption during its execution (cpu, memory, and behavior do not change). A flow between two operators: producer and consumer, is modeled as flow(producer, consumer, data), where data is the size of the data transferred.

Depends on the data transfer behavior, the execution time of a flow can be estimated in either of the following two ways:

Time cost for flow from S&F operator In this case, the network communication is modeled with a special operator that performs data transferring called *dt*. This operator is injected between the operators of the flow and the execution time of *dt* is estimated as follows:

$$dt.time = \frac{D_{A \to B}}{min(A.network, B.network)} \tag{16}$$

Time cost for flow from PL operator For pipelined data transferring, no extra operator will be inserted. However, the time cost of the data transferring will be reflected in the completion time of the initiated operator. Specifically, given a flow(A,B, $D_{A \to B}$), the time to completed operator A is increased and the CPU utilization of A is decreased. The time and CPU property of operator A are changed to

$$T = max(A.time + dt.time, B.time + dt.time) \tag{17}$$

$$CPU = \frac{A.time \times A.cpu + dt.time \times DT_{CPU}}{T} \tag{18}$$

where $dt.time = \frac{D_{A \to B}}{min(A.network, B.network)}$.

The instance is the abstraction of the host, encapsulating the resources provided by the underlying infrastructure. An instance is described as cont(cpu, memory, network). When more operators assigned to the same instance. A particular problem arises with the CPU when overlapping operators require at some point more than 100% utilization together. The authors model such case by proportionally increasing the execution time of the participating operator.

5.3.2 Monetary cost modeling

The monetary cost to complete a workflow in a cloud environment given a schedule S_G and the cloud pricing scheme (quantum time Q_t and the unit cost Q_m) is approximated as follows: On each instance, we slice time into windows of length Q_t starting from the first operator executed according to the schedule. The financial cost is then a simple count of the time windows that have at least one operator running, multiplied by Q_m (Illustrated in Figure 9).

$$m(S_G) = Q_m \times (\sum_{i=1}^{|C|} \sum_{j=1}^{|W|} \epsilon(c_i, w_j)) \tag{19}$$

with $C = \{c_i\}$ being the set of instances, $W = w_j$ being the set of time-windows, and

$$\epsilon(c_i, w_j) = \begin{cases} 1 & if\ at\ least\ one\ operator\ is\ active\ in\ w_j\ in\ c_i \\ 0 & otherwise \end{cases}$$

5.4 Summary

The work described in this section introduces an interesting two-dimension scheduling problem which considers both the workflow completion time and the

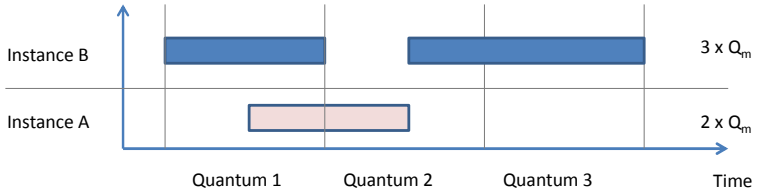

Figure 9: Illustration of the monetary cost model

monetary costs. It proposes a general framework which could solve both the constrained and skyline problem and different heuristic optimization strategies.

However, the scheduling algorithm considers a static environment assuming the in advance knowledge on detailed workflow property and the underlying system setting. For example, the number of tasks within each MapReduce job. However, the number of map tasks of the MapReduce job depends on the input data (Each map task reads a trunk of the input data which it at most 64 MB size) and is not known before the execution[7]. Besides, there are a lot of simplifications in the cost model adopted in this work, for example, it assume the execution time of each operator is the same across different instances. However, it is usually not the case even in a homogeneous environment due to factors like data locality. It also uses a over-simplified model in handling CPU overload. Specifically, it assumes the system knows the CPU usage of each operator scheduled in an instance and proportionally prolongs the execution time of the operator if the instance is overloaded (the total CPU usage of all the operator scheduled on that instance during certain time is beyond 100%). However, such assumption is usually not held in reality as the overlapping of CPU usage among different operators.

6 Summary and Discussion

In this section, we present 4 different works of performance optimization for MapReduce based workflows (We category the work from MapReduce online as another performance optimization strategy here functioning on the MapReduce architecture). Table 2 shows a summary of each optimization strategy regarding to the performance metrics it focus and the factor it targets to improve the performance.

Opportunities and challenges to apply different optimization strategies in combination In practical, depends on the requirements of different applications, we can also try to apply these techniques in combination to get more improvement. For example, we can perform the techniques form AQUA and MRShare together. Suppose we have a set of workflows, we can first optimize each particular workflow through the workflow optimizer and then apply the scan sharing techniques on the workflow set to reduce the overall completion time.

[7]It is possible to estimate the number of map tasks given the input size but the estimation will not be accurate, Moreover, the estimation is even more difficult for MapReduce based workflows since the number of map tasks depends on the output size of the previous job.

Table 2: Summary of different optimization strategies

Optimization Work	Technique	Metric	Factor
MapReduce Online	Pipelined data transmission	Response time	Data transmission costs
MRShare	Scan sharing	Completion time	Initial scan costs
AQUA	Intermediate data reduction	Completion time	Intermediate data costs
Workflow scheduler	Operator scheduling	Completion time / Monetary cost	Scheduling affect

However, there also could be challenges in applying these optimizations. For example, if we apply scan sharing and scheduling policy together, it will possibly degrade the effectiveness of the scheduling policy as scan sharing reduce the number of concurrent operators by merging MapReduce jobs which could lead to inefficient scheduling solution during the execution stage. Also, technically, MRShare and AQUA can also works with the HOP implementation, however, as with pipelined data transmission model, the I/O costs may not the dominate factor affecting the completion time, we may not get much improvement by applying these techniques.

7 Acknowledgments

I would like to thank Prof. Zachary Ives for chairing my WPE-II committee, as well as Dr. Ludmila Cherkasova, Prof. Andreas Haeberlen and Prof. Boon Thau Loo for sitting on the committee.

References

[1] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 99–110, 2010.

[2] Apache. PigMix Benchmark, http://wiki.apache.org/pig/PigMix, 2010.

[3] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[4] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, 2011.

[5] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the*

7th USENIX conference on Networked systems design and implementation, NSDI'10, 2010.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 2008.

[7] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. of the VLDB Endowment*, 2(2), 2009.

[8] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proc. of 5th Conf. on Innovative Data Systems Research (CIDR)*, 2011.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS OS Review*, 41(3), 2007.

[10] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis Ioannidis. Schedule optimization for data processing flows on the cloud. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, 2011.

[11] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, 2010.

[12] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: Sharing across multiple queries in mapreduce. *PVLDB*, 3(1):494–505, 2010.

[13] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, 2011.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a Not-So-Foreign Language for Data Processing. *Proc. of SIGMOD*, 2008.

[15] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 314–325, 1990.

[16] 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. *Proc. of VLDB*, 2009.

[17] X. Wang, C. Olston, A. Sarma, and R. Burns. CoScan: Cooperative Scan Sharing in the Cloud. In *Proc. of the ACM Symposium on Cloud Computing,(SOCC'2011)*, 2011.

[18] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, original edition, 2009.

[19] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[20] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, 2008.