공학석사학위논문

Dynamic Optimization of Large-Scale Data Shuffling in a Data Processing System

데이터 처리 시스템에서의 대규모 데이터 셔플에 대한 동적 최적화

2019 년 2 월

서울대학교 대학원

컴퓨터공학부

이 산 하

Abstract

Dynamic Optimization of Large-Scale Data Shuffling in a Data Processing System

San Ha Lee
Department of Computer Science and Engineering
The Graduate School
Seoul National University

The scale of data used for data analytics is growing rapidly and the ability to process large volumes of data is critical to data processing systems. A scaling bottleneck for processing large amounts of data in the data processing systems is the random disk read overhead that occurs while shuffling data communications between tasks. To reduce this overhead, an external shuffle process can batch the disk read by aggregating the intermediate data through an additional computation. However, the additional computation cannot take advantage of distributed execution capabilities provided by data processing systems such as scheduling, parallelization, or fault recovery. In addition, the systems cannot dynamically optimize the external shuffle process in the same way that they optimize plain jobs without an external process. Instead of launching the external shuffle process, we propose to insert the disk read batching into a job. By doing so, the tasks can fully exploit the features, including the dynamic optimization provided by data processing systems, because the computation for intermediate data aggregation is fully revealed to the systems. Moreover, we suggest a dynamic data skew handling mechanism that can be applied with the disk read batching optimization at the same time. Evaluations show that our implemented technique can mitigate random disk read overhead and data skewness and can reduce the job completion time by up to 54%.

Keywords: Data Processing, Data Analytics, Large-Scale Shuffle, Storage I/O Optimization, Dynamic Optimization, Skew Handling, Data Processing System

Student Number: 2017-28856

Contents

List of Figures

Chapter 1

Introduction

Recently, data analysis applications that process tremendous amounts of data from a few terabytes to petabytes are emerging. These applications, including log processing, machine learning, and transaction processing, are commonly run on distributed data processing systems to take advantage of parallel processing. For example, Facebook announced via [1] that over 20 petabytes of data are generated and processed daily on their Spark [2] cluster. Accordingly, it becomes critical to process these large-scale applications efficiently for the data processing systems such as Hadoop [3], Spark [2], Dryad [4], and Nemo [5].

Such data processing systems parse and execute these applications as small pieces of parallel computation units (called *tasks*) and the communication between them. In most cases, the systems persist the intermediate data to be shuffled between the tasks on hard disk drives (HDDs) for the sake of persistency and cost efficiency. However, when tasks read the intermediate data on the disk, multiple tasks simultaneously read a single intermediate data file generated by a producer task and random disk reads occur. Furthermore, as the size of the intermediate data grows, this overhead increases super-linearly. Therefore, the data shuffle becomes a scaling bottleneck for the data analysis applications.

There are several research solutions that tailor a data processing system to large-scale data shuffles by batching the disk read [1, 6]. These solutions launch a separate process on each worker machine or a separate distributed file system on the worker machines that performs intermediate data aggregation. The aggregation process or distributed file system groups the

intermediate data that a single consumer task will read and stores it in a file on the disk. This aggregation allows consumer tasks to sequentially read only one file for each task.

However, because these solutions maintain a computing process detached from the main procedure of application execution, the process (or another component) must provide additional functionalities, including computation scheduling, parallelization, and fault recovery, that are already provided by the data processing systems for ordinary tasks. Moreover, since the distribution of data from producer tasks to consumer tasks must be determined before the data shuffle, it is difficult to apply other optimizations (especially, dynamic optimizations) for the shuffle, such as data skew handling, at the same time.

Instead of performing intermediate data aggregation in a separate computing process, we suggest inserting the disk read batching into a job. This reveals the disk read batching step to the data processing system, and makes it naturally take advantage of the features of the system, including computation scheduling, parallelization, and fault recovery. In addition, exposing the disk read batching to the system in the form of tasks enables the system to apply other optimizations transparently.

To demonstrate the merit of this exposure, we propose a dynamic data skew handling technique that can be combined with our disk read batching optimization. By dynamically calculating the data size histogram of the intermediate data generated by sampled producer tasks, the intermediate data communication can be reconfigured before the original shuffle to mitigate data skew.

We implemented this technique on Apache Nemo and evaluated the performance benefit of it. The evaluation results show that our technique reduces the job completion time of a map-reduce job for 1 terabyte (TB) of non-skewed data by 54% and reduces the job completion time of a TPC-H benchmark query for 100 GB of skewed data by 31% with proper skew handling.

Chapter 2

Background

This chapter describes the overall concepts and problems we focus on. Section 2.1 explains how a distributed data processing system parses and executes data analysis applications. Section 2.2 describes the random disk read overhead that occurs during a data shuffle. In Section 2.3, we demonstrate the existing solutions to alleviate the random disk read overhead, including how they handle computation scheduling, parallelization, and fault recovery. In Section 2.4, we illustrate the difficulty of combining dynamic optimizations for the shuffle with the example of data skew handling.

2.1 Distributed Data Processing Concepts

Distributed data processing systems split a data analysis application into a number of partial computations. Typical MapReduce [7] frameworks like Hadoop [3] divide their applications into two computational phases called map and reduce, and data communication between these computations. To break away from this convention, many recent data processing systems, such as Spark [2], Dryad [4], and Nemo [5], represent data analysis applications as job directed acyclic graphs (DAGs). A job DAG contains nodes representing the partial computation of the application and edges representing the data communication between the computations.

These systems provide common functionalities for executing parsed applications including computation parallelization, scheduling, and fault recovery. At first, the job DAG is di-

vided again into partial DAGs called *stages*. Typically, computations that can be pipelined are grouped as a stage. Data processing systems parallelize the stages into computation units (i.e., tasks) and schedule them. The data communication at an edge is defined by a *communication pattern*. The communication pattern includes the one-to-one pattern in which a consumer task solely consumes the output of a producer task, the broadcast pattern in which every consumer task consumes the entire output data of all the producer tasks, and the many-to-many pattern (shuffle) in which a consumer task consumes a designated part of the output data from every producer task. The output of each task is called a *block*, which is a part of the intermediate data, and each block is divided into many *partition*s, which are units of data communication. The procedure that collects output data into partitions is called *partitioning*. This partitioning is usually determined by the communication pattern, but can be customized by the system.

2.2 Random Disk Read Overhead in the Data Shuffle

As the amounts of data processed by applications increases, it becomes more important for data processing systems to efficiently store and deliver large amounts of intermediate data. The randomness of data reading is a crucial characteristic of data processing that affects the performance of data communication. As mentioned above, when a task produces an output block to be shuffled, the block consists of many partitions that can be read by many tasks. Because the scheduling and execution order of tasks is determined at run-time based on several factors, such as the number and status of worker machines, data locality, and so on, read requests for a partition also occur randomly. Notably, the random read performance of a HDD is definitely lower than the sequential read performance due to the disk seek, as it is known. Therefore, the randomness of data reading degrades the performance of the data shuffle when it encounters the characteristics of the HDD.

Unfortunately, it is very common to persist intermediate data to be shuffled on HDDs to reduce re-computation cost. When a single task that reads some of the shuffled data that

is not storage fails, the entire tasks that generated the data must be re-executed. However, the failure of tasks is considered a norm for distributed data processing. Furthermore, it is too expensive to maintain large amounts of intermediate data in non-disk storage, such as memory or solid state drive (SSD). As a result, many data analysis applications shuffle data on HDDs while taking the random disk read overhead for every partition read request.

To make matters worse, the number of random read requests in the shuffle increases quadratically with the size of the intermediate data. As the size of the input data of an application becomes larger, the number of consumer tasks and intermediate data files usually increases, and each consumer task reads a smaller part of all intermediate data files. When there are P producer tasks and C consumer tasks connected with a shuffle edge, an output block from a producer task consists of C partitions. In general, P and C are linearly proportional to the input data size. Accordingly, the total number of random disk reads ($P \times C$) is quadratically proportional to the input data size. Consequently, every data request only reads a small amount of data with random disk read overhead, and the overhead becomes a scaling bottleneck.

2.3 Existing Solutions

To mitigate the random disk read overhead of the shuffle, Sailfish [6] and Riffle [1] aggregate the intermediate data to batch the disk read.

- **Sailfish** [6] is a MapReduce framework optimized for the large-scale data shuffle. Sailfish maintains a customized version of a separate distributed file system for intermediate data aggregation. When a producer task generates intermediate data to be shuffled in Sailfish, the task sends output data to the distributed file system immediately, instead of to a local disk. A computation process called chunkserver runs on each cluster machine forming the distributed file system and it aggregates and sorts the intermediate data. During this aggregation, all data to be read by a single consumer task is

grouped and stored in a single file, called an I-file, according to the partitioning policy of the shuffle. Because there are a fixed number of I-files, an I-file can be read by many consumer tasks. After this aggregation procedure, consumer tasks can read the intermediate data almost sequentially from I-files, rather than reading from many spread out data files and taking the random disk read overhead.

Because the chunkserver is designed as an on-demand process and runs on each cluster machine, the MapReduce framework cannot control the scheduling and parallelization of the aggregating computation. Moreover, when an I-file is lost, the framework recomputes all producer tasks to re-generate the I-file.

- **Riffle** [1] is an optimized shuffle service for the large-scale data shuffle. Riffle launches the merger scheduler process and the Spark scheduler on the driver and the external shuffle service process and the Spark executors on each worker machines. When Spark requests a data shuffle for Riffle, the merger scheduler schedules aggregating computations for the shuffle service processes. Through the aggregation, all data for a consumer task is grouped and stored in a single file, like in Sailfish. Therefore, consumer tasks can read intermediate data sequentially.

  Instead of compromising the speed of the fault recovery process like Sailfish, Riffle provides a customized fault recovery mechanism. Riffle stores the original intermediate data files as well as the aggregated files. If an aggregated file is lost, the consumer task can read data from the original files (and the partially aggregated files, if possible) without recalculating the producer tasks.

The above solutions efficiently alleviate the random disk read overhead of large-scale data shuffling by aggregating the intermediate data. However, the distributed file system or the scheduling and computing processes for intermediate data aggregation are hidden from the sight of the data processing system. Because of this, the additional computation for aggregation cannot take advantage of the system's basic functionalities for tasks, such as com-

putation scheduling, parallelization, and fault recovery. Furthermore, since the aggregation process is separate from the optimization layer of the data processing system, the system cannot modify the data distribution of the shuffle and aggregation directly; instead, the system must modify the action of the external processes. This problem gets worse when trying to modify the shuffle at run-time.

2.4 Skew Handling with Disk Read Batching

Data skew handling optimization is an example of optimization that is not easily compatible with the previous disk read batching solutions. Data skew of shuffled data in distributed data processing is a general problem, and there are several frameworks that handle the problem, such as Themis [8], SkewReduce [9], Starfish [10], Optimus [11], and Hurricane [12]. These solutions can be roughly classified into two types: input data sampling and dynamic redistribution. However, these solutions are difficult to apply with the previously described disk read batching solutions.

- **Input Data Sampling**

  Themis, Starfish, and SkewReduce are MapReduce frameworks that handle data skew with the input data sampling method. This type of framework has some sort of sampled execution phase before the original application execution. In this phase, the framework executes all or part of the original application for some sampled input data and collects an ∂intermediate data size metric for each key. After this phase, the framework reconfigures the distribution of the intermediate data to mitigate the data skew according to the collected metric and executes the original application with the optimized intermediate data distribution.

  However, this kind of technique cannot be applied to the recent data processing systems that process arbitrary job DAGs. When a part of the application is executed for some sampled input data, the contents of the intermediate data after a few shuffle or broadcast

edges are significantly different from the contents when the application is executed for the full input data. Therefore, the shuffles in later part of the job DAG cannot be optimized correctly with this kind of technique.

- **Dynamic Redistribution**

  Optimus and Hurricane are data processing systems that handle data skew with an dynamic redistribution method. This type of system collects the intermediate data size metric per key while executing the producer tasks of an arbitrary shuffle in the job DAG without any sampled execution phase. When the producer tasks are completed, the system dynamically reconfigures the intermediate data distribution to resolve the data skew according to the collected metric. Afterward, each consumer task consumes the data reassigned to the task.

  However, the disk read batching techniques aggregate the intermediate data according to the partitioning policy during the execution of the producer tasks. Because of this, the distribution of the intermediate data from the producer tasks to the consumer tasks must be determined before the execution of the producer tasks. Therefore, this kind of data skew handling is not compatible with the disk read batching.

For the above reasons, the existing disk read batching techniques are not easily compatible with the data skew handling techniques. In fact, Riffle does not provide any skew handling mechanism. Sailfish modifies the number of consumer tasks for each I-file according to the intermediate data size metric. However, this approach is only effective if the skewness is insignificant. Otherwise skew handling will not work properly and the random disk read will occur again.

Chapter 3

Disk Read Batching as a Task

In order to overcome the limitations specified in Chapter 2, we suggest to insert the disk read batching optimization into a job. Section 3.1 describes how to conduct the disk read batching by inserting an intermediate data aggregation stage. Section 3.2 proposes a hybrid version of data skew handling optimization that is compatible with the inserted intermediate data aggregation stage.

3.1 Intermediate Data Aggregation Stage

To reveal the disk read batching optimization to the data processing system, we insert a stage that performs intermediate data aggregation before the large-scale shuffle edge in the job DAG. The tasks of the inserted aggregation stage emits input data immediately without any computation. By composing this simple task and a few functionalities provided by the system, the disk read during shuffles can be batched without separate processes or a distributed file system.

Figure 3.1 illustrates the insertion of the data aggregation stage into a simple map-reduce job as an example. (a) represents the original map-reduce job DAG. The DAG has a map stage and a reduce stage connected by a shuffle edge. As mentioned earlier, each output block of a map task is read by many reduce tasks in a random order. (b) represents the optimized job DAG with the aggregation stage inserted. The aggregation stage receives the shuffled data from the map stage and is connected to the reduce stage by an one-to-one edge. Because the

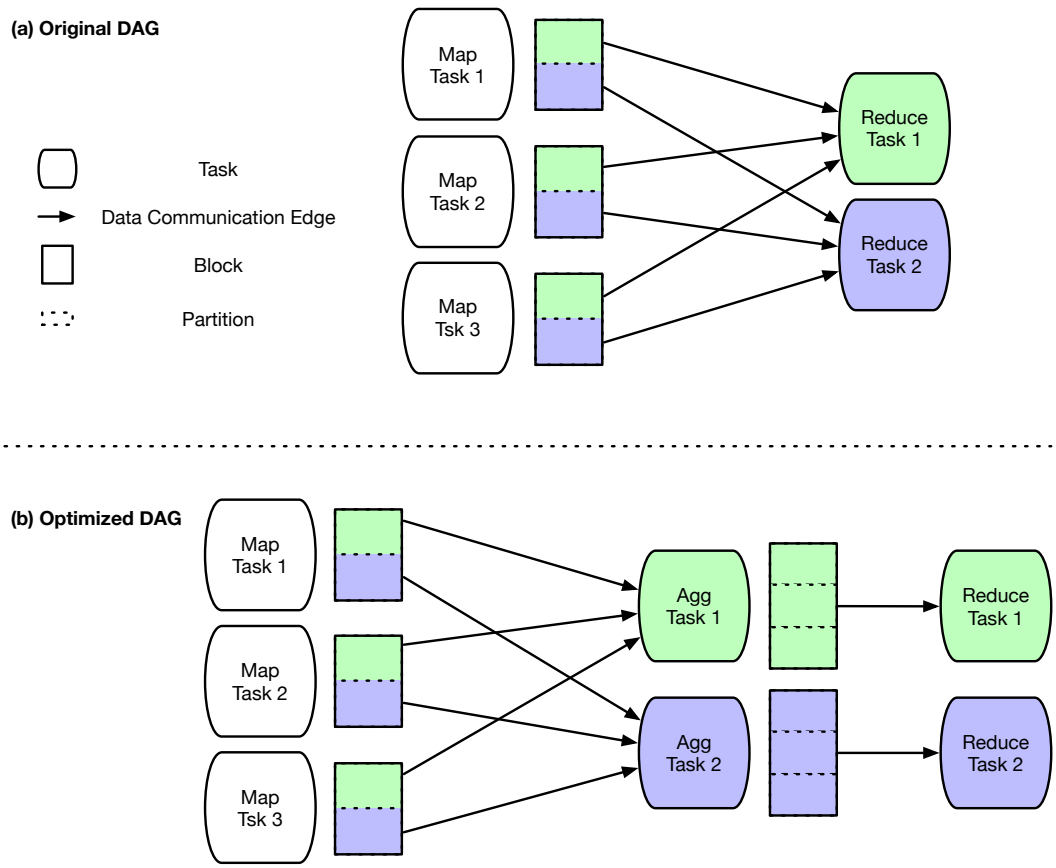

Figure 3.1: Insertion of the intermediate data aggregation stage into a simple map-reduce application. (a) shows the original task-level job DAG, and (b) represents the job DAG when the aggregation stage (Agg Task) is inserted. Each partition is consumed by tasks that have the same color.

aggregation task passes input data to the output, the intermediate data that each reduce task reads is identical to the original intermediate data in (a). As depicted, the reduce task can read the input sequentially, although the aggregation task still reads the shuffled data in a random order.

In Figure 3.1 (b), the map tasks can push the shuffled data immediately to the aggregation tasks using memory instead of disks, and the aggregation tasks can store the received data on disks. Then, each output block of the aggregation tasks becomes a data group for a particular reduce task stored on a disk as a file that can be read sequentially. This makes many random read requests occur in memory instead of in disks. If there are P producer tasks and C consumer tasks connected with a shuffle edge in the original DAG, the number of random disk reads decreases from $P \times C$ to C through the disk read batching. Accordingly, the total number of random disk reads becomes linearly proportional to the input data size, and the influence of the random read remarkably decreases.

This aggregation methodology is similar to the intermediate data aggregation solutions mentioned above, but does not require an external process or file system for aggregation. Moreover, the inserted stage and data communication follow the ordinary job execution path of the data processing system. Because of this, the disk read batching step naturally exploits the functionalities of the data processing system, such as computation scheduling, parallelization, and fault recovery. Furthermore, the new stage and edge included in the job DAG can be further optimized by the system in the same way that the system modifies other ordinary job DAGs. For example, the skew handling optimization discussed in Section 3.2 can be applied to the DAG that has the inserted aggregation stage. Section 4.1 also describes several detailed tunings for the data transfer around the aggregation stage.

3.2 Composing with Skew Handling Optimization

As described in Section 2.4, existing skew handling optimizations are not compatible with the disk read batching optimization. In order to apply skew handling and disk read batching

at the same time, we propose a skew handling optimization that is a hybrid of the input data sampling method and the dynamic redistribution method. Instead of running a job for the sampled input data and collecting a portion of the target intermediate data before executing the original job for the full input data, the data processing system can generate a portion of the intermediate data to optimize the distribution dynamically during the job execution.

When a job DAG consists of three stages S_1, S_2, and S_3 and two edges, E_1 connecting S_1 with S_2 and E_2 connecting S_2 with S_3, a portion of the output data of S_2 can be generated by executing some sampled tasks of S_2 with the intermediate data from S_1 at run-time. After the execution of S_1 is completed, the sampled tasks of S_2 can consume the intermediate data just as the original tasks of S_2 consumes it, regardless of the communication pattern of E_1. Therefore, by sampling the tasks that produce the target intermediate data instead of the input data, the data processing system can acquire the accurate data size metric of the intermediate data. After the execution of the sampled tasks, the system can reconfigure the distribution of E_2 according to the gathered metric and execute S_2. As a result, the tasks in S_3 can consume a similar amount of data. Section 4.2 presents a detailed implementation of this skew handling optimization.

Because this optimization collects the data size metric of the target data before running the original producer tasks, the distribution of the intermediate data is not changed once it is stored, unlike the other solutions classified in the dynamic redistribution method. In addition, because the data metric collection is conducted dynamically and the earlier stages before generating the target data are executed normally, the data metric is reliable regardless of the position of the target edge in the job DAG, unlike other solutions classified in the input data sampling method. Therefore, this skew handling optimization is fully compatible with the disk read batching optimization for arbitrary job DAGs.

Indeed, existing disk read batching solutions that conduct intermediate data aggregation on an external component are compatible with this skew handling mechanism. However, the external component must have an interface that enables the data processing system to

reconfigure the intermediate data distribution at run-time. Because the disk read batching described in Section 3.1 includes the intermediate data aggregation in the job DAG as a stage, the DAG with an aggregation stage can be optimized through this skew handling method just like any other job DAGs.

Chapter 4

Implementation

The disk read batching and skew handling are implemented on Apache Nemo [5], which is a distributed data processing system with a flexible optimizer. When an application is executed in Nemo, the Nemo optimizer parses the application into an intermediate representation (IR) DAG, which is an abstracted DAG representing the job. The IR DAG has annotations on each IR vertex and IR edge called execution properties that designate the job execution, such as the partitioning policy and communication pattern. The Nemo optimizer optimizes jobs by modifying the annotations in the IR DAG or reshaping the IR DAG itself. In Nemo, a series of reshaping or modifying annotations is represented as an optimization pass. The disk read batching and skew handling are also implemented as optimization passes.

4.1 Optimization Pass for Disk Read Batching

The implemented optimization pass for disk read batching inserts the aggregation vertex into the IR DAG as described earlier. Figure 4.1 illustrates how the IR DAG of a word-counting job is changed by the optimization pass. In (a), the map vertex counts the number of occurrences per word in each source block from the source vertex and the reduce task calculates the total number of occurrences per word. The source and map vertex will be pipelined as a stage. In (b), the optimization pass reshapes the IR DAG by inserting an aggregation vertex that has the same parallelism with the reduce vertex, and connects the aggregation vertex with the reduce vertex. The aggregation tasks of the inserted aggregation vertex relays the

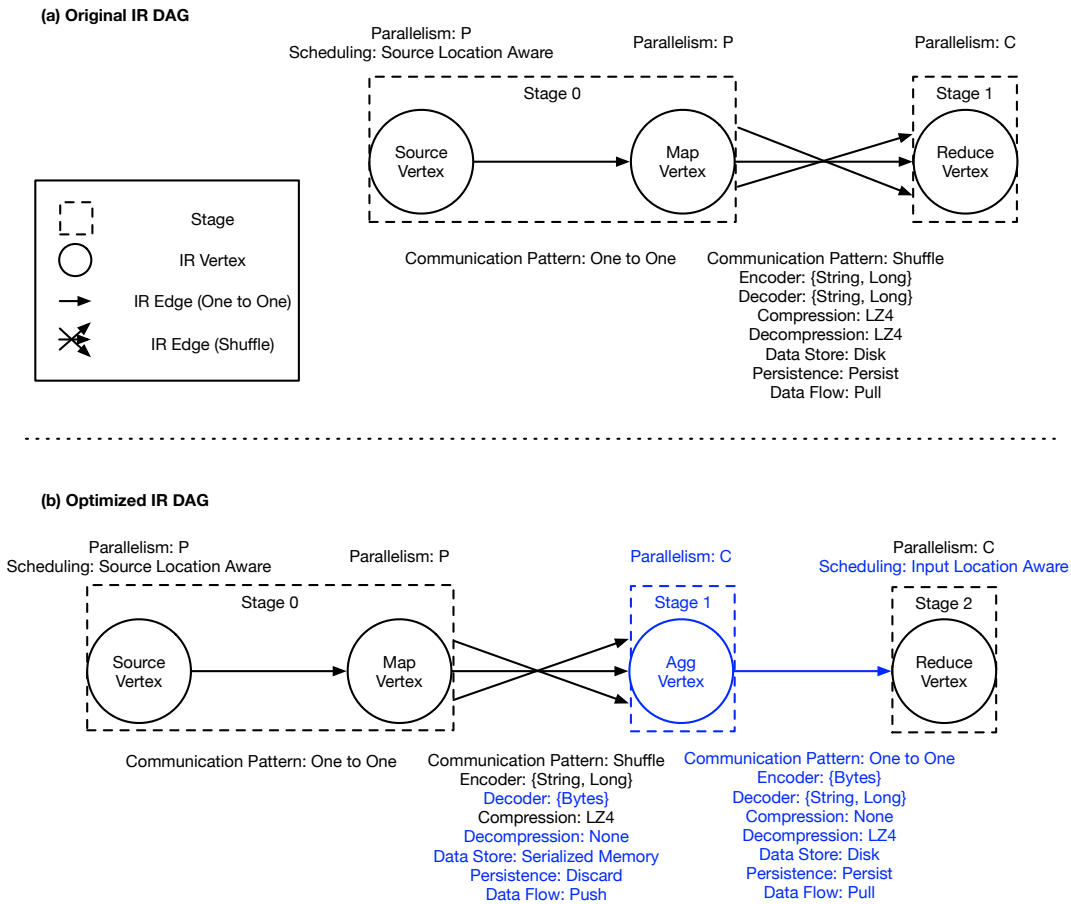

Figure 4.1: Optimization pass for disk read batching that inserts an aggregation vertex (represented as Agg Vertex) for the target shuffle edge. (a) shows the original IR DAG and the execution properties of a simple word counting job, and (b) represents the IR DAG and the execution properties after the optimization pass is applied. Only properties related to the optimization are presented. The inserted vertex and the modified execution properties are marked in blue.

input data to the output. The implemented pass always inserts an aggregation vertex before every shuffle edge, but the pass can conditionally insert the aggregation vertex depending on the parallelism of the consumer vertex.

After the reshaping, the optimization pass modifies the annotation of the IR DAG. By modifying the data store property of the shuffle from the disk to the serialized memory, the shuffled data is stored and read in the memory in a serialized format without disk access. In addition, by setting the data flow method property of the shuffle edge as push and setting the persistence as discard, the map tasks immediately push the shuffled data to the aggregation tasks and discard it after the transfer. The output of the aggregation task is kept in a disk and the reduce vertex is not pipelined with the aggregation vertex.

By these reshaping and annotating, the disk read can be batched. However, there are some additional tunings that can be applied to improve performance. First, because the aggregation task does not see the content of the input data, the input data does not need to be deserialized. Therefore, by modifying the encoder, decoder, compression, and decompression properties as in (b), the aggregation task can take arrays of partition bytes as the input and write the bytes to the output directly without any additional (de)serialization, while the data that the reduce vertex reads is unchanged. Second, the scheduling of the reduce tasks can be optimized by letting the system schedule each reduce task to the executor that contains the input data. This ensures that the data read from the disk is not transferred through the network. Finally, when a vertex receives multiple shuffle edges, the optimization pass merges the aggregation vertices for the shuffle edges into one aggregation vertex.

4.2 Optimization Pass for Skew Handling

The implemented skew handling optimization pass samples the stage that produces the target intermediate data as explained earlier. Figure 4.2 depicts how the IR DAG of an example job with 3 stages is changed by the optimization pass at compile-time. The optimization pass tries to make each task of Stage 2 in (a) consume a similar amount of data. In order to

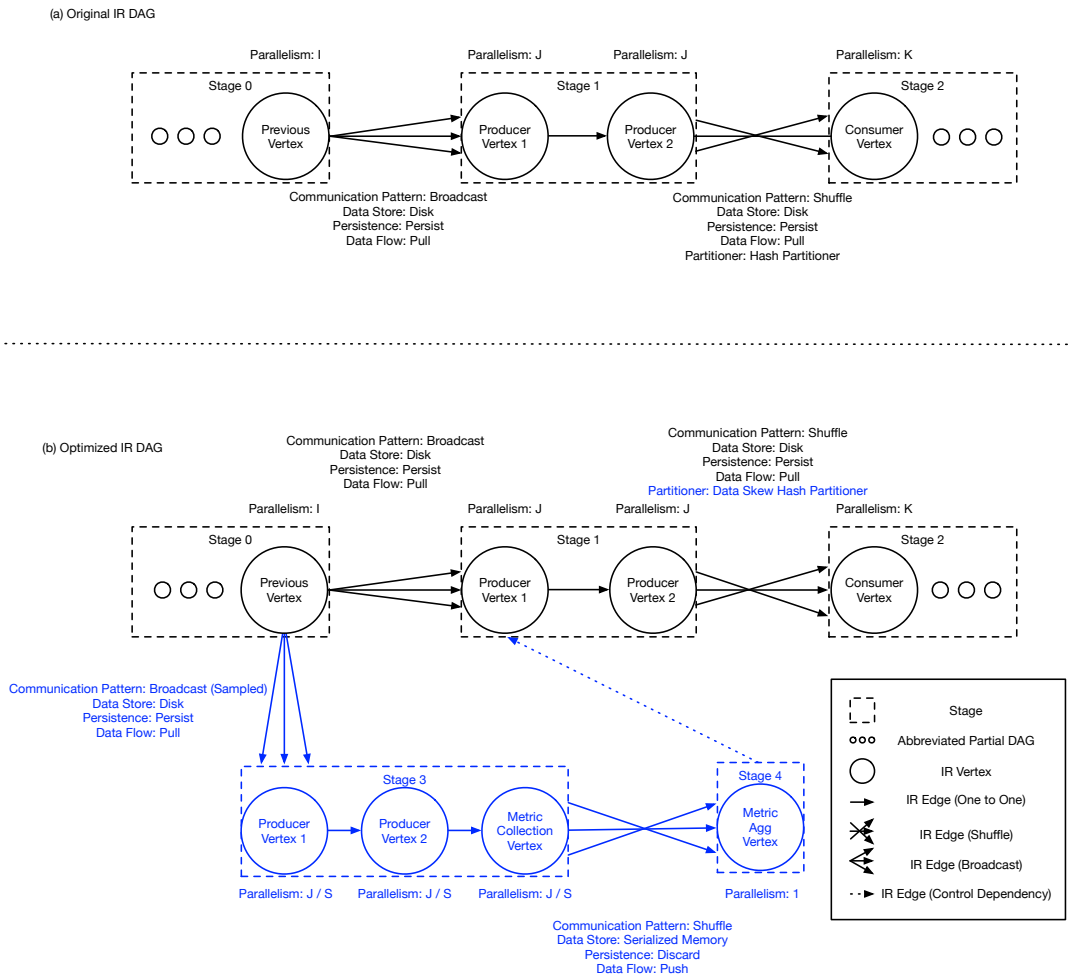

Figure 4.2: Skew handling optimization pass that inserts a sampling stage and a metric aggregation vertex (represented as Metric Agg Vertex) for the target shuffle edge. (a) shows the original IR DAG and the execution properties of an example job with the target shuffle edge, and (b) represents the IR DAG and the execution properties after the optimization pass is applied. Only properties related to the optimization are presented. The inserted vertices and the modified execution properties are marked in blue.

achieve the data size metric per key for the intermediate data, the pass samples tasks in the Stage 1, inserts a metric collection vertex at the end of the sampled tasks, inserts a metric aggregation vertex after the sampled stage, and creates a control dependency between the metric aggregation vertex and the original Stage 1 to allow Stage 1 to run after the metric aggregation. When the sampling rate is $\frac{1}{S}$ and the parallelism of the original Stage is J, the pass sets the parallelism of the vertices of the sampled Stage 3 in (b) as $\frac{J}{S}$. The inserted metric collection vertex groups the input data from the Producer Vertex 2 per key and calculates the size of the data of each group. The inserted metric aggregation vertex aggregates the collected metric by summing the data size per key. To perform the metric aggregation concurrently with the metric collection, the output data of the metric collection vertex is pushed through memory and discarded. When a vertex receives multiple shuffle edges, a metric aggregation vertex collects all metrics for the receiving vertex and the reconfigured data distribution is applied for the shuffle edges identically for correctness.

Because an optimization pass for skew handling in a dynamic redistribution style already exists in Nemo, many concepts of this pass, such as the custom hash partitioner, the dynamic reconfiguration mechanism, the skewed task aware scheduling, and the run-time optimization pass that contains the shuffled data redistribution algorithm, are reused for the new skew handling optimization pass. The custom hash partitioner marked as ”Data Skew Hash Partitioner” increases the key range of the partitions several times. When a task of the metric aggregation stage aggregates the data size metric at run-time, the task sends the metric to the Nemo driver. By using this metric, the run-time optimization pass coupled to the compile-time optimization pass calculates a better shuffled data distribution and reconfigures the shuffle edge. After the reconfiguration, the metric aggregation vertex is marked as complete and the control dependency is resolved.

As we mentioned above, this optimization pass can be applied to an arbitrary DAG including the DAG optimized by the disk read batching optimization pass. Although the sampled stage of the example only receives a single broadcast edge, the stages that receive many

edges of the arbitrary communication pattern can be sampled. When the compile-time optimization pass samples the target stage, an index of the task in the original stage is assigned to each task of the sampled stage. The sampled task reads the data exactly the same as the data that is read by the original task of the assigned index as follows.

- When the original task has a source vertex, the sampled task reads the source block of the designated index.
- If the original task receives a one-to-one edge, the sampled task reads the output data from the producer task of the one-to-one edge with the index.
- Since all the tasks in the stage receiving a broadcast edge read the same broadcasted data, the sampled task reads the broadcast data in the same way as the original tasks.
- When the original task receives a shuffle edge, the sampled task reads the partitions of the key range assigned to the original task with the index.

Chapter 5

Evaluation

In this chapter, we evaluate our implementation of the optimization to answer the following questions:

- Does the disk read batching optimization effectively reduce the random disk read overhead?
- Is the skew handling optimization compatible with the disk read batching optimization?

5.1 Cluster Setup

We composed an evaluation cluster with AWS EC2 h1.4xlarge instances in a single placement group. The h1.4xlarge instances have 16 virtual cores, 64 gigabytes (GB) of memory and two 2TB HDDs, and are connected to a 10 Gbps network. One instance was used as a driver and others were used as workers. A Hadoop distributed file system (HDFS) [13] that occupied one 2TB HDD of each worker instance was launched to store the job input data and output data. These instances also configure the YARN [14] cluster. The rest of the HDDs are used as the local storage of the YARN cluster and store the intermediate data at run-time. To see the performance of Nemo, we executed the word-counting job used for Section 5.2 on Spark [2] 2.3.0 and confirmed that the JCT on Spark (102 minutes on average) is comparable for the JCT of the default execution on Nemo. Each experiment is repeated 5 times.

5.2 Disk Read Batching Optimization

To see the effectiveness of the disk read batching optimization, we evaluated the job completion time (JCT) and disk throughput of the word-counting job described in Section 4.1. The application is programmed in Apache Beam [15] and executed on Nemo. A 1 TB dataset from the Wikipedia 2016 page count dataset [16] is used as the input data, and the job counts the view count of each page in a specific period. 11 instances were used for this experiment.

Figure 5.1 represents the JCT of the word-counting job and Figure 5.2 represents the disk input output (I/O) throughput during the word-counting job execution. Default is the case in which the job runs with the default optimization passes of Nemo but without the disk read batching optimization. Optimized is the case in which the job runs with the default optimization passes and the disk read batching optimization together. For the optimized case in Figure 5.1, the execution time of the original producer stage and the aggregation stage belong to the shuffled data production time.

The JCT results show that the disk read batching causes a slight increase in production time, but a significant reduction in consumption time. The JCT is decreased by 54% on average. The disk I/O throughput results also show that the disk read batching slightly slows down the intermediate data write but largely accelerates the read. The slow disk read throughput of the default case indicates that the consumer stage suffers from the random disk read overhead. The JCT and disk I/O throughput results show that the insertion of the aggregation stage causes the modest increase of the production time, but the time needed to read the shuffle data is significantly decreased because the consumer tasks can read the data sequentially.

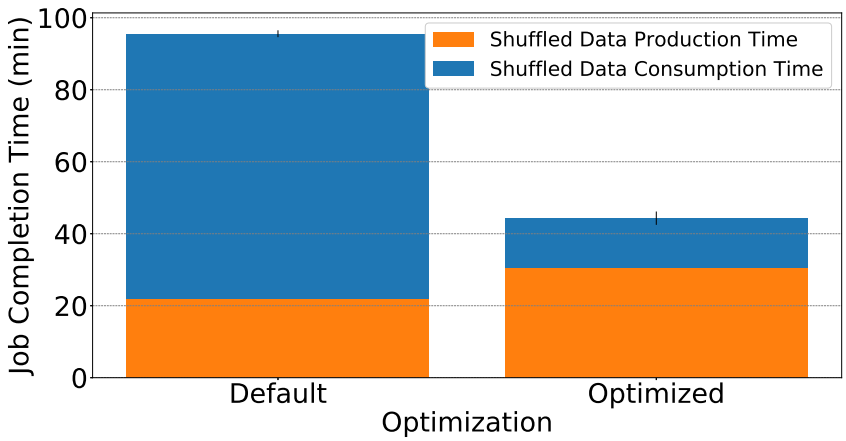

Figure 5.1: Job completion time of the word-counting job with and without the disk read batching optimization. Error bar represents the average standard deviation of JCT.

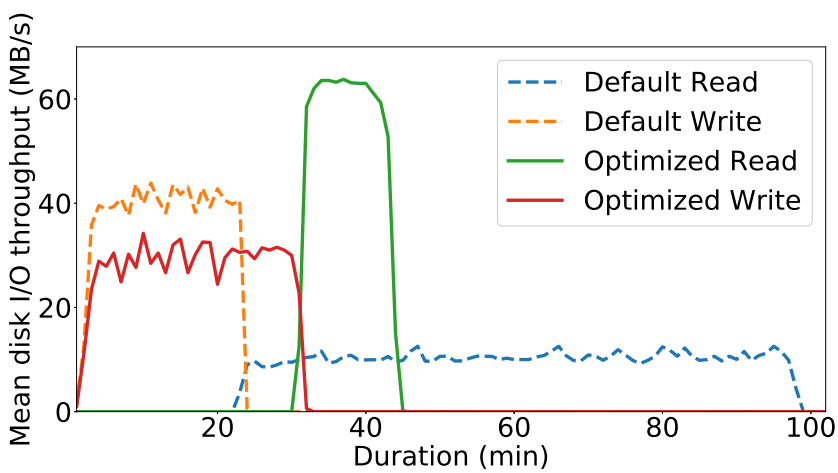

Figure 5.2: Mean disk I/O throughput during a single run of the word-counting job with and without the disk read batching optimization. The measured disks are the YARN local disks for maintaining intermediate data.

5.3 Skew Handling Optimization with Disk Batching

To evaluate whether the skew handling optimization alleviates the data skew and is compatible with the disk read batching for a complicated DAG, we ran the TPC-H [17] query 10, which is a decision support SQL query. The application is programmed by using the Beam SQL library [18] and is executed on Nemo. To introduce data skew, we generated a 100 GB dataset by using the skewed TPC-H dataset generator [19] introduced in [20]. We selected query 10 because the query produces various degree of data skew. The job has 8 shuffle edges and 9 stages as a default, and the optimization pass optimizes all the shuffle edges. 4 instances were used for this experiment.

Because the intermediate data size metric is collected from the sampled tasks only, the performance of skew handling largely relies on the quality of sampling. To confirm that the sampled intermediate data can represent the total intermediate data, we evaluated the intermediate data size metric with various sampling rates before the skew handling experiment. Figure 5.3 shows the results. The ratio of the input intermediate data size per consumer task is presented instead of actual size because the size is varied by the sampling rate. The result shows that the trend of data sizes is detected in low sampling rates (1%, 5%), and the actual distribution is calculated in higher sampling rates. We decided to sample the 5% of tasks in producer stage for each shuffle edge in the skew handling experiment to reduce the sampling overhead.

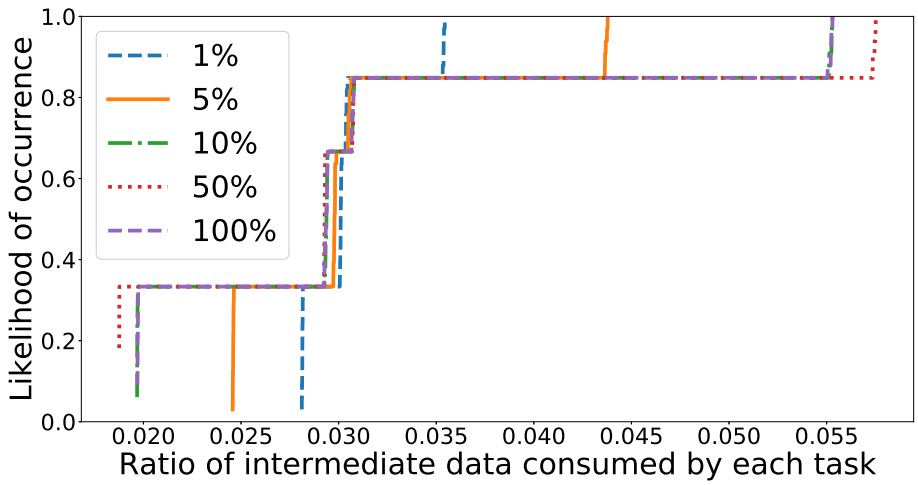

Figure 5.3: Cumulative distribution function of the ratio of the input intermediate data size per consumer task of a join vertex in a single run depending on sampling rate without skew handling.

Figure 5.4 represents the JCT of the TPC-H query and Figure 5.5 represents the sorted size of the input intermediate data per consumer task. Default is the case in which the job runs with the default optimization passes of Nemo. Optimized is the case in which the job runs with the default optimization passes, the disk read batching optimization, and the skew handling optimization together. In Figure 5.5, the input intermediate data size is collected for the vertex that joins ”orders” table and ”customer” table, which consumes various size of input data.

The JCT results show that the skew handling optimization with the disk read batching reduces the JCT by 31 % on average. The high standard deviation of JCT in default is due to the skew unaware scheduling. The data size results also show that the skew handling optimization mitigates the skew of the intermediate data size that each consumer task reads. These results show that the data skew handling optimization handles the data skew well by dynamically sampling stages and reconfiguring the data distribution and it is compatible with the disk read batching.

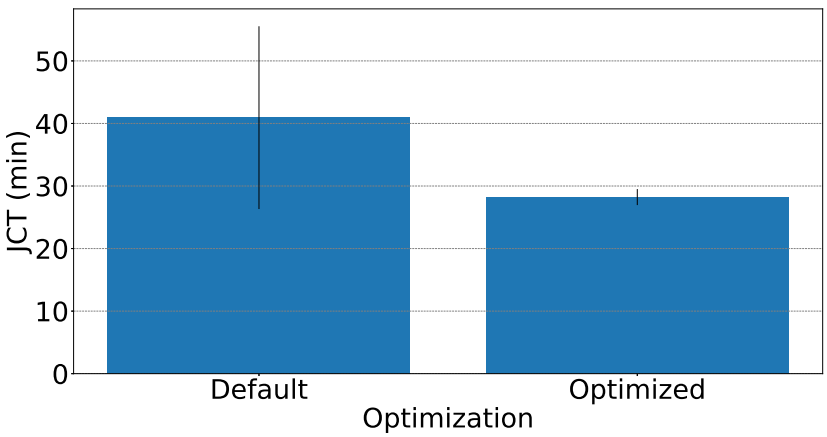

Figure 5.4: Job completion time of the TPC-H query with and without the skew handling and disk read batching optimization. Error bar represents the average standard deviation of JCT.

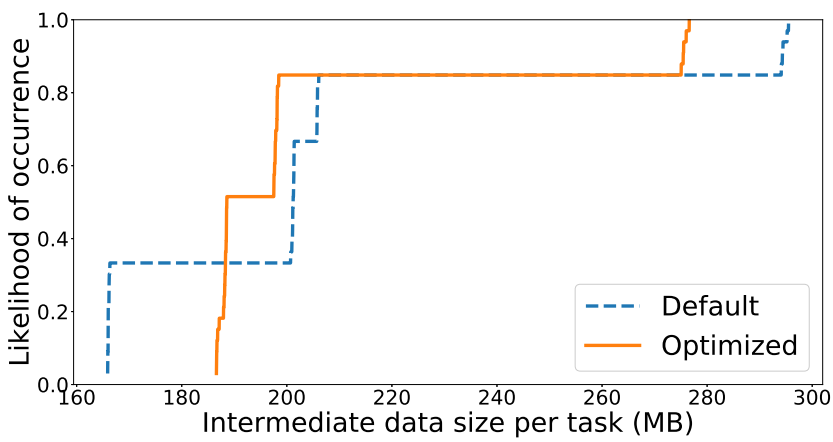

Figure 5.5: Cumulative distribution function of the input intermediate data size per consumer task of a join vertex in a single run with and without the skew handling and disk read batching optimization.

Chapter 6

Conclusion

As the scale of data analysis is growing rapidly, the random disk read overhead that occurs during a large-scale shuffle becomes a scaling bottleneck. Existing systems deal with this problem by launching a separate intermediate data aggregation service or file system. However, this approach requires a separate computation scheduling, parallelization, and fault recovery mechanism and makes it hard to apply further optimization, such as data skew handling, to the shuffle.

To overcome this limitation, we propose to insert the disk read batching computation into the job. By doing so, the inserted tasks can fully exploit the features provided by the data processing system and the system can apply other optimizations transparently. As an example of a further optimization, we suggest a skew handling optimization technique compatible with our disk read batching optimization.

By evaluating these techniques, we show that our disk read batching optimization can largely mitigate the random disk read overhead while keeping the flexibility of the job execution and optimization. We hope that the design of our disk read batching optimization will help other silo-versions of optimization techniques to be revealed and combined with other optimizations.

Bibliography

[1] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman, "Riffle: optimized shuffle service for large-scale data analytics," in *Proceedings of the Thirteenth EuroSys Conference*, p. 43, ACM, 2018.

[2] "Spark." `http://spark.apache.org`.

[3] "Apache Hadoop." `http://hadoop.apache.org`.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007.

[5] "Nemo." `http://nemo.apache.org`.

[6] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *SOCC*, 2012.

[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[8] A. Rasmussen, M. Conley, G. Porter, R. Kapoor, A. Vahdat, *et al.*, "Themis: an i/o-efficient mapreduce," in *Proceedings of the Third ACM Symposium on Cloud Computing*, p. 13, ACM, 2012.

[9] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 75–86, ACM, 2010.

[10] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: a self-tuning system for big data analytics.," in *Cidr*, vol. 11, pp. 261–272, 2011.

[11] Q. Ke, M. Isard, and Y. Yu, "Optimus: A dynamic rewriting framework for data-parallel execution plans," in *EuroSys*, 2013.

[12] L. Bindschaedler, J. Malicevic, N. Schiper, A. Goel, and W. Zwaenepoel, "Rock you like a hurricane: taming skew in large scale analytics," tech. rep., 2018.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pp. 1–10, Ieee, 2010.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013.

[15] "Apache Beam." `https://beam.apache.org/`.

[16] "Wikipedia Pagecounts." `https://wikitech.wikimedia.org/wiki/Analytics/Archive/Data/Pagecounts-raw`.

[17] "TPC-H: a Decision Support Benchmark." `http://www.tpc.org/tpch/`.

[18] "Apache Beam SQL Library." `https://beam.apache.org/documentation/dsls/sql/walkthrough/`.

[19] `https://github.com/ldbc/dbgen.JCC-H`.

[20] P. Boncz, A.-C. Anatiotis, and S. Kläbe, "Jcc-h: Adding join crossing correlations with skew to tpc-h," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 103–119, Springer, 2017.

국문초록

오늘날 데이터 분석 작업에서 사용하는 데이터의 크기가 빠르게 커지고 있으며, 이 때문에 데이터 처리 시스템은 대용량의 데이터를 효율적으로 처리할 수 있어야 한다. 분산 데이터 처리 시스템에서 큰 데이터를 처리할 때의 병목은 태스크 간 데이터 셔플시 발생하는 랜덤 디스크 읽기 비용이다. 이 비용을 줄이기 위하여, 외부 셔플 프로세스가 데이터 처리 시스템 바깥에서 추가적인 계산을 통해 중간 데이터를 병합하여 디스크 읽기를 일괄 처리하도록 할 수 있다. 그러나, 이 경우 추가된 계산은 기존에 데이터 처리 시스템이 제공하는 계산 스케쥴링, 병렬화, 실패 복구 등의 기능을 이용할 수 없다. 또한, 데이터 처리 시스템이 다른 일반적인 작업을 최적화하는 것처럼 이 외부 셔플 프로세스의 동작을 최적화할 수 없다. 이 문제를 해결하기 위하여, 본 논문에서는 디스크 읽기를 일괄 처리하도록 만드는 계산을 작업 수행 내부에 끼워넣는 방식을 고안하였다. 중간 데이터 병합을 위한 계산을 태스크로서 작업에 끼워넣어 데이터 처리 시스템이 이 태스크를 수행하도록 하면 이 태스크들은 동적 최적화를 포함하여 데이터 처리 시스템이 제공하는 모든 기능들을 사용할 수 있다. 또한, 본 논문에서는 이러한 중간 데이터 병합과 호환되는 데이터 치우침 처리 방식을 제안한다. 수행된 실험의 결과를 통해 구현된 최적화가 랜덤 디스크 읽기 비용을 줄이고 데이터 치우침을 완화하여 최대 54%의 성능 향상을 보임을 확인할 수 있다.

주요어: 데이터 처리, 데이터 분석, 대규모 셔플, 저장소 입출력 최적화, 동적 최적화, 데이터 치우침 처리, 데이터 처리 시스템

학번: 2017-28856