Building High Performance Data Analytics Systems based on Scale-out Models

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Yin Huai, B.E.

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Dissertation Committee:

Xiaodong Zhang, Advisor
Feng Qin
Spyros Blanas
Abstract

To respond to the data explosion, new system infrastructures have been built based on scale-out models for the purposes of high data availability and reliable large-scale computations. With an increasing amount of adoptions of data analytics systems, users continuously demand high throughput and high performance on various applications. In this dissertation, we identify three critical issues to achieve high throughput and high performance for data analytics, which are efficient table placement methods (i.e. the method to place structured data), generating high quality distributed query plans without unnecessary data movements, and effective support of out-of-band communications. To address these three issues, we have conducted a comprehensive study on design choices of different table placement methods, designed and implemented two optimizations to remove unnecessary data movements in distributed query plans, and introduced a system facility called SideWalk to facilitate the implementation of out-of-band communications.

In our first work of table placement methods, we comprehensively studied existing table placement methods and generalized the basic structure of table placement methods. Based on the basic structure, we conducted a comprehensive evaluation of different design choices of table placement methods on I/O performance. Based on our evaluation and analysis, we provided a set of guidelines for users and developers to tune their implementations of table placement method.
In our second work, we focused on building our optimizations based on Apache Hive, a widely used open source data warehousing system in the Hadoop ecosystem. We analyze operators that may require data movements in the context of the entire query plan. Our optimization methods remove unnecessary data movements from the distributed query plans. Our evaluation shows that these optimization methods can significantly reduce the query execution time.

In our third work, we designed and implemented SideWalk, a system facility to implement out-of-band communications. We designed the APIs of SideWalk based on our abstraction of out-of-band communications. With SideWalk, users can implement out-of-band communications in various applications instead of using ad-hoc approaches. Through our evaluation, we show that SideWalk can effectively support out-of-band communications, which will be used in implementing advanced data processing flows, and users can conduct out-of-band communications in a reusable way. Without SideWalk, users commonly need to build out-of-band communications in an ad hoc way, which is hard to reuse and limit the programming productivity.

The proposed studies in this dissertation has been comprehensively tested and evaluated to show their effectiveness. The guidelines on table placement methods in our table placement method study has been verified by newly implemented and widely used file formats, Optimized Record Columnar File (ORCFile) and Parquet. Optimization methods in our query planner work have been adopted by Apache Hive, which is a widely used data warehousing system in the Hadoop ecosystem and is shipped with all of major Hadoop vendors.
To my wife, Xing Xing, and my family.
Acknowledgments

I would like to extend my great appreciation and respect to my advisor, Xiaodong Zhang. Six years ago, he extended me the offer to study at The Ohio State University, an opportunity to work in one of the top Computer Science labs in the United States. During the years, he has continuously provided guidance to my research and my career. He has granted me with the freedom and the resources to research on topics in which I am most interested. He has provided me with valuable inputs to my research and challenged me to achieve more than I thought I could ever achieve. It has been a great honor to work with him.

The work of this dissertation is a product of several collaborations. I am thankful to my fellow labmates, Yuan Yuan, Kaibo Wang, Tian Luo, Siyuan Ma, Lei Guo, Xiaoning Ding, Feng Chen, and Rubao Li. The thought-inspiring environment shaped by this group is the crib to several excellent results in this dissertation. I want to thank especially Rubao Li, who is an experienced and talented researcher himself. He has been providing me with some of the most critical inputs that helped to complete the tasks. I am grateful to his mentorship throughout my PhD. I also want to thank Fusheng Wang who introduced the application of Pathology Image Data Cross-comparison to our lab, which inspired Kaibo Wang and I to start our work on using GPU to first on this specific applications [72] and later, to expand our GPU research efforts in a much larger scope.
Beyond my co-workers at the lab, I also had the privilege to work with people from the industrial world. Ashutosh Chauhan gave me the opportunity to work on Apache Hive at Hortonworks. Ashutosh Chauhan and Gunther Hagleitner spent a tremendous amount of time helping me design and implement optimizations introduced in Chapter 3 based on the project of YSmart [60] from our lab. I still remember the moment of joy after Ashutosh Chauhan committed my code into the codebase of Apache Hive. Owen OMalley and I had several discussions on table placement methods and he shared lots of experiences on building ORC File. These discussions and his experiences are very helpful to my work on table placement methods [55]. Eric N. Hanson provided lots of valuable comments to me during my preparation of the paper about Apache Hive’s recent major advancements [54]. Reynold Xin and Michael Armbrust got me started on contributing to Apache Spark and Spark SQL. I always enjoy having discussions with them on various topics from query optimizations to API design. I also want to thank Ali Ghodsi and Ion Stoica who gave me valuable advices on computer science research and career development. Moreover, I am thankful to Eric Baldeschwieler, who provided several of constructive comments to my work on query optimizations and handling semi-structured data. I want to thank the user and development community of Apache Hive and Apache Spark. I have learned a lot on what are critical issues in the daily operations of data analytics from users and other developers. This dissertation has been partially supported by the National Science Foundation under grants CCF-0913050, OCI-1147522, and CNS-1162165.

Last but not least, I would like to thank my familys support through my PhD. I want to especially thank my wife, Xing Xing. Without her support, I cannot complete my graduate work.
Vita

2005.7 ................................. Beijing National Day School, Beijing, China
2009.7 ................................. B.E. Automation, Tsinghua University, Beijing, China
2012.6-2012.9 ............................... Research Summer Intern, IBM T.J. Watson Research Center, Hawthorne, NY, USA
2013.5-2013.8 ............................... Software Engineer Intern, Hortonworks, Inc., Palo Alto, CA, USA
2014.5-2014.8 ............................... Software Engineering Intern, Databricks, Inc., Berkeley, CA, USA
2009.9-present ................................. Graduate Research Associate, Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
2015.1-present ................................. Software Engineer, Databricks, Inc., San Francisco, CA, USA

Publications

Research Publications


Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problems with Existing Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.1.1 Inefficiency of Storing Structured Data Organized in Tables</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 Inefficiency of Compiled Distributed Query Plans</td>
<td>4</td>
</tr>
<tr>
<td>1.1.3 Lack of System Facilities for Out-of-band Communications</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Summary of Results</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Dissertation Outline</td>
<td>7</td>
</tr>
<tr>
<td>2. Basic Structure and Essential Issues of Table Placement Methods</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Table Placement Method</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1 Definition</td>
<td>11</td>
</tr>
<tr>
<td>2.2.2 Existing Work</td>
<td>15</td>
</tr>
<tr>
<td>2.2.3 A Unified Evaluation Framework</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Study Methodology</td>
<td>19</td>
</tr>
<tr>
<td>2.4 Micro-benchmark Results</td>
<td>22</td>
</tr>
<tr>
<td>2.4.1 Rationality and Objectives</td>
<td>22</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A summary of symbols used in Table 2.2</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Definitions of six table placement methods.</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>A summary of conventions.</td>
<td>29</td>
</tr>
<tr>
<td>2.4</td>
<td>A summary of datasets.</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>The size of records from lineitem emitted for shuffling (bytes).</td>
<td>93</td>
</tr>
<tr>
<td>4.2</td>
<td>Three scales used in the experiment of running the query shown in Fig. 4.9</td>
<td>97</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The software stack of data analytics systems.</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>A demonstration of a table placement method.</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>The entire process of accessing a table at run time.</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Results of experiments on changing row group sizes.</td>
<td>30</td>
</tr>
<tr>
<td>2.4</td>
<td>Results of experiments on changing row group sizes.</td>
<td>31</td>
</tr>
<tr>
<td>2.5</td>
<td>Results of experiments on different column grouping schemes.</td>
<td>33</td>
</tr>
<tr>
<td>2.6</td>
<td>Results of experiments on different column grouping schemes.</td>
<td>34</td>
</tr>
<tr>
<td>2.7</td>
<td>Results of experiments on different data packing schemes.</td>
<td>35</td>
</tr>
<tr>
<td>2.8</td>
<td>Results of experiments on different data packing schemes.</td>
<td>36</td>
</tr>
<tr>
<td>2.9</td>
<td>Results of macro-benchmarks using Hive.</td>
<td>41</td>
</tr>
<tr>
<td>2.10</td>
<td>The change of data reading efficiency.</td>
<td>46</td>
</tr>
<tr>
<td>3.1</td>
<td>The architecture of Hive.</td>
<td>53</td>
</tr>
<tr>
<td>3.2</td>
<td>The running example used in Section 3.3.</td>
<td>57</td>
</tr>
<tr>
<td>3.3</td>
<td>The optimized operator tree of the running example shown in Figure 3.2.</td>
<td>61</td>
</tr>
<tr>
<td>3.4</td>
<td>Elapsed times of TPC-DS query 27 and query 95.</td>
<td>64</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.1</td>
<td>An example query to evaluate a rank function.</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>An example query evaluation plan used in Section 4.1.</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>The Abstraction of Out-of-band Communications</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>An example query used in Section 4.4.1</td>
<td>80</td>
</tr>
<tr>
<td>4.5</td>
<td>The optimized query plan for the query shown in Fig. 4.4</td>
<td>82</td>
</tr>
<tr>
<td>4.6</td>
<td>The plan shown in Fig. 4.5 with debugging support.</td>
<td>85</td>
</tr>
<tr>
<td>4.7</td>
<td>The software architecture of SideWalk.</td>
<td>87</td>
</tr>
<tr>
<td>4.8</td>
<td>The query execution times for three queries.</td>
<td>93</td>
</tr>
<tr>
<td>4.9</td>
<td>The SQL query with window functions</td>
<td>95</td>
</tr>
<tr>
<td>4.10</td>
<td>The query plan.</td>
<td>96</td>
</tr>
<tr>
<td>4.11</td>
<td>Query execution times for implementations of out-of-band communications</td>
<td>98</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Since the last decade, the appearance of web-scale applications has been driving a shift of the paradigm that we store, manage, and interact with various datasets. In this shift of the paradigm, experiences from companies and organizations have shown that existing and traditional data analytical technologies and business models cannot sustain the explosion of data. In the high performance computing field, applications’ performance is mainly improved by increasing the memory capacity and CPU computing power in a single or tightly-coupled platform. This type of system scaling method is called “scale-up”, or vertical scalability. For large scale data processing, computing and storage nodes need to be continuously added to improve the throughput on the increasing size of the data. This type of system scaling method is called “scale-out”, or horizontal scaling. To cope with the continuous explosion of data in a cost-efficient way, scale-out storage and computing models, such as Google File System [48] and Google MapReduce [42], were introduced. Nowadays, scale-out storage and computing models are the foundation to deal with high volumes of data and highly diverse data processing demands.

Figure 1.1 shows the software stack of data analytical systems based on scale-out models. At the bottom of the stack, a distributed file system stores data in a shared-nothing cluster. This file system automatically partition files to blocks and replicate blocks to provide fault-tolerance. Hadoop Distributed File System (HDFS) [1] is a widely used distributed
file system. On top of the distributed file system, there are data processing engines, such as Hadoop MapReduce [2], Tez [3], and Spark [4], that execute distributed programs in a cluster of computing and storage nodes. These data processing engines provide their abstractions of programming for a distributed system and hide complexities of distributing user programs, scheduling user programs, executing user programs, failure recovery, and resource management. With these engines, users can easily write imperative programs to process datasets stored in the distributed file system. These programs will be executed in a fault-tolerant and scalable way by the data processing engine. Although existing data processing engines hide those complexities in distributed programming, further improving the programming productivity is still very important. Users demand to further hide
the complexities of writing imperative programs by using high level declarative programming languages like SQL to simply describes the data processing tasks they want and let the compiler automatically pick the efficient execution plans. With a high level declarative programming language, users will not need to optimize imperative programs and they can write much less lines of code for their data processing tasks, which is a significant improvement on programming productivity. To respond to this demand, systems, such as Hive [71] and Pig [46], were built to automatically compile high level declarative programming languages, such as SQL, to jobs executed by those data processing engines.

1.1 Problems with Existing Systems

Although the software stack for data analytical systems based on scale-out models has necessary components, systems in this stack are far from mature. We have identified critical problems in data analytical systems, which are the Inefficiency of Storing Structured Data Organized in Tables, the Inefficiency of Compiled Distributed Query Plans, and the Lack of System Facilities for Out-of-band Communications. In this section, we provide an overview of these problems that we have been working on.

1.1.1 Inefficiency of Storing Structured Data Organized in Tables

Processing structured data organized in Tables is one of the most important practices in data analytics systems. For a data analytical system, when a dataset is loaded into it, it needs to determine how to store this dataset physically, i.e. how to store a dataset in the bottom layer of the software stack shown in Figure 1.1. The way that a dataset is stored determines (1) the speed of loading this dataset, (2) the size of stored dataset, (3) the efficiency of accessing this dataset, and (4) the adaptivity of this stored dataset to various workload patterns. Unlike traditional parallel database systems, a data analytical system built based
on scale-out storage models usually does not have the control on how its underlying distributed file system places files to those physical machines. For structured data organized in tables, the table placement method determining how to organize data inside a dataset is usually the only component that can influence data placement decisions of the underlying distributed file system. When designing a suitable placement method, we need to consider all of above four issues as well as the characteristics of the underlying distributed file system. However, existing projects have been done independently, and a comprehensive and systematic study of table placement methods is absent. Without a comprehensive and systematic study, the basic structure of table placement methods is not defined and thus, general guidelines on how to adjust table placement methods to achieve optimized I/O read performance are not clear.

1.1.2 Inefficiency of Compiled Distributed Query Plans

For a user submitted query, it will be compiled by a query compiler in the Query Compilers layer in Figure 1.1 to executable jobs of an underlying data processing engine shown in the Data Processing Engines layer in Figure 1.1. The quality of compilation directly determines the quality of generated query plans. However, existing query compilers, such as Hive [71] [5], translate a specific operation in the query to a physical data operation or movement without analyzing if this data operation or movement is actually needed. With this query compilation approach, unnecessary data operations and movements are introduced to the query plan and the efficiency of the generated plan is significantly degraded. To generate efficient query plan, the compiler needs to analyze the submitted query and eliminate unnecessary data operations and movements from the distributed query plan.
Also, unlike traditional parallel database systems, a data analytical system based on scale-out models relies on an underlying data processing engine to execute the generated query plan. The compiler should carefully consider the characteristics of the data processing engine while compiling submitted queries to executable jobs.

1.1.3 Lack of System Facilities for Out-of-band Communications

For a data processing engine, it exposes a programming model to users. Users describe their computing tasks through this programming model. Operations defined by a programming model have data movements and local computation in an individual worker machine. The data movement formally defined by a programming model is also called in-band communications, such as synchronous data communications in the shuffling phase between a map phase and a reduce phase in MapReduce. Although users can conduct data movements and communications between worker machines through operations defined by the programming model, a special kind of communications that is not defined in existing programming models, but is often used in ad-hoc ways by developers, is called out-of-band communication, which involves the use of an alternative communication pathway to exchange messages. Despite the fact that out-of-band communications are commonly used, there is little attention on how to effectively and efficiently implement out-of-band communications in a systematic way. In practice, developers and users usually find that there is not enough system support on out-of-band communications, and they have to implement ad hoc solutions, which are hard to reuse, error-prone, and easy to introduce negative side effects on the scalability and fault-tolerance of the execution of distributed data processing flows. To address these issues, a system facility for out-of-band communications is needed.
So, user applications and query compilers (the top two layers of Figure 1.1) can build advanced data processing pipelines with the help of this system facility.

1.2 Summary of Results

We have conducted three projects to address problems mentioned above. To be able to store table data efficiently, we have conducted a comprehensive and systematic study on the basic structure and essential issues of table placement methods. This work comprehensively studied the process of storing tables in distributed filesystems. We generalize the basic structure of table placement methods as three consecutive procedures, which are row reordering, table partitioning, and data packing procedures. Based on the basic structure of table placement methods, we studied impacts the design choices of table placement methods on the performance of accessing data in a table. Then, we provided a set of guidelines on designing table placement methods.

Towards generating high quality distributed query plans, we focused on reducing unnecessary data movements, which is generally one of the main issues hurting the performance of the query execution. In our work, we analyze the query plan and carefully evaluate if an operation may need to move data actually need to move data in the context of the given query. Based on this idea, we developed two optimizations for Apache Hive, which is a widely used data warehousing system for Hadoop ecosystem. These optimizations can significantly reduce query execution time by removing unnecessary data movements.

Finally, to systematically support out-of-band communications, we have built a system facility, called SideWalk. SideWalk exposes a set of APIs to implement out-of-band communications in various use cases. SideWalk serves as the only common information exchange place for senders and receivers of an out-of-band communication. For the senders
of an out-of-band communication, they send initial contents for the communication by placing these contents in SideWalk. Then, based on users’ requirements, SideWalk generates the desired contents for the out-of-band communication from the initial contents. For the receivers of an out-of-band communication, they receive the desired contents by reading these contents placed in SideWalk. With SideWalk, users can easily conduct out-of-band communications and use out-of-band communications to implement advanced data processing flows.

1.3 Dissertation Outline

This dissertation is organized as follows. Chapter 2 introduces our study on the basic structure and essential issues of table placement methods. Chapter 3 introduces the two optimizations we have built to remove unnecessary data movements in distributed query plans. Chapter 4 introduces SideWalk, a system facility for out-of-band communications. Chapter 5 concludes three projects and discusses potential future work.
Chapter 2: Basic Structure and Essential Issues of Table Placement Methods

2.1 Introduction

Structured data analytics is a major Hadoop application, in the context of which various query execution systems and query translators have been designed and implemented, such as Hive [5], Pig [6], Impala [7], Cheetah [41], Shark [75], and YSmart [60]. In these systems and translators, a critical component is a table placement method that determines how the data values in a two-dimensional table are organized and stored on the storage devices. One critical issue of a table placement method is the I/O performance of reading data from the placed table since it can fundamentally affect the overall query execution performance.

In Hadoop-based computing environments, several table placement methods have been proposed and implemented, such as CFile in Llama [62], Column Format (CF) [45], Record Columnar File (RCFile) [53] in Hive [5], Optimized Record Columnar File (ORC File) [8] in Hive, Parquet [9], Segment-Level Column Group Store (SLC-Store) in Mastiff [50], Trevni [10] in Avro [11], and Trojan Data Layouts (TDL) [58]. These projects have been done independently, and a comprehensive and systematic study of table placement methods is absent, leaving three critical issues unaddressed.
1. The basic structure of a table placement method has not been defined. This structure should abstract the core operations to organize and to store data values of a table in the underlying cluster. It also serves as a foundation to design and implement a table placement method.

2. A fair comparison among different table placement methods is almost impossible due to heavy influences of diverse system implementations, various auxiliary data and optimization techniques, and different workload configurations. Performance evaluations from existing work commonly report overall experimental results. Influences from different sources are hard to be distinguished.

3. There is no general guideline on how to adjust table placement methods to achieve optimized I/O read performance. To adapt various workload patterns and different underlying systems, a systematic approach to optimize a table placement method is highly desirable.

In this chapter, we present our study aiming to address the above three issues. Our study focuses on the basic structure of a table placement method, which describes core operations to organize and store data values of a table. With the basic structure, we define different table placement methods and identify critical factors on which these methods are different from each other. Then, we comprehensively and systematically evaluate impacts of these factors on the aspect of I/O read performance by our micro-benchmarks. Based on the experimental results of our micro-benchmarks, we provide general guidelines for performance optimization. To show impacts of different table placement methods on production workloads, we also provide experimental results of several large-scale experiments (macro-benchmarks). These results confirm our findings from micro-benchmarks. Then, we discuss the trade-off between the data reading efficiency and the degree of parallelism
when choosing a suitable row group size. Finally, we present ORC File as a case study to show the effectiveness of our findings and suggested actions.

Our study makes the following three main contributions.

1. We define the basic structure of a table placement method, which is used to form a table placement method, to abstract existing implementations, and to characterize differences between existing table placement methods.

2. We design and implement a benchmarking tool (for micro-benchmarks) to experimentally study different design factors of table placement methods. To provide a fair evaluation and insights into essential issues, this benchmarking tool uses an implementation of a table placement method (also called a table-placement-method implementation) to simulate variations of each design factor.

3. We comprehensively study data reading issues related to table placement methods and provide guidelines on how to adjust table placement methods to achieve optimized I/O read performance. Our guidelines are applicable to different table placement methods. Because I/O read performance is closely related to table placement methods and is a critical factor of the overall data processing performance, we believe that our results and guidelines lay a foundation to existing and future table placement methods.

The remainder of this chapter is organized as follows. Section 2.2 presents the basic structure of table placement methods and describes existing table placement methods under this structure. Section 2.3 gives an overview of our study methodology. Section 2.4 details our experimental results of micro-benchmarks. Section 2.5 details our experimental results of macro-benchmarks. Section 2.6 discusses the trade-off when choosing a suitable row
group size in clusters. Section 2.7 introduces ORC File in which we explain its design as a case study. Section 2.8 is the conclusion.

2.2 Table Placement Method

In this section, we first provide the definition of the basic structure of table placement methods. Through this definition, we are able to study existing table placement methods under a unified way. Then, we summarize design factors on which existing table placement methods are different from each other. Finally, we describe how to use implementations of RCFile and Trevni to simulate variations of each design factor.

2.2.1 Definition

The basic structure of a table placement method comprises three consecutive procedures, a row reordering procedure, a table partitioning procedure, and a data packing procedure. These three procedures are represented by three corresponding functions, which are \( f_{RR} \), \( f_{TP} \), and \( f_{DP} \), respectively. In our definition, all rows of a table form a row sequence. We will use the position of a specific row in the row sequence to refer to this row, e.g. the first row. Also, all columns of a table form a column sequence. We will use the position of a specific column in the column sequence to refer to this column, e.g. the second column. In this way, we use the position to refer to a specific data value in the table, e.g. the data value at the first row and the second column.

Row Reordering

The row reordering procedure rearranges rows of a given table based on a given function \( f_{RR} \) shown in Equation 2.1.

\[
i' = f_{RR}(i).
\]  
(2.1)
Figure 2.1: A demonstration of a table placement method. $V_{i,j}$ represents the data value at the $i$th row and $j$th column. For the purpose of a clear presentation, all two dimensional indexes of a data value are generated based on the original table.

This function will form a new sequence of rows by assigning a new position $i'$ to a row referred to as $i$ in the original sequence of rows. For example, $f_{RR}(1) = 10$ means that the first row in the original table will be the 10th row after the row reordering procedure. It is worth noting that rows in the table will not be reordered in the subsequent two procedures of table partitioning and data packing.

There are two representative examples of the row reordering procedure. First, the entire table is reordered based on the data values of certain columns. Second, we can divide
the table to multiple non-overlapping subsets and then reorder every subset. This row reordering procedure can be used when reordering the entire table is not cost-effective or not feasible. One representative purpose of the row reordering procedure is to encode or compress data values efficiently. A demonstration of a simple row reordering procedure is shown in Figure 2.1(a). In this figure, the row reordering procedure reverses the original row sequence.

**Table Partitioning**

In general, the table partitioning procedure divides the entire set of data values of the table into multiple non-overlapping subsets (called *logical subsets*) based on a given partitioning function $f_{TP}$ shown in Equation 2.2.

$$LS_{x,y} = f_{TP}(i, j).$$

(2.2)

Through $f_{TP}$, the value at the $i$th row and the $j$th column $^1$ is assigned to the logical subset $LS_{x,y}$. Like a value in a table, a logical subset $LS_{x,y}$ is identified by a two-dimensional index, which is $(x, y)$, and we also refer to this logical subset as the one located at the $x$th logical-subset-row and the $y$th logical-subset-column. For example, in Figure 2.1(b), the original table is divided to 6 logical subsets, where $LS_{1,1}$ belongs to the first logical-subset-row and the first logical-subset-column. In a logical subset, values will be stored in a row-by-row fashion, i.e. a value at the $i_1$th row and the $j_1$th column is stored before the one at the $i_2$th row and the $j_2$th if and only if $i_1 < i_2$, or $i_1 = i_2$ and $j_1 < j_2$.

Table partitioning functions in existing table placement methods (Section 2.2.2) commonly exhibit two properties which are

$^1$To be specific, the value is at the $i$th row and $j$th column of the reordered table, i.e. the output of the row reordering procedure.
1. ∀i, ∀j1, ∀j2, when \(LS_{x_1,y_1} = f_{TP}(i, j_1)\) and \(LS_{x_2,y_2} = f_{TP}(i, j_2)\), we have \(x_1 = x_2\); and

2. ∀i1, ∀i2, ∀j, when \(LS_{x_1,y_1} = f_{TP}(i_1, j)\) and \(LS_{x_2,y_2} = f_{TP}(i_2, j)\), if \(i_1 < i_2\), then \(x_1 \leq x_2\).

For these table partitioning functions, a logical-subset-row represents a set of contiguous rows. In this case, a row group is also used to refer to a logical-subset-row.

**Data Packing**

After a table being divided into multiple logical subsets, the data packing procedure will place those logical subsets into physical blocks based on a given function \(f_{DP}\) shown in Equation 2.3.

\[
P_{B_{p,q}} = f_{DP}(LS_{x,y}).
\]  

(2.3)

Through this function, a logical subset \(LS_{x,y}\) will be assigned to a physical block \(P_{B_{p,q}}\), which is identified by a two-dimensional index, \((p, q)\). For example, in Figure 2.1(c), logical subsets \(LS_{1,1}, LS_{1,3}, LS_{2,1},\) and \(LS_{2,3}\) are packed into the physical block \(P_{B_{1,1}}\). A physical block is filled by a set of logical subsets and two different physical blocks do not have any common logical subset. Also, a physical block is the storage unit of the underlying storage system. For example, a HDFS block is a physical block in Hadoop distributed filesystem (HDFS). In a physical block, logical subsets are stored by the order of their indexes, i.e. \(LS_{x_1,y_1}\) is stored before \(LS_{x_2,y_2}\) if and only if \(x_1 < x_2\), or \(x_1 = x_2\) and \(y_1 < y_2\).
Table 2.1: A summary of symbols used in Table 2.2

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>a set of columns</td>
</tr>
<tr>
<td>$C_i$</td>
<td>the $i$th column</td>
</tr>
<tr>
<td>$C_{\text{sort}}$</td>
<td>a subset of columns used as sorting keys</td>
</tr>
<tr>
<td>$f_{CG}(C_i)$</td>
<td>the column group presented by an integer index that the column $C_i$ belongs to. For example, if there are $n$ column groups, the range of $f_{CG}$ is $[1,n]$.</td>
</tr>
<tr>
<td>$RG$</td>
<td>a set of row groups (logical-subset-rows).</td>
</tr>
<tr>
<td>$RG_i$</td>
<td>the $i$th row group.</td>
</tr>
<tr>
<td>$S_r(i)$</td>
<td>the size of the $i$th row</td>
</tr>
<tr>
<td>$S(LS_{i,j})$</td>
<td>the size of the logical subset $LS_{i,j}$</td>
</tr>
<tr>
<td>$S_{PB}$</td>
<td>the user-specified maximum size of a physical block.</td>
</tr>
<tr>
<td>$S(RG_i,C_j)$</td>
<td>the size of the column $C_j$ in the row group $RG_i$.</td>
</tr>
</tbody>
</table>

2.2.2 Existing Work

With the basic structure of a table placement method defined in Section 2.2.1, we are able to describe different table placement methods in a unified way. Here, we choose Column Format (CF), CFile, Record Columnar File (RCFile), Segment-Level Column Group Store (SLC-Store), Trevni, and Trojan Data Layouts (TDL) as six examples to show the generality of our definition of the basic structure. For the purpose of simplicity, we consider that a column has a primitive data type (e.g. Integer, Double and String). If a column has a complex data type (e.g. Map), it can be considered as a single column or it can be decomposed to multiple columns with primitive types.

Optimized Record Columnar File (ORC File) and Parquet are not presented at here due to space limit. For primitive data types, they are similar to RCFile from the perspective of the basic structure. For complex data types, they provide their own approaches to decompose a column with a complex data type to multiple columns with primitive data types.
Table 2.2: Definitions of six table placement methods. For the column of $f_{RR}$, NS means not specified, and sort1 and sort2 are sorting functions specified in corresponding table placement methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>$f_{RR}$</th>
<th>$f_{TP}(i, j)$</th>
<th>$f_{DP}(x, y)$</th>
<th>Restrictions</th>
</tr>
</thead>
</table>
| CF            | NS       | $\left(\left\lceil \sum_{k=1}^{T} S_r(k) \right\rceil , j \right)$ | $(x, y)$ | 1. all row groups have the same size $S_{RG}$  
2. $\max\{S(RG_i, C_j) | RG_i \in RG, C_j \in C\} \leq S_{PB}$ |
| CFile         | sort1($C_{sort}$) | $\left(\left\lceil \frac{T}{N_{RG}} \right\rceil , j \right)$ | $\left(\left\lceil \sum_{k=1}^{T} S(LS_k, y) \right\rceil , y \right)$ | 1. all row groups have the same number of rows $N_{RG}$  
2. $\max\{S(RG_i, C_j) | RG_i \in RG, C_j \in C\} \leq S_{PB}$ |
| RCFile        | NS       | $\left(\left\lceil \sum_{k=1}^{T} S_r(k) \right\rceil , j \right)$ | $\left(\left\lceil \frac{S_{RG} \times x}{S_{PB}} \right\rceil , 1 \right)$ | 1. all row groups have the same size $S_{RG}$  
2. $S_{RG} \leq S_{PB}$ |
| SLC-Store     | sort2($C_{sort}$) | $\left(\left\lceil \sum_{k=1}^{T} S_r(k) \right\rceil , f_{CG}(C_j) \right)$ | $(x, 1)$ | 1. all row groups have the same size $S_{RG}$  
2. $S_{RG} = S_{PB}$ |
| Trevni        | NS       | $\left(\left\lceil \sum_{k=1}^{T} S_r(k) \right\rceil , j \right)$ | $(x, 1)$ | 1. all row groups have the same size $S_{RG}$  
2. $S_{RG} = S_{PB}$ |
| TDL           | NS       | $\left(\left\lceil \sum_{k=1}^{T} S_r(k) \right\rceil , f_{CG}(C_j) \right)$ | $(x, 1)$ | 1. all row groups have the same size $S_{RG}$  
2. $S_{RG} = S_{PB}$ |

After the decomposition, a table can be considered as one with only primitive data types. Other features of these two data placement methods are beyond the scope of this paper. Interested readers may refer to [12] for ORC File and to [9] for Parquet.

Using the symbols summarized in Table 2.1, we present the definitions of these six table placement methods in Table 2.2.
**Row Reordering Variations**

Among those six table placement methods in Table 2.2, CFile and SLC-Store can specify the functions of the row reordering procedure. CFile applies a sort to the entire table based on certain columns. SLC-Store can apply an optional sort to rows in every in-memory write buffer based on certain columns before flushing data to disks. These specific row reordering procedures target specific workloads. For example, CFile is designed for join operations [62]. Other table placement methods do not specify the row reordering function, and different functions can be applied as a pre-process.

**Table Partitioning Variations**

Regarding the procedure of table partitioning, those six table placement methods are divided into three categories. In the first category, CF, RCFile, and Trevni only store a single column in a row group to a logical subset. Also, these three methods set that the sizes of all row groups are the same. CFile is in the second category. Like methods in the first category, it stores a single column in a row group to a logical subset. However, CFile sets that all row groups have the same number of rows. In the third category, SLC-Store and TDL both store a column group (a subset of columns) in a row group to a logical subset, and the sizes of all row groups are the same. For these two table placement methods, a function mapping columns to column groups is required.

**Data Packing Variations**

On the data packing procedure, those six table placement methods are divided into four categories. In the first category, CF stores a single logical subset to a physical block. In the second category, CFile stores multiple logical subsets of a logical-subset-column to a physical block. A column group can have only one column.
physical block. In the third category, RCFile stores multiple row groups to a physical block. In the fourth category, SLC-Store, Trevni 3, and TDL store a single row group to a physical block.

Summary

Comparing six existing representative table placement methods, we find that they are different on four design factors, which determine their differences on functions of $f_{RR}$, $f_{TP}$, and $f_{DP}$. These factors are:

1. **Row reordering**: How to reorder rows in the table. Variations in this factor aim to facilitate specific workloads.
2. **Row group size**: The size of a row group.
3. **Grouping columns**: How to group columns.
4. **Packing column groups**: How to pack column groups in a row group into physical blocks. This factor is dependent on underlying distributed filesystems.

The first factor determines the row reordering function $f_{RR}$. Then, the second and third factors determine the table partitioning function $f_{TP}$. Finally, the fourth factor determines the data packing function $f_{DP}$.

2.2.3 A Unified Evaluation Framework

Although table placement methods shown in Table 2.2 are different from each other, to provide a fair evaluation, we use a single implementation to simulate variations of each

3Although the implementation of Trevni allows that a Trevni file can be larger than a physical block, according to the design goal of Trevni described in [10], we consider that the size of a file of Trevni is equal to the size of a physical block.
design factor of table placement methods. In this way, we are able to only focus on vari-
ations of the basic structure and to eliminate impacts from diverse codebases, different
optimizations, and various auxiliary data,

Because only implementations of RCFile and Trevni were open sourced at the point
when we started this study, we use these two as examples to illustrate how to simulate
variations of each design factor summarized in Section 2.2.2.

**Row reordering:** Row reordering can be done as a pre-process of storing tables to
RCFile or Trevni. Different approaches of row reordering can be implemented.

**Row group size:** In RCFile, the size of a row group can be adjusted. Trevni orig inally
stores a single row group in a physical block. We can adjust the row group size by adjusting
the size of a physical block. For example, to decrease the row group size used in Trevni-
based simulations, applications can explicitly write less number of rows to a physical block
before switching to write a new physical block.

**Grouping columns:** In RCFile and Trevni, we can use a composite column (e.g.
STRUCT in Hive) to store multiple grouped columns in a row by row fashion.

**Packing column groups:** We pack column groups in a row group into one or multiple
physical blocks by writing to multiple RCFile or Trevni files.

### 2.3 Study Methodology

In this work, we study the impacts of different table placement methods on I/O perfor-
ance of read operations for three reasons. First, since a table placement method describes
how data in a table are organized and stored in the underlying system, the results reasoned
from the evaluation of I/O read performance are useful to different table-placement-method
implementations. Second, computing performance is tightly coupled with the implementation, but a fair comparison among different table placement methods requires eliminating impacts from different implementations. Thus, evaluating computing performance is fundamentally against the purpose of our study. Third, a table placement method aims to store a table in a way through which applications can efficiently access the table. For this reason, the performance of read operations is a major concern for table placement methods.

Next, we identify six factors that affect the I/O performance of reading data from a table. Figure 2.2 illustrates the entire process of accessing a table. A table is stored in the storage device with a given table placement method. Then, applications (readers) will access the data of this table through an application-level filesystem (e.g. HDFS) and the
filesystem inside the local operating system (e.g. ext4 in Linux). In this entire process, there are a total number of six factors (numbered in Figure 2.2) which can have impacts on the performance of data accessing. These six factors are:

1. Table organizations, which represent data structures of the table, e.g. the number of columns, the type of a column, and the size of a value in a column;
2. Table placement methods, which describe how values in the table are organized and stored in the underlying system;
3. Storage device type, which represents the device actually storing the table, e.g. a HDD or an SSD;
4. Access patterns of the application, which represent how an application accesses the data (user-explicit operations), i.e. a sequence of read operations, each of which starts at a certain position of the file(s) storing the table and loads a certain amount of data from the underlying system;
5. Processing operations of the filesystem at the application level (application-level filesystem), which represent (1) the way that the application-level filesystem issues read operations originally from the application to the filesystem inside the local OS, and (2) the background read operations (system-implicit operations) issued by the application-level filesystem, i.e. application-level filesystem asynchronous readahead;
6. Processing operations of the filesystem inside the local OS, which represent (1) the way that the local OS issues read operations from the application-level filesystem to the storage device, and (2) the background read operations (system-implicit operations) issued by this local OS, i.e. local-OS asynchronous readahead;
To understand the impact of variations of table placement methods, we need to study how table placement methods are interacted with other five factors listed above.

To provide a comprehensive and unbiased evaluation, we used two sets of benchmarks in our experiments and our experiments were designed under the guidance of the unified evaluation framework described in Section 2.2.3. First, a set of micro-benchmarks was used to show insights into I/O read performance of a single Map task. We set three requirements for our micro-benchmarks:

1. Different software implementations of table placement methods should not affect the results of the study;
2. Experimental cases used in the study should cover a wide range to provide comprehensive results; and
3. Factors not shown in Figure 2.2 should be eliminated.

Experimental results from our micro-benchmarks will be presented in Section 2.4. Second, we used a set of macro-benchmarks to show implications of different table placement methods in real applications. We present results of our macro-benchmarks in Section 2.5.

2.4 Micro-benchmark Results

2.4.1 Rationality and Objectives

In an execution environment of Hadoop, reading a table is decomposed into several independent read operations executed by independent Map tasks. Thus, gaining insights into performance of a single Map task is critical to understand and to improve the overall table reading performance. However, gaining insights into a single Map task is hard when benchmarking a table placement method using real-world applications because lots of factors other than table placement methods can affect the performance. To solve this issue,
micro-benchmarks are needed. We aim to design a set of micro-benchmarks that can be controlled to individually test each factor summarized in Section 2.2.2. This set of micro-benchmarks should also provide insights which are applicable to different implementations of table placement methods.

### 2.4.2 Controlled Experimental Environment

In this section, we describe our controlled experimental environment. Specifically, we detail those six factors described in Section 2.3 that affect the I/O performance of reading data from a table.

**Table Organizations**

Since we focus on I/O read performance, only two aspects of the table organization matter in our study, which are the number of columns and the size of a value of a column. Because impacts from these two aspects are predictable, we believe that showing the results on tables with a fixed number of columns and a fixed value size is able to provide insights that are applicable to different table organizations. Thus, we used tables with 16 columns in our experiments and the data type of each column was string. We fixed the length of a string, which was 40 characters. In our micro-benchmarks, two tables $T_1$ and $T_2$ were used. $T_1$ had 2100000 rows, and the size of a column was 80.1 MiB $^4$. While, $T_2$ had 700000 rows, and the size of a column was 26.7 MiB.

**Table Placement Methods**

Since impacts of different row reordering functions ($f_{RR}$) are tightly coupled with workloads and we aim to provide workload-independent insights, we did not use any row

$^41$ MiB = $2^{20}$ bytes.
reordering function. We only studied impacts of different table partitioning functions ($f_{TP}$) and data packing functions ($f_{DP}$). Specifically, we considered different functions of $f_{TP}$ and $f_{DP}$ based on the second, third, and fourth factors described in Section 2.2.2.

$T_1$ and $T_2$ were stored with different table placement methods. We refer to every stored table by a 3-tuple ($f_{TP}$, $f_{DP}$, table), where $f_{TP}$, $f_{DP}$, and table are the table partitioning function, the data packing function, and the name of the table, respectively.

$f_{TP}$ is represented by a 2-tuple ($RG$, $CG$), where $RG$ shows the size of a row group, e.g. 64 MiB, and $CG$ means how columns were grouped. $CG$ can be grouped, which means that 2 columns were grouped into a column group in our micro-benchmarks, or it can be non-grouped, which means that a single column was stored in a column group. Combining both $RG$ and $CG$, we can know the table partitioning function $f_{TP}$ of a stored table. For example, (256 MiB, grouped) means that the row group size was 256 MiB, and 2 columns were grouped into a column group. Also, we use * to indicate that varied options were used. For example, (*, non-grouped) means that varied row group sizes were used and a column group only had a single column. If a table was stored by Trevni, since it does not have a knob to directly configure the row group size, we use $RG = max$.

$f_{DP}$ is represented by $n$-file, which means the number of files (physical blocks) that a row group was packed into. Specifically, we used 1-file and 16-file in our experiments. For 16-file, a column group only had a single column, i.e. for $f_{TP}$, $CG = non$-grouped.

Storage Device Type

A HDD was used because HDDs are commonly used in existing production deployments of HDFS. Also, we did not use RAID because in Hadoop clusters, Just a Bunch of Disks (JBOD) is the recommended configuration for slave nodes and using RAID is not recommended [13].
Accessing Patterns of the Application

RCFile and Trevni both determine the size of the data needed in a read operation based on metadata. For reading a row group, RCFile loads data in a column by column way. For Trevni, we used both column by column and row by row methods. Additionally, to provide unbiased results, it is worth noting that when multiple columns were accessed, we chose those columns that were not stored contiguously. Also, in our experiments, the size of unneeded data between two needed columns was always larger than the size of a read buffer.

Processing Operations of the Application-level Filesystem

We used two application-level filesystems provided by Hadoop, which were local filesystem (LFS) and distributed filesystem (DFS)\(^5\). For LFS, it uses buffered read operations and the buffer size is configurable. For DFS, it has two approaches to read data. First, if data is co-located with the client (the reader), the client can directly access the file through local OS. In HDFS, this approach is called DFS short circuit. Second, clients will contact the server having the required data in HDFS to read data through socket. In this approach, buffered read operations are used.

On the aspect of the application-level asynchronous filesystem readahead, LFS does not have a mechanism to issue asynchronous requests to prefetch data which may be used in future. However, DFS can be configured to issue asynchronous requests to local OS through \texttt{posix\_fadvise}\(^6\). Since this mechanism relies on local OS readahead, we disabled DFS readahead in our study and focused on the impact from local OS readahead.

\(^5\)LFS is a middleware on the top of the local OS. It provides the same set of interfaces as DFS
\(^6\)http://linux.die.net/man/2/posix\_fadvise
Processing Operations of the Filesystem in OS

For read operations from application-level filesystems or those directly from applications, local OS will execute these operations as requested. Also, local OS will issue asynchronous readahead requests in an adaptive way [74]. In our study, we used different maximum readahead sizes.

2.4.3 Benchmarking Tool

To have apple-to-apple comparisons, we have developed a benchmarking tool that uses RCFile and Trevni to simulate variations of each design factor in the scope of our study based on approaches discussed in Section 2.2.3. In this tool, all values of a table are serialized by the serialization and deserialization library in Hive. To focus on I/O read performance, this tool does not deserialize data loaded from underlying storage systems. To store a row of a column group, we use a composite data type (STRUCT in Hive) and this column group will be stored as a single column.

RCFile and Trevni have complementary merits. By combining them, our benchmarking tool covers a large variety of both table placement methods and behaviors of applications. The main differences of these two implementations are summarized as follows. First, RCFile has a knob to explicitly configure the size of a row group, but Trevni does not have this flexibility. Second, when reading a column in a row group, RCFile reads the entire column at once. However, Trevni only reads a block (the unit for compression) at a time. Third, when reading referenced columns in a row group, RCFile reads all needed columns in a column by column way before applications can access values of this row group, which

7 Because we want to compare results of variations of each design factor in the scope of our study, it is not meaningful to compare results from RCFile-based simulations with results from Trevni-based simulations.

8 The uncompressed size of a block is 64 KiB.
means that RCFile determines the sequence of read operations issued from applications. However, Trevni does not have this restriction, and applications can determine how to read those referenced columns.

### 2.4.4 Eliminating Unnecessary Factors

We aim to only focus on the second, third, and fourth design factors summarized in Section 2.2.2. To accomplish this objective, we need to eliminate impacts from three unnecessary factors. First, columns with different data types can blur experimental results since different columns may have significantly different impacts on the I/O performance. To eliminate the impact of this factor, in our experiments, columns of a table had the same data type and all values had the same size (also discussed in Section 2.4.2). Second, optimizations existing in different implementations of table placement methods provide workload-dependent benefits and can make experimental results hard to reason. Because table placement methods studied in this paper are following the same basic structure presented in Section 2.2.1, they are equally affected by those optimizations. Thus, we did not use any optimization in our micro-benchmarks. Third, the variety of workloads can introduce unnecessary complexity and can cause biased results. In our experiments, if a column was referenced, it was read entirely because when only a portion of rows is needed, the initial ordering of rows directly affects the performance. However, in this way, our experiments cannot evaluate the performance of different table placement methods when indexes are used to access data in a physical block. We believe that the I/O issue of randomly accessing data in a physical block is beyond the scope of this paper and it will be studied in our future work.
It is worth explaining reasons why we did not consider compression. Because we focus on I/O performance, for a given table, compression techniques only shrink the size of this table by reducing (1) the size of a column in a row group and thus (2) the size of a row group. Thus, in our study, whether a table is compressed and how a column is compressed do not affect our results.

2.4.5 Experimental Settings

In our micro-benchmarks, the machine we used had an Intel Core i7-3770K CPU and 32 GB memory. The OS was Red Hat Enterprise Linux Workstation 6.4 and the kernel release was 2.6.32-279.22.1.el6.x86_64. In our experiments, the HDD was a Western Digital RE4 1 TB SATA hard drive. We formatted the HDD with an ext4 filesystem and the HDD was mounted with options \texttt{-o noatime -o data=writeback}. The sequential bandwidth of this HDD measured through \texttt{hdparm -t} was around 130 MB/s. The version of Hadoop used in micro-benchmarks was 1.1.0. RCFile was distributed with Hive 0.9.0, and Trevni was distributed with Avro 1.7.3. Before every experiment, we first freed all cached objects, and then freed OS page cache, dentries and inodes.

In our experiments, $T_1$ was stored in the local OS filesystem and it was accessed directly or through LFS. $T_2$ was stored in DFS and it was accessed through DFS with or without DFS short circuit. The reason that we stored a smaller table in DFS is that the size of the table is comparable to the size of the data processed by a Map task of Hadoop MapReduce. We configured DFS to use a large physical block size, so every file associated with $T_2$ was a single physical block (a HDFS block). If $T_2$ was stored in a single file, only a single physical block was used.
Table 2.3: A summary of conventions of presenting different parameters in our experimental results.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>rg=x MiB</td>
<td>The row group size was set to x MiB</td>
</tr>
<tr>
<td>io=y KiB</td>
<td>The size of read buffer was set to y KiB</td>
</tr>
<tr>
<td>OS_RA=z KiB</td>
<td>The maximum size of local OS readahead was set to z KiB</td>
</tr>
</tbody>
</table>

2.4.6 Experimental Results

First, we summarize conventions of presenting different parameters in our experimental results in Table 2.3. We also follow these conventions in our macro-benchmarks (Section 2.5).

Row Group Size

With a given table, the size of a row group determines the size of every column in this row group. Because data of a column stored in two row groups may not stored contiguously, if the size of this column in a row group is small, lots of seeks will be introduced and the performance of reading this column will be degraded.

Figure 2.3 shows results of reading 4 columns of tables stored with ((*, non-grouped), I-file, T1) through LFS with different LFS read buffer sizes and different maximum local OS readahead sizes. We also did the same experiments on reading 1 and 2 columns, and we have the same observations as reading 4 columns.

Figure 2.3(a) shows sizes of data needed and sizes of data actually read from the HDD through LFS (collected by iostat). Through Figure 2.3(a), we have three observations.
First, we focus on results when local OS readahead is disabled (cases with the prefix of OS_RA=0 KiB). When a small row group is used, a buffered read operation through LFS will read lots of unnecessary data from unneeded columns. For example, when the row group size was set to 4 MiB, in our benchmarks, the size of a column in a row group was implicitly set to 256 KiB. However, the size of a column in a row group cannot be exactly 256 KiB and it is actually slightly larger than 256 KiB. Thus, when the read buffer size was 256 KiB, the size of the data actually read from the storage device was around two times as large as the size of the data needed. Through Figure 2.3(a), we can see that the size of four columns was 320.4 MiB data. However, 655.5 MiB data was loaded from the storage device when settings mentioned above were used.

![Figure 2.3: Sizes of needed data and actually read data, and throughputs on reading 4 columns of tables stored with ((*, non-grouped), 1-file, \(T_1\)) through LFS.](image-url)
Second, if buffered read operations read both needed data and unneeded data from the device, local OS readahead will amplify the size of unneeded data since local OS will prefetch data from those unneeded columns. For example, when the row group size was 4 MiB and read buffer size was 256 KiB, to read four columns, more than half of the data in the table was loaded from the HDD when local OS readahead was enabled.

Third, with the increase of the row group size, there will be less buffered read operations that read unneeded data from the device, and local OS readahead will do less useless work.
For example, when the row group size was 256 MiB, read buffer size was 256 KiB, and maximum local OS readahead size was 1024 KiB, to read four columns, 360.6 MiB data was actually read and the size of needed data was 320.4 MiB.

Figure 2.3(b) shows throughputs of reading data through LFS corresponding to Figure 2.3(a). When a read operation loads unnecessary data and OS readahead cannot efficiently prefetch needed data, the throughput of data reading is very low. For example, when the row group size was 4 MiB, read buffer size was 256 KiB, and maximum local OS readahead size was 1024 KiB, the throughput of reading four columns was 31.8 MiB/s. When the row group size was increased to 256 MiB, the throughput was 86.8 MiB/s.

Figure 2.4 shows results when reading data of tables stored with ((*, non-grouped), 1-file, $T_2$) through DFS with 256 KiB read buffer size. We can see that sizes of data actually read and throughputs have the same trend as those presented in Figure 2.3. It is worth noting that, when DFS short circuit is enabled, RCFFile does not use buffered read operations. Thus, when reading the metadata of a row group, it generates lots of small read operations (e.g. 1 byte request). This behavior is the reason that throughputs of cases with DFS short circuit were significantly lower than those cases without DFS short circuit when local OS readahead was disabled.

Grouping Columns

To present the impacts of grouping columns to column groups, we first compare results of reading two columns of tables stored with ((max, non-grouped), 1-file, $T_1$) and ((max, grouped), 1-file, $T_1$) directly from the filesystem in the local OS, which are shown in Figure 2.5. We configured Trevni to directly access local filesystem instead of accessing files through LFS of DFS. Through this way, we can eliminate the impacts from buffered read operation introduced by LFS and DFS. When accessing the table that does not
group columns, we used both the row-oriented access method and column-oriented access method. Through Figure 2.5, we can see that grouping columns has a significant advantage only when the row-oriented access method is used. Because row-oriented access method will generate lots of seeks among two columns to fetch the data, by grouping these two columns together, applications only read a single column group and thus, seeks are eliminated. However, compared with the column-oriented access method on non-grouped two columns, accessing two grouped columns does not have significant performance improvement (only 3 to 4 MiB/s improvement). This small difference on throughputs is caused by one additional disk seek and handling different metadata (in our experiments, the table stored with ((max, non-grouped), 1-file, $T_1$) had 16 columns and the table stored with ((max, grouped), 1-file, $T_1$) had 8 columns).

Additionally, we used tables stored with ((*, non-grouped), 1-file, $T_2$) and ((*, grouped), 1-file, $T_2$) to study combined impacts of different row group sizes and grouping columns. Figure 2.6 shows results on DFS when accessing two columns of a table with and without HDFS short circuit, varied row group sizes, and different maximum local OS readahead.
Figure 2.6: Sizes of needed data and actually read data, and throughputs on reading 2 columns of tables stored with ((*, non-grouped), 1-file, $T_2$) and ((*, grouped), 1-file, $T_2$) through DFS. DFS has a 256 KiB read buffer.

sizes. From cases with and without DFS short circuit, we can see that, when a small row group size is used, grouping columns can have performance improvement. Without DFS
short circuit, buffered read operations are used. In this case, grouping columns can increase the size of needed data in a read buffer when a small row group size is used. Also, when a small row group is used, local OS readahead can easily do useless work by prefetching unneeded columns. By grouping columns, local OS readahead can do more useful work. However, when a large row group size is used, grouping columns does not yield significant performance improvement. The limited improvements over non-grouped cases are caused by saving a limited number of disk seeks and handling different metadata.

**Packing Column Groups**

Storing a row group to multiple physical blocks introduces a new problem on how to group columns or column groups into physical blocks. For simplicity, when columns in a row group were packed into multiple physical blocks, we stored a single column in a physical block.
To show impacts of packing columns in a row group into multiple physical blocks, we first present results of accessing 4 columns of tables stored with \((\ast, \text{non-grouped}), 1\text{-file}, T_1\) and \((\ast, \text{non-grouped}), 16\text{-file}, T_1\) directly from the filesystem in the local OS in Figure 2.7. We can see that, for both the row-oriented access method and column-oriented access method, using multiple physical blocks to store a row group does not provide significant performance improvement over corresponding cases that store all columns in a physical block.

We also present results of reading 4 columns from tables stored with \((\ast, \text{non-grouped}), 1\text{-file}, T_1\) and \((\ast, \text{non-grouped}), 16\text{-file}, T_1\) through LFS in Figure 2.8. Through results,
we can see that packing columns into different physical blocks only have performance advantages when a smaller row group size is used and a relative large maximum local OS readahead size is used. In this configuration, when a single physical block (a single file in the local OS) is used to store a row group, the local OS can do lots of useless work by prefetching data from unneeded columns because it is not aware of boundaries between columns. When a large row group size is used, storing columns in different physical blocks does not have performance advantages over storing all columns in one file. We also evaluated performance of reading 4 columns from tables stored with ((*, non-grouped), 1-file, $T_2$) and ((*, non-grouped), 16-file, $T_2$) through DFS. Results of these experiments show the same trend as Figure 2.8.

### 2.4.7 Summary

With our thorough study and insightful findings, we suggest three general-purpose and effective actions on how to adjust table placement methods to achieve optimized I/O read performance.

**Row Group Size**

**Action 1: using a sufficiently large row group size.** As shown in our experimental results, the size of a row group should be sufficiently large. Otherwise, a buffered read operation issued by the underlying storage system can load more data from unneeded columns and the disk seeking cost cannot be well amortized.

In a distributed environment, determining the suitable row group size requires careful considerations on the single machine data reading performance and the available degree of parallelism. We will discuss these considerations in Section 2.6.
Grouping Columns

**Action 2:** if a sufficiently large row group size is used and columns in a row group can be accessed in a column-oriented way, it is not necessary to group multiple columns to a column group. Once two or more columns are grouped, reading a subset of columns stored in this column group requires to load the data of the entire column group from the underlying system, and thus, unneeded data will be read. Also, because of the variety of workloads, determining how to group columns may also introduce burden and overhead on grouping columns.

If applications have to access columns in a row group in a row-oriented way, it may worth grouping columns to column groups. For example, if all columns in a column group are needed, this column group can be read sequentially.

Packing Column Groups

**Action 3:** if a sufficiently large row group size is used, it is not necessary to pack column groups (or columns) into multiple physical blocks. When a sufficiently large row group size is used, the negative impacts from local OS asynchronous readahead are negligible. Also, if columns in a row group need to be placed in the same machine, packing column groups into multiple physical blocks may require additional supports from the underlying system, which may not be feasible and can introduce overhead.

If the maximum size of a row group is limited by the data processing environment and the negative impacts from local OS asynchronous readahead are observable, it may worth packing column groups into multiple physical blocks, which can indirectly enlarge the size of a row group. In this case, the issue of co-locating physical blocks that store column groups in a row group is still needed to be carefully considered and addressed.
Table 2.4: A summary of datasets we used in our macro-benchmarks. The column of Scale Factor shows the scale setting we used to generate every dataset, and the column of Total Size shows the corresponding size of the entire dataset.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Scale Factor</th>
<th>Total Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSB</td>
<td>1000</td>
<td>1000 GB</td>
</tr>
<tr>
<td>TPC-H</td>
<td>1000</td>
<td>1000 GB</td>
</tr>
<tr>
<td>SS-DB</td>
<td>large</td>
<td>9.9 TB</td>
</tr>
</tbody>
</table>

2.5 Macro-benchmark Results

In this section, we present results of a set of macro-benchmarks to show impacts of different row group sizes (Section 2.5.3) and impacts of grouping columns (Section 2.5.3) in production workloads. We do not present results of packing column groups of a row group into multiple physical blocks at here because of two reasons. First, packing column groups of a row group into multiple physical blocks prevents local filesystem from prefetching unnecessary data. The impact of it has been studied clearly in Section 2.4.6. Second, in existing systems, packing column groups in a row group into more than one physical blocks is not practical because it increases the memory usage of the master node (e.g. NameNode in HDFS), and it requires customized block placement policy to achieve expected performance.

2.5.1 Workloads

In our macro-benchmarks, we evaluated queries selected from Star Schema Benchmark (SSB) [35], TPC-H [14], and SS-DB (A standard science DBMS benchmark) [34]. In our evaluation, we generated a large dataset for every benchmark. Table 2.4 shows the scale factor we used and the size of every dataset. Queries used in our experiments are summarized as follows.
SSB Query 1.1, 1.2, and 1.3: Every query in these three queries first has a join operation between a fact table \((\text{lineorder})\) and a dimension table \((\text{date})\), and then has an aggregation operation on results of the join operation. These queries have different filter factors (the ratio of needed rows to the total number of rows) on the fact table. In our results, query 1.1, 1.2, and 1.3 are referred to as ssb.1.1, ssb.1.2, and ssb.1.3, respectively.

TPC-H Query 6: This query is an aggregation query on the table of \text{lineitem}, which is the largest table in TPC-H.

SS-DB Query 1.easy, 1.medium, and 1.hard: These queries are aggregation queries on images. These three queries differ from each other on the difficulty, i.e. easy, medium, and hard. With the increase of the difficulty (e.g. from easy to medium), the volume of data that will be processed in the query increases. It is worth noting that queries in SS-DB processed only one cycle of images. In our dataset, one cycle of images has a raw size of around 200 GB. In our results, query 1.easy, 1.medium, and 1.hard are referred to as ssdb.q1.easy, ssdb.q1.medium, and ssdb.q1.hard, respectively.

These queries were selected based on two reasons. First, they are from widely used benchmarks. Second, they can be executed in a single MapReduce job and thus make it easier to reason impacts of different table placement methods on real applications than complex queries. If we use queries that need to be evaluated by multiple MapReduce jobs, we would introduce more operations which are invariant to different choices of table placement methods. Those irrelevant operations will equally contribute to elapsed times in our experiments. Thus, it is not reasonable to use queries that need to be evaluated by multiple MapReduce jobs in this paper.
Figure 2.9: Sizes of data actually read and elapsed times of queries introduced in Section 2.5.1. GroupingColumn=x represents the scheme of grouping columns. The unit of the row group size (rg) is MiB. The unit of the read buffer size (io) is KiB.

2.5.2 Experimental Settings

We used Hive 0.11.0 running on Hadoop 1.2.0 in our macro-benchmarks. Because Trevni has not been integrated into Hive, we only used the RCFile-based simulation. We used Hive in a 101-node Hadoop cluster, which was setup in Amazon EC2. This cluster had one master node and 100 slave nodes. The virtual machine (VM) instance of the master node was m1.large. The VM instance of 100 slave nodes was m1.medium. In this
cluster, the master node ran the NameNode and the JobTracker. Each slave node ran a DataNode and a TaskTracker.

HDFS, MapReduce and the local filesystem in every node were setup as follows. For HDFS, we set replication factor to 3 and HDFS block size to 256 MiB. For MapReduce, we set every slave node to have 1 Map slot and 1 Reduce slot based on specifications of the slave node. We also set that a spawned JVM can be reused unlimited times. To measure exact elapsed times of a Map phase and a Reduce phase, the Reduce phase was started after all Map tasks had finished. For the local filesystem, we set the maximum local OS readahead sizes to 512 KiB, which is a recommended size for Hadoop clusters [15].

For every experiment, we report the median of three repetitions. Before every repetition of our experiments, we first freed all cached objects, then freed OS page cache, dentries and inodes.

2.5.3 Experimental Results

Our experimental results of macro-benchmarks are presented in Figure 2.9. In our experiments, we used 4 MiB, 64 MiB, and 128 MiB as row group sizes. We used 64 KiB and 128 KiB as read buffer sizes. We used two schemes to group columns. In figures presented in this section, GroupingColumn=no means that we did not group columns and we stored a single column in a column group. GroupingColumn=Best means that we stored all needed columns of a query into a single column group.

Row Group Size

Based on Figure 2.9, we can see that when the row group size is small, Hive will read significantly more data than needed for all queries from three benchmarks. With the increase of row group size, the size of the data actually read decreases significantly.
Additionally, elapsed times of Map phases are affected by different row group sizes. From 4 MiB row group size to 64 MiB row group size, the elapsed times of the Map phase decrease significantly. However, when further increase the row group size from 64 MiB to 128 MiB, the change on the elapsed times of the Map phase is not significant. It is because 64 MiB row group size is large enough for these workloads. Any further increase of the row group size only result in a small gain of query processing performance.

**Grouping Columns**

When the row group size is small, grouping columns can provide benefits in two aspects. First, because needed columns are stored together, a buffer read operation can load a less amount of unnecessary data from disks. Second, a less number of disk seeks is needed to read needed columns. However, when the row group size is large enough, grouping columns cannot provide significant performance benefits. Based on Figure 2.9, for 64 MiB or 128 MiB row group size, there is no significant difference between storing a single column in a column group and the best column grouping scheme which stored only needed columns in a column group.

**2.5.4 Summary**

From experimental results of our macro-benchmarks, we can confirm that our observations of our micro-benchmarks also exist in a production-like environment. Moreover, our suggested actions based on results of micro-benchmarks are also valid in practice.

**2.6 The Data Reading Efficiency and the Degree of Parallelism**

As shown in our experimental results, a sufficiently large row group size can make read operations efficient on HDDs. In recent work, e.g. ORC File [8] and Parquet [9], a large
row group size is also recommended. However, in a distributed environment, increasing the row group size may not always result in performance improvement. On one hand, for a given table, using a large row group size can improve the efficiency of data reading because more values of a column (or a column group) in a row group can be stored contiguously, which improves the efficiency of disk read operations. Cost-efficient read operations are highly desirable, e.g. in a cloud environment like Amazon Elastic MapReduce [16], a higher data reading efficiency implies a lower price-performance ratio. On the other hand, a smaller row group size can increase the available degree of parallelism. A high degree of parallelism has three benefits. First, more tasks can be used to process data in parallel, which can increase the aggregated data reading throughput. Second, tasks assigned to machines can achieve better dynamic load balance, which reduces elapsed time caused by unbalanced loads. Third, when a machine fails, more available ones can be used to process tasks originally assigned to the failed one, which decreases the recovery time in the presence of machine failure events.

The essential issue to be considered is the trade-off between the degree of parallelism in a cluster and the data reading efficiency in each individual machine (or each data processing task). Users having different tables, infrastructures and requirements on costs should choose different row group sizes with considerations of the above trade-off. Here we provide an example to illustrate our point of view. The table in this example is based on lineitem in TPC-H benchmark [14] and we call this table $L$. Values from the same column in $L$ have the same size. The size of a value in a column is the average value size of the corresponding column in lineitem calculated based on TPC-H specification (revision 2.15.0) and the actual data type used in Hive [17]. This table $L$ has four integer columns, four double columns, two 1-byte string columns, four 10-byte string columns,
one 25-byte string column, and one 27-byte string column. We assume that a value in an integer column occupies four bytes and that a value in a double column occupies eight bytes.

In this example, we assume that a logical subset in a row group only has values of a single column. We then analyze the change of data reading efficiency on reading a single column of a row group with the increase of the row group size. The data reading efficiency is calculated as the ratio of the data reading throughput to the disk bandwidth. Equation 2.4 shows how we calculate the data reading efficiency with a given size of a column in a row group ($S_C$), disk seek time ($t_{seek}$), and disk bandwidth ($B_D$).

\[
\text{Efficiency} = \frac{\text{Throughput}}{B_D} = \frac{S_C}{t_{seek} + \frac{S_C}{B_D} \times \frac{1}{B_D}}, \quad (2.4)
\]

Figure 2.10 shows the data reading efficiency on reading different columns with the increase of the row group size. In this analysis, we set $t_{seek} = 10$ ms and $B_D = 100$ MiB/s. For the purpose of simplicity, we do not consider the impact from buffered read operations. Because different columns have different sizes of values, they have different data reading efficiency for a given row group size.

We summarize the implications presented in Figure 2.10 as follows. First, when we start to increase the row group size from a small number, the increase of data reading efficiency of those columns with small value sizes increases much slower than those columns with large value sizes. For the smallest columns (those two 1-byte string columns), when the row group size is 64 MiB, the efficiency is only 31%. However, for the 27-byte string column, it can achieve 75% efficiency when the row group size is less than 20 MiB. Second, the increase of data reading efficiency via enlarging the row group size decreases when the row group becomes bigger. Taking columns with type double as an example, to achieve 50% and 75% data reading efficiency, 17.85 MiB and 53.57 MiB row group sizes should
Figure 2.10: The change of data reading efficiency with the increase of the row group size.

be used, respectively (hollow squares in the figure). Third, for a table with columns having a high variance of average value sizes, achieving a certain data reading efficiency goal for all columns may not be practical because a very large row group size can limit the degree of parallelism. We conclude that an optimal row group size is determined by a balanced trade-off between the data reading efficiency and the degree of parallelism.

2.7 ORC File: A Case Study

Recently, Optimized Record Columnar File (ORC File) has been designed and implemented to improve RCFile. In this section, we explain its design from the perspective of
our study as a case study. Specifically, we describe ORC File from the aspect of the basic structure of table placement methods, i.e. row reordering, table partitioning, and data packing. In the end, we also highlight other improvements in ORC File.

**Row reordering:** The writer of ORC File does not sort a table, but users can explicitly sort it as a pre-process. Like other table placement methods, a sorted table may improve the efficiency of data encoding and compression. Also, ORC File supports predicate pushdown [12] [18]. The improvement of performance contributed by this feature on reading a sorted column is more significant because the reader of ORC File can effectively skip rows which do not satisfy the given predicates.

**Table partitioning:** On table partitioning, ORC File is similar to RCFile. For primitive data types, the function of table partitioning of RCFile is also applicable to ORC File. However, ORC File uses a 256 MiB row group size by default (a row group is called by a stripe in ORC File). This large stripe size improves the performance of reading data, which is consistent to our suggested action in Section 2.4.7. Since the default stripe size is large, as we discussed in Section 2.6, it is possible that the degree of parallelism on processing a given table is bounded by the stripe size [19]. ORC File takes this issue into consideration in its optimization. Also, ORC File does not group columns, which is consistent with our suggested action in Section 2.4.7. For complex data types like Map, ORC File decomposes a column with a such data type to multiple actual columns with primitive data types [20]. In contrast, RCFile does not decompose a column with a complex data type.

**Data packing:** ORC File does not store a stripe (a row group) to multiple files. Because the size of a stripe is usually large and an ORC File reader uses the column-oriented access method, it is not necessary to store a stripe to multiple files from the perspective of I/O read performance. Also, storing all columns of a stripe in a single file is friendly to the
NameNode of HDFS. This design choice is consistent with our suggested action in Section 2.4.7.

ORC File also have several major improvements on auxiliary data and optimization techniques. On the aspect of auxiliary data, ORC File has integrated indexes that support rapidly seeking to a specific row (identified by the row number) and predicate pushdown. In contrast to RCFile, ORC File records the locations of stripes in the footer. Thus, the reader of ORC File does not need to scan the file to locate the starting point of a stripe. On the aspect of optimization techniques, ORC File uses two levels of compression. The writer first automatically applies type-specific encoding methods to columns with different data types. Then, an optional codec can be used to compress encoded data streams. Interested readers may refer to [12] for details of these improvements.

2.8 Summary

We have presented our study on the basic structure and essential issues of table placement methods in Hadoop-based data processing systems. We have proposed a general abstraction framework that makes it possible to abstract and compare different table placement methods in a unified way. Based on this framework, we have developed a benchmarking tool that simulates variations of table placement methods. We have conducted a set of comprehensive experiments with our benchmarking tool to understand the fundamental impacts of different variations of each design factor. Experimental results from our large-scale experiments have also confirmed results of micro-benchmarks in production workloads.

Our main findings are four-fold: (1) the row group size should be large enough so that the column (or column group) size inside a row group will be large enough to facilitate applications to achieve efficient read operations; (2) when the row group size is large enough
and the column-oriented access method is used to read columns, it is not necessary to group multiple columns to a column group; (3) when the row group size is large enough, it is not necessary to store columns or column groups in a row group to multiple physical blocks; (4) in a distributed computing cluster, the row group size should be selected with considerations on the trade-off between the data reading efficiency in each machine or task and the degree of parallelism. Our presentation of ORC File makes a case on the effectiveness of our findings and suggested actions.

Our benchmarking tool used in micro-benchmarks and scripts used in macro-benchmarks are available at https://github.com/yhuai/tableplacement.
Chapter 3: Reducing Unnecessary Data Movements in Distributed Query Plans

3.1 Introduction

A query execution plan is a sequence of operations to access and process data in a data analytics system. After a query is submitted to a database, the query optimizer will choose an efficient execution plan. In this chapter, we present our query optimization methods, which reduce unnecessary data movements. The system we are focusing on is Apache Hive.

Apache Hive is a data warehouse system for Apache Hadoop [21]. It has been widely used in organizations to manage and process large volumes of data, such as eBay, Facebook, LinkedIn, Spotify, Taobao, Tencent, and Yahoo!. As an open source project, Hive has a strong technical development community working with widely located and diverse users and organizations. In recent years, more than 100 developers have made technical efforts to improve Hive on more than 3000 issues. With its rapid development pace, Hive has been significantly updated by new innovations and research since the original Hive paper [71] was published four years ago. We will present its major technical advancements in this section.
Hive was originally designed as a translation layer on top of Hadoop MapReduce. It exposes its own dialect of SQL to users and translates data manipulation statements (queries) to a directed acyclic graph (DAG) of MapReduce jobs. With an SQL interface, users do not need to write tedious and sometimes difficult MapReduce programs to manipulate data stored in Hadoop Distributed Filesystem (HDFS).

This highly abstracted SQL interface significantly improves the productivity of data management in Hadoop and accelerates the adoption of Hive. The efficiency and productivity of Hive are largely affected by how its data warehouse layer is designed, implemented, and optimized to best utilize the underlying data processing engine (e.g. Hadoop MapReduce) and HDFS. In order to make Hive continuously satisfy requirements of processing increasingly high volumes of data in a scalable and efficient way, we must improve both data storage as well as query execution aspect of Hive. First, Hive should be able to store datasets managed by it in an efficient way which guarantees both storage efficiency as well as fast data access. Second, Hive should be able to generate highly optimized query plans and execute them using a query execution model that utilizes hardware resources well.

The query planning component is one of the most important components that determine the query execution performance. The query planner in the original Hive translates every operation specified in this query to an operator. When an operation requires its input datasets to be partitioned in a certain way, Hive will insert an operator called ReduceSink-Operator (RSOp) as the boundary between a Map phase and a Reduce phase. Then, the query planner will break the entire operator tree to MapReduce jobs based on these boundaries. During query planning and optimization, the planner only focuses on a single data

\[\text{In the rest of this chapter, this kind of data operations are called major operations and an operator evaluating a major operation is called a major operator.}\]
operation or a single MapReduce job at a time. This query planning approach can significantly degrade the query execution performance by introducing unnecessary and time consuming operations. For example, the original query planner was not aware of correlations between major operations in a query [60]. Thus, the key shortcoming of the query planner is that the query translation approach ignores relationships between data operations, and thus introduces unnecessary data movements hurting the performance of query execution.

To reduce unnecessary data movements, we introduce two optimizations that carefully analyze the distributed plan of a query. These optimizations can reduce unnecessary data movements in distributed query plans and thus, significantly improve the performance of the query execution.

The rest of this chapter is organized as follows. Section 3.2 provides an overview of Hive’s Architecture. Section ? presents shortcomings we have identified and the rationale of the related advancements. Section 3.3 introduces the advancement on the query planning component. Section 3.4 reports results of our evaluation on these advancements. Section 3.5 surveys related work. Section 3.6 is the conclusion.

### 3.2 Hive Architecture

Figure 3.1 shows the architecture of Hive. Hive exposes two interfaces to users to submit their statements. These interfaces are Command Line Interface (CLI) and HiveServer2 [22]. Through these two interfaces, a statement will be submitted to the Driver. The Driver first parses the statement and then passes the Abstract Syntax Tree (AST) corresponding to this statement to the Planner. The Planner then chooses a specific planner implementation to analyze different types of statements. During the process of analyzing a submitted
Figure 3.1: The architecture of Hive. Rounded rectangles are components in Hive. Shaded rounded rectangles are advanced components, and they also show major advancements that will be introduced in this chapter.

statement, the Driver needs to contact the Metastore to retrieve needed metadata from a Relational Database Management System (RDBMS), e.g. PostgreSQL.

Queries used for data retrieval and processing are analyzed by the Query Planner. Hive translates queries to executable jobs for an underlying data processing engine that is currently Hadoop MapReduce\textsuperscript{10}. For a submitted query, the query planner walks the AST of this query and assembles the operator tree to represent data operations of this query. An operator in Hive represents a specific data operation. For example, a FilterOperator

\textsuperscript{10}Starting from Hive 0.13, a query can also be translated to a job that is executable in Apache Tez [3]. Without loss of generality, we will mainly use MapReduce in this chapter since it is the original data processing engine used by Hive.
is used to evaluate predicates on its input records. Because a query submitted to Hive will be evaluated in a distributed environment, the query planner will also figure out if an operator requires its input records to be partitioned in a certain way. If so, it then inserts a boundary represented by one or multiple ReduceSinkOperators (RSOps) before this operator to indicate that the child operator of these RSOps need rows from a re-partitioned dataset. For example, for a group-by clause GROUP BY key, a RSOp will be used to tell the underlying MapReduce engine to group rows having the same value of key. After an operator tree is generated, the query planner applies a set of optimizations to the operator tree. Then, the entire operator tree will be passed to the task compiler, which breaks the operator tree to multiple stages represented by executable tasks. For example, the MapReduce task compiler generates a DAG of Map/Reduce tasks assembled in MapReduce jobs based on an operator tree. In the end of query planning, another phase of optimizations are applied to generated tasks.

After the query planner has generated MapReduce jobs, the Driver will submit those jobs to the underlying MapReduce engine to evaluate the submitted query. In the execution of a Map/Reduce task, operators inside this task are first initialized and then they will process rows fetched by the MapReduce engine in a pipelined fashion. To read/write a table with a specific file format, Hive assigns the corresponding file reader/writer to tasks reading/writing this table. For a file format, a serialization-deserialization library (called SerDe in the rest of this chapter) is used to serialize and deserialize data. After all MapReduce jobs have finished, the Driver will fetch the results of the query to the user who submitted the query.

Besides processing data directly stored in HDFS, Hive can also process data stored in other storage systems, e.g. HBase [23]. For those systems, a corresponding Storage
Handler is needed. For example, the HBase storage handler is used when a query needs to read or write data from or to HBase.

3.3 Optimizations

To address the shortcoming of the query planner, we first have identified three major issues caused by the the original query translation approach of Hive, introducing unnecessary operations and data movements, and significantly degrading the performance of a query. These issues are summarized as follows.

- **Unnecessary Map phases.** Because a MapReduce job can only shuffle data once, it is common that a query will be executed by multiple MapReduce jobs. In this case, the Map phase loading intermediate results is used merely to load data back from HDFS for data shuffling. Thus, if a MapReduce job generating intermediate results does not have a Reduce phase, it introduces an unnecessary Map phase to load its outputs back from HDFS.

- **Unnecessary data loading.** For a query, a table can be used by multiple operations. If these operations are executed in the same Map phase, Hive can load the table once. However, if these operations are in different MapReduce jobs, this table will be loaded multiple times, which introduce extra I/O operations to the query evaluation.

- **Unnecessary data re-partitioning.** Originally, Hive generates a MapReduce job for every major operation (a data operation requiring its input datasets to be partitioned in a certain way). In a complex query, the input datasets of a major operation may be already partitioned in a proper way by its previous major operations. For this case, we call these two operations are correlated. The original Hive ignores
correlations between major operations and thus, can introduce unnecessary data re-
partitioning, which results in unnecessary MapReduce jobs and poor query evalua-
tion performance.

For unnecessary Map phases, we have analyzed when Hive will generate Map-only jobs and added an optimization to merge a Map-only job to its child job (Section 3.3.1). For unnecessary data loading and re-partitioning, we have introduced a Correlation Optimizer to eliminate unnecessary MapReduce jobs (Section 3.3.1). In this section, we use a running example shown in Figure 3.2(a) to illustrate optimizations that we will introduce in the rest of this section.

3.3.1 Eliminating Unnecessary Map phases

In Hive, a Map-only job is generated when the query planner converts a MapReduce job for a Reduce Join to a Map Join. In a Reduce Join, input tables are shuffled and they are joined in Reduce tasks. On the other hand, in a Map Join, two tables are joined in Map tasks. There are several join schemes for a Map Join. One representative example is, for a two way join, to build a hashtable for the smaller table and load it in every Map task reading the larger table for a hash join.

Because we convert a Reduce Join to a Map Join after MapReduce jobs have been assembled, we have a Map-only job for every Map Join at first. Consequently, we would introduce unnecessary operations and elapsed time to the evaluation of the submitted query. To eliminate those unnecessary Map phases, every time when we convert a Reduce Join to a Map Join, we try to merge the generated Map-only job to its child job if the total size of small tables used to build hash tables in the merged job is under a configurable threshold.
SELECT big1.key, small1.value1, small2.value1, big2.value1, sq1.total
FROM big1
JOIN small1 ON (big1.sKey1 = small1.key)
JOIN small2 ON (big1.sKey2 = small2.key)
JOIN (SELECT key, 
        avg(big3.value1) AS avg,
        sum(big3.value2) AS total
    FROM big2 JOIN big3 ON (big2.key = big3.key)
    GROUP BY big2.key) sq1 ON (big1.key = sq1.key)
JOIN big2 ON (sq1.key = big2.key)
WHERE big2.value1 > sq1.avg;
(a) Query

Figure 3.2: The running example used in Section 3.3. For an arrow connecting two operators, it starts from the parent operator and ends at the child operator.
This threshold is used to prevent a Map task loading a partition of the big table running out of memory.

In our example shown in Figure 3.2, small1 and small2 are two small tables, and big1 is a large table. At first, Hive generates regular Reduce Joins for Joins involving small1 and small2. Then, Hive automatically converts these two Reduce Joins to Map Joins, which are shown as M-JoinOp-1 and M-JoinOp-2 in Figure 3.2(b). With the optimization introduced in this subsection, these two Map Joins are merged into the same Map phase and will be executed in a pipelined fashion. The results of these two Map Joins will be emitted to the shuffling phase for the Reduce Join R-JoinOp-4.

3.3.2 Correlation Optimizer

To eliminate unnecessary data loading and re-partitioning, we have introduced a Correlation Optimizer into Hive. This optimizer is based on the idea of correlation-aware query optimizations proposed in YSmart [60]. The main idea of this optimizer is to analyze if a major operation really needs to re-partition its input datasets and then to eliminate unnecessary MapReduce jobs through removing boundaries between a Map phase and a Reduce phase. The optimized plan will have less number of shuffling phases. Also, in the optimized plan, a table originally used in those MapReduce jobs are used in the same MapReduce job and Hive can automatically load the common table once instead of multiple times in the original plan.

In this section, we first introduce correlations exploited by this optimizer. Then, we introduce how we have implemented this optimizer. We will also cover new challenges we have overcome and how the design of the Correlation Optimizer is different from the original YSmart.
Correlations

In the implementation of the Correlation Optimizer, we consider two kinds of correlations in a query, input correlation and job flow correlation. An input correlation means that a table is used by multiple operations originally executed in different MapReduce jobs. A job flow correlation means that when a major operator depends on another major operator, these two major operators require their input datasets to be partitioned in the same way. Interested readers may refer to the original paper of YSmart [60] for details about correlations.

Implementation

During the development of the Correlation Optimizer, we found that the biggest challenge was not to identify optimization opportunities in a query, but to make the optimized plan executable. Except for scanning a table once for operations appearing in the same Map phase, we found that Hive did not have any support for multi-query optimizations. Moreover, the push-based model inside a Reduce task requires an intrinsic coordination mechanism between major operators executed in the same Reduce task to make those major operations work in the same pace. To make the optimized plan work, we need to extend the query execution layer with the development of the optimizer.

Correlation Detection  In Hive, every query has one or multiple terminal operators which are the last operators in the operator tree. FileSinkOperator((FSoPs)) is the name of the terminal operator. To give an easy explanation, if an operator A is on another operator B’s path to a FSoP, A is the downstream of B and B is the upstream of A.

For a given operator tree like the one shown in Figure 3.2(b), the Correlation Optimizer starts to visit operators in the tree from those FSoPs in a depth-first way. The tree walker
stops at every RSOp. Then, a correlation detector starts to find a correlation from this RSOp and its siblings by finding the furthest correlated upstream RSOps in a recursive way. If we can find any correlated upstream RSOp, we find a correlation. Currently, there are three conditions to determine if an upstream RSOp and a downstream RSOp are correlated, which are (1) emitted rows from these two RSOps are sorted in the same way; (2) emitted rows from these two RSOps are partitioned in the same way; and (3) these RSOps do not have any conflict on the number reducers. Interested readers may refer to our implementation [24] and design document [25] for details.

For the example shown in Figure 3.2(b), the Correlation Optimizer can find one correlation containing six RSOps, which are RSOp−1 to RSOp−6.

**Operator Tree Transformation**  After the Correlation Optimizer finds all correlations in an operator tree, it starts to transform the operator tree.

There are two kinds of RSOps in a correlation. First, we have a list of bottom layer RSOps which are necessary ones used to emit rows to the shuffling phase. Originally, because the MapReduce engine considers the input data of the Reduce phase as a single data stream, each bottom layer RSOp was assigned a tag which is used to identify the source of a row at the Reduce phase. For a correlation, we reassign tags to those bottom layer RSOps and keep the mapping from new tags to old tags. For example, in Figure 3.3, RS1, RS2, RS3 and RS4 are bottom layer RSOps and their old tags are 0, 0, 1, and 2, respectively. After the transformation, their new tags are 0, 1, 2, and 3, respectively.

Second, we have a set of unnecessary RSOps which can be removed from the operator tree. Because the Reduce phase of the optimized plan can have multiple major operators consuming input rows from the shuffling phase, we added *DemuxOperator* (DemuxOp)
in the beginning of the Reduce phase (after the Reducer Driver, which is the entry point of a Reduce phase) to reassign rows to their original tags and dispatch rows to different major operators based on new tags. For example, two mappings tables of the DemuxOp in Figure 3.3 show how DemuxOp reassigns tags and dispatches rows. Then, we remove all unnecessary RSOps in this correlation inside the Reduce phase. Finally, for every

Figure 3.3: The optimized operator tree of the running example shown in Figure 3.2.
GroupByOperator (GBYOp) and JoinOp, we add a MuxOperator (MuxOp) as the single parent. For a GBYOp, its parent MuxOp is used to inform when this GBYOp should buffer input rows and emit output rows. For a parent MuxOp of a JoinOp, besides what a MuxOp does for a GBYOp, its also needs to assign original tags to rows passed to this JoinOp. For example, the MuxOp−2 shown in Figure 3.3 basically forwards rows from the DemuxOp and it needs to assign tags for rows from the GBYOp.

**Operator Coordination**  
Hive inherits the push-based data processing model in a Map and a Reduce task from the MapReduce engine. Because of this execution model, simply removing unnecessary RSOp{s} does not make the plan executable. For example, an operator in a Reduce phase generated by the Correlation Optimizer can have two JoinOps and one is at the upstream of another one. By simply removing the downstream RSOp, this JoinOp will not know when to start buffer its input rows and when to generate output rows. To make the optimized plan work, an operator coordination mechanism is needed.

Currently, this coordination mechanism is implemented in the DemuxOp and MuxOp. When a new row is sent to the Reducer Driver, it checks if it needs to start a new group of rows by checking values of those key columns. If a new group of rows is coming, it first sends the signal of ending the existing row group to the DemuxOp. Then, the DemuxOp will propagate this signal to the operator tree. When a MuxOp gets this ending group signal, it will check if all of its parent operators have sent this signal to it. If so, it will ask its child to generate results and send this signal to its child. After the signal of ending group has been propagated through the entire operator tree in the Reduce phase, the Reducer Driver then will send a signal of starting a new row group to the DemuxOp. This signal will also be propagated through the entire operator tree. Finally, the Reducer Driver will forward the

3.4 Performance Evaluation

3.4.1 Setup

We conducted our experiments in a cluster with 11 nodes launched in Amazon EC2. The instance type was m1.xlarge which has 4 cores, 15 GB memory, and 4 disks. The operating system used for these nodes was Ubuntu Server 12.04.3 LTS 64-bit. The Hadoop version was 1.2.1. The Hive version was Hive 0.14-SNAPSHOT built on Mar. 7th, 2014. This Hadoop cluster had 1 master node running NameNode and JobTracker, and 10 slave nodes running DataNode and TaskTracker.

In our experiments, we used two queries from TPC-DS [26]. TPC-DS is a standard decision support benchmarks. We used the scale factor of 300 for it. In our experiments, all tables TPC-DS were loaded into Hive. For every query used in our experiments, we tested it five times. For elapsed times, we will report medians, and 25% and 50% percentiles will be reported as error bars. To eliminate the impact from OS buffer caches, before every run of a query, we freed all cached objects, and then freed OS page cache, dentries and inodes.

For Hadoop, we set that the Reduce phase starts after the entire Map phase has finished. For a slave node, it can run three concurrent Map tasks or Reduce tasks. The HDFS block size was set to 512 MB. Due to the page limit, we cannot introduce all configuration properties used in Hadoop and Hive, and show queries used in our experiments. Interested readers may refer to https://github.com/yhuai/hive-benchmarks for details.
3.4.2 Results

To show the effectiveness of optimizations introduced in Section 3.3, we show performance of TPC-DS query 27 and query 95. The TPC-DS query 27 first has a five-table star join. Then, the result of this star join is aggregated and sorted. For the star join in this query, it involves a large fact table and four small dimension tables. Without the optimization introduced in Section 3.3.1, the plan of this query has four Map-only jobs and one MapReduce job. Each Map-only job corresponds to a join between the large table and a small dimension table. The last MapReduce job is used to generate the final result. This plan yields poor query evaluation performance because it has four unnecessary Map phases. After eliminating these unnecessary Map phases, the optimized plan has a single MapReduce job. Those Map Joins are executed in the Map phase. Figure 3.4(a) shows the performance of these two plans. The speedup of the optimized plan is around 2.34x.
The TPC-DS query 95 is a very complex query. Because the limitation of Hive on supporting sub-queries in the \texttt{WHERE} clause, we flatten sub-queries in this query for this experiment. This query can be optimized by eliminating unnecessary Map phases and exploiting correlations. Without these two optimizations, the plan of this query has three Map-only jobs and five MapReduce jobs. By exploiting correlations with the Correlation Optimizer, the plan has three Map-only jobs and two MapReduce jobs. By further eliminating unnecessary Map phases, the optimized plan has only two MapReduce jobs. Figure 3.4(b) shows the performance of these three plans. With the Correlation Optimizer, we can achieve a speedup of 2.57x. Then, after further eliminating unnecessary Map phases, we can achieve a combined speedup of 2.92x.

### 3.5 Related Work

Planning a query based on its semantics and data properties can be traced back to System R which cooperates ordering information of intermediate results in a query when choosing the join method [68]. There are several projects that aim to infer and exploit intermediate data properties for optimizing the plan of a query. Representatives are [70] on sorting operations; [52] on partitioning operations; [63] on both sorting and grouping operations; and [77] on partitioning, sorting and grouping operations. To increase effective disk bandwidth and reduce unnecessary operations, both data sharing and work sharing are also exploited in [69] [51] [78] [40].

In the ecosystem of Hadoop, there have been several recent research projects exploiting sharing opportunities and eliminating unnecessary data movements, e.g. [60] [64] [73] [61]. The Correlation Optimizer in Hive is a YSmart-based design. YSmart [60] looks at a single query. It exploits shared data scan opportunities and eliminates unnecessary data
shuffling operations by merging multiple MapReduce jobs into a single one. The main
difference between Correlation Optimizer and other related work (including YSmart) is that
Correlation Optimizer is specifically designed for the push-based data processing model
used by Hive. Existing work mainly focuses on generating optimized query plans. While,
Correlation Optimizer also considers how to execute optimized plans under the push-based
model.

3.6 Summary

In this chapter, we have presented major advancements in Hive. Specifically, we in-
troduced (1) a highly efficient file format, ORC File; (2) an updated query planner that
effectively reduces unnecessary data operations and movements by eliminating unneces-
sary Map phases and exploiting correlations in a query; and (3) a new vectorized query
execution engine that significantly improves the performance of query execution through
better utilizing modern CPUs. The performance and resource utilization efficiency of the
updated Hive have been demonstrated by our experiments. We have also received posi-
tive feedbacks from the Hive-based data processing community. For example, the stor-
age efficiency in Facebook’s Hive production system has been significantly improved after
adopting ORC File [66].

These major technical advancements come from strong collaborative efforts from both
research and development communities. Some academic research projects have directly
influenced the new developments of Hive with strong technical and analytical basis. On
the other hand, the engineering efforts in systems implementation have addressed several
challenges in order to make Hive gain substantial benefits in practice.
Recently, several new features have been added into Hive or under development. Interested readers may refer to [5] for details. From the perspective of functionalities, we have been working on expanding Hive’s SQL capabilities, such as advanced analytic functions, new data types, extending the sub-queries support, common table expressions, and improved support for join syntax. Hive has introduced limited form of support for transactions. Language level support for transaction is under development and is expected to be released later this year. Also, Hive now supports SQL standard based authorization model. HiveServer2 has been enhanced to support different protocols and Kerberos authentication. From the perspective of performance, we have been integrating Hive with Apache Tez [3] for its support on more general query execution plans and better performance. Except eliminating unnecessary Map phases (it is a MapReduce-specific issue), all of advancements introduced in this chapter are still applicable and important to Hive running on Tez. Hive has introduced cost based optimizer. Currently, its used to do join ordering. Work is in progress to use cost based optimizer for wider range of queries.
Chapter 4: Building System Facility for Out-of-band Communications

4.1 Introduction

To effectively handle data with ever-increasing scale, several distributed data processing engines have been developed, such as MapReduce [43], Dryad [57], Spark [76], and Tez [3]. To build an application with a distributed data processing engine, a developer constructs distributed data processing flows based on the programming model provided by this engine, such as MapReduce. A distributed data processing flow comprises two parts, data processing operations and data movements. Essentially, data processing operations define computing tasks, which are executed in different worker machines; and data movements are communications between worker machines generating results and worker machines that consume these results in future. To execute data processing flows, the distributed data processing engine analyzes a submitted flow and breaks it to stages, which contain a series of pipelined operations. Then, the distributed data processing engine executes the given data processing flow in a stage by stage fashion.

The data movement formally defined by a programming model is also called *in-band communications*, such as synchronous data communications in the shuffling phase between a map phase and a reduce phase in MapReduce. A special kind of communications that
is not defined in existing programming models, but is often used in ad-hoc ways by developers, is called *out-of-band communication*, which involves the use of an alternative communication pathway to exchange messages. The out-of-band communication has the following three properties. First, the nature of the communicated data is different and independent from the main in-band data stream and out-of-band communications are used to facilitate the implementation of advanced data processing flows in a productive way. Second, the out-of-band of communication must further support the principle defined by the programming model, such as scalability and fault-tolerance. Finally, the usage of out-of-band communication must result in a significant improvement of the throughput of data processing systems and the productivity of application development.

Fig. 4.1 presents an example to explain important roles of out-of-band-communication in implementing an advanced distributed data processing flow. This query contains a simple operation, which is adding a new column \( r \) to show the numerical rank of every row based on the value of column \( \text{num} \). To evaluate this query in a distributed environment, one practical plan with three stages is shown in Fig. 4.2. For the purpose of simplicity, we assume that there are \( n \) parallel tasks used in every stage. This plan has the following steps.

1. \( n \) parallel tasks conduct a parallel sort operations (stage 1 and stage 2). After the sort, the table \( t \) is distributed in \( n \) partitions with disjoint ranges of \( \text{num} \). The maximum value of \( \text{num} \) in partition \( i - 1 \) is less than the minimal value of \( \text{num} \) in partition \( i \).
2. To generate the rank for every row, we need to know the number of rows appearing before a given row in a given partition \(i\). Thus, for partition \(i\), we need to know the total number of rows appearing in partition 1 to partition \(i - 1\). To generate this information, before ending stage 2, every parallel task counts the number of rows that it has processed. Then, this count is sent to an individual task called “Prefix Sum Task”. This “Prefix Sum Task” internally maintains a vector with \(n\) elements and it stores the row count from parallel task \(i\) in \(i\)th element of the vector. After all tasks in stage 2 finish, the “Prefix Sum Task”, conducts a prefix sum operation on its vector. In the result vector, element \(i\) will be the total number of rows appearing in partition 1 to partition \(i\).

3. In stage 3, a parallel task \(i (i \neq 1)\) takes the \(i - 1\)th element from the result vector of the “Prefix Sum Task” and generate the correct rank for every row that it processes.

For this example, solid lines in Fig. 4.2 are in-band communications, which can be expressed by the programming model of a data processing engine. While, dashed lines in Fig. 4.2 are out-of-band communications, which are hard, if not impossible, to be expressed by a programming model.

The example shown above is a representative use case of out-of-band communications in data processing. Furthermore, out-of-band communications have been commonly used by developers and users in various places, such as the debugging facility proposed for Pig [65], debug mode in Spark SQL [27], statistics collecting mechanisms in Hive [28] and Spark SQL [29], load balancing controller proposed for MapReduce [49], load balancer in Oracle Loader for Hadoop [67], automatic reduce parallelism in Tez [30], scalable execution models for ranking and cumulative window functions in Oracle RDBMS [37], query re-optimization in Scope [36], and dynamic query optimizations in Scope [39] and
DryadLINQ [59]. Despite the fact that out-of-band communications are commonly used, there is little attention on how to effectively and efficiently implement out-of-band communications in a systematic way. In practice, developers and users usually find that there is not enough system support on out-of-band communications, and they have to implement ad hoc solutions, which are hard to reuse, error-prone, and easy to introduce negative side effects on the scalability and fault-tolerance of the execution of distributed data processing flows.

In this work, we focus on out-of-band communications carrying lightweight contents, or lightweight out-of-band communications (in the rest of this chapter, we use the term of out-of-band communications to refer to lightweight out-of-band communications), which
are commonly used in cases mentioned above. Lightweight out-of-band communications are important to improve the development productivity and to implement advanced data processing flows. To support lightweight out-of-band communications in a principled way, we design and implement a system facility called SideWalk. SideWalk serves as the only common information exchange place for senders and receivers of an out-of-band communication. For the senders of an out-of-band communication, they send initial contents for the communication by placing these contents in SideWalk. Then, based on users’ requirements, SideWalk generates the desired contents for the out-of-band communication from the initial contents. For the receivers of an out-of-band communication, they receive the desired contents by read these contents placed in SideWalk.

To design SideWalk, we first abstract common use cases of out-of-band communications. With the abstraction, we design a set of programming APIs with well defined semantics, which can minimize the chances of writing error-prone programs. With these APIs, users can conduct out-of-band communications with reusable programs. Also, based on our abstract, we restrict the communication patterns that users can conduct with SideWalk to prevent them from introducing negative side effects. In summary, the contributions of this chapters are:

1. an abstraction of out-of-band communications based on common use cases;
2. a set of APIs for conducting out-of-band communications in a regulated way;
3. a standalone system facility of out-of-band communications called SideWalk; and
4. a set of case studies that show the effectiveness of our abstraction of out-of-band communications, programming APIs, and SideWalk.
The rest of this chapter is organized as follows. Section 4.2 introduces the background of this work. Specifically, we introduce distributed data processing engines and out-of-band communications in this section. Section 4.3 presents the programming abstraction of SideWalk. In this section, we first propose our abstraction of out-of-band communications. Then, we present SideWalk’s core programming APIs designed based on our abstraction of out-of-band communications. Section 4.4 presents three case studies, which demonstrate how to use SideWalk to implement out-of-band communications for various use cases. Section 4.5 introduces the implementation of the SideWalk prototype. Section 4.6 presents our evaluation of SideWalk, which illustrates the effectiveness of SideWalk. Section 4.7 surveys existing efforts on supporting out-of-band communications. Section 4.8 summarizes this chapter.

4.2 Background

In this section, we review distributed data processing engines and out-of-band communications. At a high level, a distributed data processing engine abstracts communications between tasks generating data and tasks consuming data through data processing operations provided by its programming model. Thus, the process of constructing a distributed processing flow also defines needed communications, or in-band communications. In practice, there are out-of-band communications that cannot be described by the programming model of a data processing engine and these communications need to be handled differently from in-band communications.

4.2.1 Distributed Data Processing Engines

For a distributed data processing engine, it generally has three components:
1. A programming model comprises a set of data parallel operations, e.g., map, filter, and reduce, through which users can construct distributed data processing flows by combining different operations.

2. A scheduler that breaks the distributed data processing flow to executable stages and manages the execution of stages, e.g., distributing tasks of a stage to worker machines, handling stragglers and task failures.

3. A task executor running in every worker machine that executes assigned tasks by fetching input data items (communications) and conducting local computing work.

Through these three components, users can write programs for processing large datasets in a distributed environment without worrying about complications like task scheduling, fault-tolerance, load balancing, and data transfers.

### 4.2.2 Out-of-band Communications

In addition to communications defined in operations of a programming model, in various cases, users and developers use out-of-band communications to facilitate the development of their systems and applications, as well as to implement advanced data processing flows that cannot be solely expressed by a given programming model. For example, to debug a distributed data processing flow, users usually need to gather the record triggering a runtime exception to understand the root cause of this exception. However, the communication pattern of passing a such kind of record to the user is beyond the capability of existing programming models, which only define data processing operations and related communications.
Because of the limited programming and system support, developers and users usually find that they have to manually implement out-of-band communication. These ad hoc solutions introduce three kinds of problems, which are:

1. **Low productivity**: Ad hoc solutions are hard to reuse and port to a case that is different from the original scenario. It is also common that different developers have to build the solution for almost the same kind of out-of-band communications because they do not know others’ existing solutions or others’ solutions are not reusable.

2. **Error-prone solutions**: To build an ad hoc solution, users need to understand how the underlying data processing engine work to build a correct solution. For example, users need to know how to handle stragglers and task failures in order to filter out duplicate initial contents of an out-of-band communication.

3. **Negative side effects**: To build an ad hoc solution, users need to decide what communication patterns are allowed and how to deliver contents of out-of-band communications. A naive solution can significantly hurt the scalability of users’ programs. For example, using HDFS [1] to delivery contents of lightweight out-of-band communications can significantly slow down the execution time of a MapReduce job because of the high overhead of HDFS on handling small files.

### 4.3 The SideWalk Programming Abstraction

In this section, we first present the SideWalk abstraction of lightweight out-of-band communication, including how the contents of an out-of-band communication is generated and the how the contents are distributed to the receiver of the communication. Then, we introduce programming APIs of SideWalk.
4.3.1 The Abstraction of Out-of-band Communications

Based on our observations, we abstract lightweight out-of-band communications as a sequence of operations shown in Fig. 4.3, which are Gather, Transform, and Distribute. At a high level, to conduct a lightweight out-of-band communication, initial lightweight contents generated from $n$ parallel senders are gathered to a common place and then transformed to the desired lightweight contents. Finally, the desired contents are distributed to $k$ receivers.

Gather

The gather operations describes the way that the initial contents of an out-of-band communication are collected to a common place. The initial contents are generated by $n$ parallel senders and every parallel sender generates one part of the initial contents. These $n$ parts are organized in a vector with $n$ elements called initial content vector, and they are indexed by the order of senders’ identifiers. In practice, senders can either be tasks in a stage or the
driver program of the distributed data processing flow (in this case, there is only a single sender). For the former case, an example is that tasks in a stage are sending out the number of records processed by a certain operation. For the latter case, an example is that a user sends a list of keys that will be used to filter out necessary records. Tasks from different stages are not allowed to be senders for a single out-of-band communication because they cannot be considered as parallel senders (two stages are generally executed in different time).

**Transform**

The transform operation describes the way that the desired contents of an out-of-band communication is generated from the initial contents. Basically, the transform operation applies a function $f$ to the initial contents represented by a vector with $n$ elements (initial content vector) and generates the desired contents represented by a vector with $M$ elements called desired content vector, i.e.

$$ [o_1, o_2, ..., o_m] = f([i_1, i_2, ..., i_n]). $$

The representative examples of function $f$'s are an identity function, an average function, and a prefix sum function.

**Distribute**

The distribute operation describes the way that desired contents of an out-of-band communication are sent to $k$ receivers. In practice, receivers are usually tasks of a stage or the driver program of the distributed data processing flow (in this case, there is only a single receiver). For a receiver, it can choose to receive the entire vector generated from the transform operation or to receive certain elements of the vector. For the former case, an example
is that every receiver gets a list of keys that is used to filter out unnecessary records. For the latter case, an example is that to evaluate the rank function shown in Fig. 4.1, task \( i \) in stage 3 shown in Fig. 4.2 should get the total number of records from the partition 1 to partition \( i - 1 \) generated by the stage 2. In this case, task \( i \) only gets the \( i \)th element from the desired content vector of the out-of-band communication shown in Fig. 4.2.

### 4.3.2 Programming APIs

SideWalk has two kinds of APIs, one for managing out-of-band communications and one for conducting out-of-band communications.

There are three APIs for managing out-of-band communication, which are

- **register(name, itype, otype, func)**: Registers an out-of-band communication with the name of name. The type of elements of the initial contents is itype and that of elements of the desired contents is otype. The function to transform the initial contents to the desired contents is func.

- **isReady(name)**: Returns true if the desired contents of the out-of-band communication named as name is read to be consumed by receivers.

- **clear(name)**: Deletes the out-of-band communication name. It is used when the desired contents of this out-of-band communication has been consumed by receivers.

There are five APIs for conducting an out-of-band communications. For the gather operation, the following two APIs are defined

- **report(name, data)**: For the out-of-band communication name, reports an element of the initial contents represented by data to SideWalk. This element will be placed in the vector representing the initial contents and the position of this element in the vector will be derived automatically from the context.
• **report**(name, data, index): For the out-of-band communication name, reports an element of the initial contents represented by data to SideWalk. This element will be placed in the vector representing the initial contents and the position of this element is index.

For the transform operation, the following API is defined.

• **prepare**(name): Invokes the transformation function func registered for the out-of-band communication name to transform the initial contents to the desired contents.

For the distribute operation, the following two APIs are defined.

• **query**(name): Retrieves the entire desired contents for the out-of-band communication name.

• **query**(name, index): Retrieves indexth element from the vector representing the desired contents of the out-of-band communication name.

The way that we complete the distribute operation is to allow receivers read the desired contents from SideWalk. This choice significantly simplified the design of SideWalk because it only needs to serve receivers’ requests instead of knowing the status of receivers of every out-of-band communication and actively pushing desired contents of out-of-band communications to corresponding receivers.

### 4.4 Case Studies

In this section, we use three cases to demonstrate how to use SideWalk. For every case, we first provide an overview of the case and what out-of-band communications are needed. Then, we present how to use SideWalk to implement the out-of-band communications described in the case.
SELECT
  t1.key,
  t2.key
FROM t1 JOIN t2 ON (t1.sk = t2.sk)
WHERE
  t2.key >= 100 AND
  t2.key <= 1000

Figure 4.4: An example query used in Section 4.4.1.

4.4.1 Semi-join Reduction

Use Case Description

This case is about a join operation between two table t1 and t2. Fig. 4.4 shows the query used in this case. Before the join operation, two predicates are applied to rows of t2. In this case, the column of t2.key and t2.sk are correlated and the value of t2.sk is a function of t2.key, i.e. t2.sk = f(t2.key). Also, the function f is a monotonic function.

If t1 and t2 are large, a shuffled join operation needs to be used as the physical join operation, which shuffle rows of two tables based on values of columns in the join condition t1.sk = t2.sk. Then, all rows with the same value of sk will be grouped in a single task, and this task can conduct the join operation locally. To efficiently evaluate this query, the idea of semi-join reduction [38] can be used to optimize the query plan. One useful optimization is to first evaluate the table t2 with two predicates and collect the range of needed values of t2.sk, which will be used to filter out necessary rows from t1. When predicates on t2 is selective, the range of needed values of t2.sk can be tight and lots of unneeded rows in t1 can be filtered out, which can significantly reduce the time of data transfers for the following join operation. With this optimization, the query plan will be the one shown in Fig. 4.5. It basically has three stages:
1. In the first stage, concurrent tasks scan rows of \( t_2 \) and apply predicates \( t_2.\text{key} \geq 100 \text{ AND } t_2.\text{key} \leq 1000 \). Needed rows of \( t_2 \) are materialized for shuffling operation used by the join operation in the stage 3. During the table scan, the maximum and minimum values of \( t_2.\text{sk} \) are also derived.

2. In the second stage, concurrent tasks scan rows of \( t_2 \) and apply predicates to check if a \( t_1.\text{sk} \) is within the range of \( t_2.\text{sk} \) derived from the first stage. Needed rows of \( t_2 \) are materialized for shuffling operation used by the join operation in the stage 3.

3. Needed rows of \( t_2 \) and \( t_1 \) generated from the first and second stage are partitioned by the value of \( \text{sk} \) and a join operation is executed in concurrent tasks of stage 3. Finally, a select operation is used to generate the final result.

**Using SideWalk**

SideWalk is used to implement the out-of-band communications in the above query plan. The out-of-band communications for this case is to (1) gather maximum and minimum values from tasks of stage 1, (2) merge these local maximum and minimum values to generate the range of \( t_2.\text{sk} \), and (3) distribute the range of \( t_2.\text{sk} \) to tasks in stage 2. In Fig. 4.5, the dashed line from stage 1 to stage represent out-of-band communications. The filter operation in stage 1 is an operation that is not in the original query and is constructed based on the range of \( t_2.\text{sk} \).

In this case, SideWalk is used as follows.

1. Before the execution of stage 1, `register` is called to register two out-of-band communications, one for the maximum value of \( t_2.\text{sk} \) and one for the minimum value of \( t_2.\text{sk} \). The transformation functions used are \texttt{Max} and \texttt{Min}, respectively.
2. In stage 1, every task tracks the maximum and minimum values of \( t_2 \).sk and at the end of the task, it calls \texttt{report} to send its local maximum and minimum values of \( t_2 \).sk to SideWalk.

3. Before the execution of stage 2, \texttt{prepare} is called to invoke the transformation functions for these two out-of-band communications to generate the maximum and minimum values of \( t_2 \).sk for all rows generated by stage 1.

4. In stage 2, every task calls \texttt{query} to retrieve the maximum and minimum values of \( t_2 \).sk and then initialize the filter operation to filter out unneeded rows of \( t_1 \).

5. After this query finishes, \texttt{clear} is called to delete two registered out-of-band communications.
4.4.2 Evaluating Window Functions

Use Case Description

The second case is the query shown in Fig. 4.1 in Section 4.1. The \texttt{rank} function shown in the query is a typical window function [31]. Window functions, introduced as a part of SQL 2003 [44], add new analytical capabilities to SQL and have been widely adopted. Because a window function operates on a \textit{window} of rows (a set of rows with a start and end boundary), evaluating a window function often introduce challenges when the size of a window is too large to fit in a single worker machine. The query in Fig. 4.1 shows such a window function and the window for \texttt{rank} covers all rows of the table \texttt{t}. To efficiently evaluate this query in a distributed environment, the plan described in Section 4.1 is used. Out-of-band communications are a critical part of to implement this plan.

Using SideWalk

With SideWalk, the out-of-band communication from the stage 2 to stage 3 shown in Fig. 4.1 is implemented as follows.

1. Before the execution of stage 2, \texttt{register} is called to register an out-of-band communication from the stage 2 to stage 3. The transformation function used is \texttt{Prefix Sum}.

2. In stage 2, every task tracks the number of records it has processed and at the end of the task, it calls \texttt{report} to send its local record count to SideWalk. For task \textit{i} in stage 2, its local record count is placed in the \textit{i}th position of the initial content vector for the out-of-band communication.

3. Before the execution of stage 3, \texttt{prepare} is called to invoke the transformation function generate the prefix sums based on the initial content vector.
4. In stage 3, for task 1, it does not need to contact SideWalk, because there is no row before the first row that this task has. For task \( i \) \((i \neq 1)\), it calls query to retrieve the \( i - 1 \)th element from the desired content vector storing prefix sums from SideWalk and we call this value offset\(_i\). Then, for every record in a task \( i \) processing a partition, the final value of rank is calculated by adding offset\(_i\) with the rank value of this record in this task. For example, for a record processed by task \( i \), if the offset\(_i\) is 100 and this record is the 10th record of all records processed by task \( i \), its final rank value will be 110.

5. After this query finishes, clear is called to delete the registered out-of-band communication.

### 4.4.3 Debugging Distributed Data Processing Flows

#### Use Case Description

This case is about debugging distributed data processing flows. In the development of data processing systems and applications, users and developers often find that it is hard to debug their programs because their programs are executed in multiple worker machines. Debugging tools for distributed programs are not as effective as those for single machine programs, and useful debugging information is separated in different places, which significantly add burdens to users and developers to locate components introducing bugs.

We show the way that uses SideWalk to help debugging a data processing flow. Specifically, the case to be presented here is based on the first case shown in Section 4.4.1. In our first case, we have introduced how to use SideWalk to optimize the query shown in Fig. 4.4. If we find that the implementation of the optimized query plan generates wrong result, it is intuitive to first check if we correctly implemented the optimization. To do so, we can compare the number of records generated by the join operation in the stage 3 between the
optimized and non-optimized query plans. Basically, while the execution of the optimized and non-optimized query plans, we need out-of-band communications to pass out the number of records generated by the join operation in every task (shown in Fig. 4.6). Also, to be able to find tasks that generate the wrong number of results, it is necessary for a user to get all record counts from all tasks for the stage 3 instead of just at an aggregated record counts.
Using SideWalk

SideWalk is used to implement the out-of-band communication passing out the number of records generated by the join operation for every task in the stage 3. This out-of-band communication is added to both optimized and non-optimized query plans, so the user can check which tasks generate the wrong number of records. In this case, SideWalk is used as follows.

1. Before the execution of stage 3, `register` is called to register an out-of-band communication for passing out record counts. The transformation functions used is an identity function because we want to be able to look at a record count from an individual task.

2. In stage 3, every task tracks the number of records generated by the join operation. At the end of the task, it calls `report` to send it local record count to SideWalk.

3. Before the execution of stage 3, `prepare` is called and because the transformation function is an identify function, the initial contents of the out-of-band communication are the desired contents.

4. The user calls `query` to retrieve the entire desired contents of the out-of-band communication, which is a vector storing all record counts. Then, this user then check these record counts derived from both optimized and non-optimized plans to find which tasks generate the wrong number of record counts.

5. After the user checks these record counts, `clear` is called to delete the registered out-of-band communication.

After the user find which tasks generate the wrong number of records, he or she can start to analyze the root cause of this problem and determine what to do next. For example, if
he or she determines the problem is caused by the implementation of the optimization, it is possible that the range of $t_{2}.sk$ generated in the stage 1 was wrong.

4.5 The Implementation of SideWalk

In this section, we introduce the implementation of SideWalk. We first introduce its software architecture and then introduce mechanisms to minimize possible negative side effects that can be introduced when out-of-band communications are implemented in an ad hoc approach.
4.5.1 Software Architecture

Fig. 4.7 shows the software architecture of SideWalk. SideWalk has two kinds of nodes, storage nodes and computing nodes. A storage node is used as the place to store initial and desired contents of out-of-band communications as well as various configuration variables associated with an out-of-band communication, such as the data type of elements in the initial content vector, the data type of elements in the desired content vector, and the class name of the transformation function. A computing node is used to evaluate the transformation function for an out-of-band communication by reading the initial contents from storage nodes, applying the transformation function, and writing the desired contents to storage nodes. For a SideWalk client, it will contact computing nodes for invoking prepare and contact storage nodes for other API calls.

We built our prototype to work with data processing engines on top of HDFS. Because HDFS is not efficient on handling lightweight data, we need to use a different storage system to implement storage nodes. In our prototype, we implemented storage nodes based on ZooKeeper [56], which can efficiently handle lightweight data in a scalable and fault-tolerant manner.

4.5.2 Minimizing Negative Side Effects

One important goal of SideWalk is to let users implement out-of-band communications without worrying about introducing potential negative side effects on the scalability and fault-tolerance of the execution of distributed data processing flows. At here, we summarize typical causes of negative side effects and the way that SideWalk avoids these causes.

Through our analysis and observation, there are four kinds of causes of negative side effects, which are:
1. The solution for out-of-band communications cannot be scaled to match the scale of distributed data processing flows and the number of out-of-band communications, e.g. the solution was not designed to be able to scale or the storage system used for lightweight out-of-band communications has high overhead on handling lightweight data. For this case, when the scale of distributed data processing flows and the number of out-of-band communications is getting large, the overhead of conducting out-of-band communications can be significant.

2. The solution for lightweight out-of-band communications are used to pass large datasets. For this case, because the solution of out-of-band communications basically is not designed for handling large datasets, the performance of data processing flows passing large datasets in out-of-band communications will not be acceptable.

3. Complex communication patterns and task dependencies can be introduced through out-of-band communications. For this case, users may introduce complex task dependencies in a stage, e.g. point-to-point communications, which hurts the scalability of this stage.

4. The solution for out-of-band communications is not fault-tolerant and its failure can make the execution of a distributed data processing using it not be able to finish. For this case, if the solution for out-of-band communications is failed, all data processing flows using out-of-band communications will not be fault-tolerant.

In the implementation of SideWalk, the above issues are addressed as follows. To accomplish scalability, for storage nodes, we implemented them based on a scalable storage system, which is ZooKeeper. For computing nodes, we implemented them as independent RPC servers. Basically, when a client invokes \texttt{prepare}, it randomly chooses a computing node and the selected computing node will do all of the work of the \texttt{prepare} call. To
ensure the costs of using SideWalk will not hurt the scalability of the execution of data processing flows, it is importnat that storage nodes should efficiently handle lightweight data items. Thus, systems like HDFS are not suitable because they are not designed to efficiently handle small datasets. Because we implemented storage nodes based on ZooKeeper, storage nodes are suitable to handle lightweight data items and we can ensure that the costs of using SideWalk will not hurt the scalability of the execution of data processing flows.

To ensure that users cannot use SideWalk to pass large datasets, we set size limit to elements in the initial and desired content vectors. The default size limit is 1 MB. Thus, for an out-of-band communication, a sender cannot use report to send data larger than the size limit.

SideWalk only supports regulated communication patterns based on our abstraction presented in Section 4.3. For an out-of-band communication, SideWalk does not allow that senders and receivers of this out-of-band communication in the same stage. Thus, users cannot introduce task dependencies hurting the scalability of the execution of data processing flows in a stage through SideWalk.

We implemented SideWalk as a system which can tolerant partial node failures. For storage nodes, because our implementation is based on ZooKeeper, for a SideWalk instance with $2f + 1$ storage nodes, it can tolerate $f$ storage node failures. For computing nodes, in the presence of node failure, a client basically randomly selects another computing node and retries the prepare call. So, for a SideWalk instance with $k$ computing nodes, it can tolerate $k - 1$ computing node failures.
4.6 Evaluation

To show the benefits of using out-of-band communications and demonstrate the effectiveness of SideWalk in a quantitative way, we have conducted experiments with workloads based on cases introduced in Section 4.4.1 and Section 4.4.2. In this section, we report results of our evaluation.

In our evaluation, we used Amazon EC2 as our platform. The machines we used were large VMs (m1.large), each of which had 2 virtual cores with 2 EC2 Compute Units each, 7.5 GB memory and high performance I/O. We used Hadoop MapReduce (version 0.20.2) as the data processing engine in our experiments. For HDFS, 256 MB was used as the block size.

4.6.1 Semi-join Reduction

The first experiment is based on the case introduced in Section 4.4.1. We used SideWalk to implement the out-of-band communications for the semi-join reduction optimization. In this section, we compare the results of query execution times with and without the optimization.

Workload

In this experiment, we used three queries from the TPC-H benchmark [14], which were query 3, query 5, and query 10. These queries are shown in Appendix ???. To demonstrate the optimization introduced in Section 4.4.1, we used a modified TPC-H dataset with a scale factor 100 (around 100 GB raw files). We changed the method to generate o_orderkey in the table of order. In the original TPC-H dataset, values of o_orderkey and o_orderdate in the table of order are generated independently. In
our dataset, \texttt{o\_orderkey} is generated based on the value of \texttt{o\_orderdate} and the value of \texttt{o\_orderkey} is increased when the value of \texttt{o\_orderdate} is increased.

The implementations of these three queries on MapReduce are summarized as follows.

1. Query 3: This query needs four MapReduce jobs. The first job is to execute the join between \texttt{customer} and \texttt{orders}. Then the second job is to execute the join between the result of the first job and \texttt{lineitem}. Finally two jobs are used to execute aggregation and sorting.

2. Query 5: This query needs five jobs. The first job is to execute the join between \texttt{customer} and \texttt{orders}. The second job is to execute the join amount \texttt{nation}, \texttt{region}, and \texttt{supplier}. Then the third job is to execute the join among the results of the first and the second job and \texttt{lineitem}. Finally two jobs are used to execute aggregation and sorting.

3. Query 10: The query needs four jobs. The first job is to execute the join amount \texttt{nation}, \texttt{customer}, and \texttt{orders}. The second jobs is to execute the join between the result of the first job and \texttt{lineitem}. Finally two jobs are used to execute aggregation and sorting.

For these queries, the join condition between the table \texttt{order} and \texttt{lineitem} is \texttt{o\_orderkey = l\_orderkey}. To implement the optimization, when the semi-join reduction is enabled, the range of \texttt{o\_orderkey} will be passed out through out-of-band communications when \texttt{order} is being scanned. Then, the range of \texttt{o\_orderkey} will be used to help the scan operation of \texttt{lineitem} to filter out rows with unneeded \texttt{l\_orderkey}. 
Figure 4.8: The query execution times for three queries with and without the optimization.

<table>
<thead>
<tr>
<th>Query</th>
<th>Default</th>
<th>Optimized</th>
<th>Reduction Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q3</td>
<td>6,653,166,996</td>
<td>576,817,912</td>
<td>91%</td>
</tr>
<tr>
<td>Q5</td>
<td>3,938,321,949</td>
<td>100,193,012</td>
<td>97%</td>
</tr>
<tr>
<td>Q10</td>
<td>4,178,385,064</td>
<td>1,145,410,062</td>
<td>73%</td>
</tr>
</tbody>
</table>

Table 4.1: The size of records from \texttt{lineitem} emitted for shuffling (bytes).

Setup

For the cluster used in this experiment, it had 10 Hadoop slave nodes and 1 master node. The SideWalk was running on a Micro Instance that had 613 MB memory, and up to 2 EC2 Compute Units (for short periodic bursts).
Result

Fig. 4.8 shows the execution times of three queries with and without the semi-join reduction optimization. Because records from lineitem need to be shuffled to execute the join operation between lineitem and other tables. With the optimization, the size of records from lineitem emitted for shuffling is significantly reduced (Table 4.1) and the reduction ratios are 91%, 97%, 73% for these three queries, respectively. For the jobs that take the table lineitem as the input, with the optimization, 1.95x speedup was achieved for the job 2 of query 3; 1.51x speedup was achieved for the job 3 of query 5; and, 1.69x was achieved for the job 2 of query 10.

For the overall execution times of these three queries, because of the reduced amount of data from lineitem for shuffling, we achieved 1.42x, 1.25x, and 1.23x speedup, respectively. Through this experiment, we can see that out-of-band communications can be used to help users and developers build advanced data processing flows with better performance. Without SideWalk, this kind of optimizations can be hard to implement because users and developers usually need to build their ad hoc solutions.

4.6.2 Evaluating Window Functions

The second experiment is based on the case introduced in Section 4.4.2. We aim to use results of this experiment to show the effectiveness of SideWalk on handling a large number of out-of-band communications with a large number of tasks and also show the ineffectiveness of using HDFS to implement out-of-band communications. Despite the fact that HDFS’s high overhead on handling lightweight files, users still commonly use HDFS as the way to implement their ad hoc solutions for out-of-band communications because there does not exist an effective solution for out-of-band communications.
SELECT
  employee_id,
  employee_name,
  department,
  age,
  level,
  salary,
  avg(salary) OVER (PARTITION BY department),
  avg(salary) OVER (PARTITION BY age),
  avg(salary) OVER (PARTITION BY level)
FROM
  emp_table
ORDER BY
  employee_id;

Figure 4.9: The SQL query with window functions

Workload

In this experiment, we used the query shown in Fig. 4.9. This query has a single window function `avg` with three different kinds of window definitions. The function `avg` calculate the average value for every window. The three kinds windows are defined based on values of `department`, `age`, `level`. Basically, rows with the same value of `department/age/level` are grouped in the same window.

Because the number of distinct values for these three columns are usually small, we can optimize this query by using a single MapReduce job to evaluate it. The query plan is shown in Fig. 4.10. At a high level, the map phase of the MapReduce job is the stage 1 and the reduce phase of the job is the stage 2. The shuffling operation between these two stages sorts the rows of the table `emp_table` based on the value of `employee_id`. Now, we use the column of `age` to show how to evaluate the window function. Basically, every mapper tracks the sum and number of values of `salary` for every distinct value
of age. At the end of the mapper, it sends out information described above through out-of-band communications. After the finish of the map phase, for every distinct number of age, the average of salary is calculated by merging information from all mappers. In the reduce phase, every reducer first gets the average of salary for every distinct value of age through out-of-band communications, and then it attaches this value to the row it gets to generate the final result.

In our experiment, we intentionally created a large number of out-of-band communications to stress SideWalk and to show why HDFS-based out-of-band communications are not negative side effect free. We used 41 distinct age values, 41 distinct dept values
Table 4.2: Three scales used in the experiment of running the query shown in Fig. 4.9

and 31 distinct level values. To count the average value, both record counts and sum of salary are needed. Thus, in total, for the query shown in Fig. 4.9, 226 out-of-band communications were created. We want to note that in practice, the number of out-of-band communications used in a data processing flow is generally lower than this number.

Setup

For this experiment, we tested execution times of the plan described above in three scales, which are summarized in Table 4.2. For the largest scale, there were 160 mappers and 80 reducers contacting with SideWalk and the total number of records of the table emp_table was 896 million. For every scale, we tested implementations with HDFS-based out-of-band communications and SideWalk-based out-of-band communications.

Result

Fig. 4.11 shows query execution times with HDFS-based out-of-band communication implementation and SideWalk’s out-of-band communication implementations with three scales. We can see that the query implemented with SideWalk always finishes faster than the query implemented with HDFS-based out-of-band communications. The execution times for the query implemented with SideWalk’s out-of-band communications are 248s, 276s, and 400s for three scales. While, the execution times for the query implemented with HDFS-based out-of-band communications are 284s, 373s, and 611s for three scales. The
Figure 4.11: The query execution times for two implementations of out-of-band communications with three scales. HDFS means that out-of-band communications were implemented by using HDFS. SideWalk means that out-of-band communications were implemented by using SideWalk.

execution times of the query plan with SideWalk’s out-of-band communications are 1.14, 1.35, and 1.53 times as fast as execution times of the query plan with HDFS-based out-of-band communications, respectively. Also, it is clear that the implementation with HDFS-based out-of-band communications has a worse scalability than that with SideWalk’s out-of-band communications. The reason of this difference is that SideWalk uses a storage system (ZooKeeper in our prototype) suitable for handling lightweight data items with a lower overhead.
4.7 Related Work

Although out-of-band communications are commonly used in practice, work on how to implement out-of-band communications is limited. In this section, we summarize work that aims to provide a systematic way to support out-of-band communications.

MapReduce (and also Hadoop) framework provides a Counter facility to count the occurrences of events [43]. With this facility, Map/Reduce tasks periodically report counter values to the master node (i.e. JobTracker). When the MapReduce job finishes, the master will sum values of a counter from successful tasks and return the global value of this counter to the user. In the Counter facility, the desired contents of an out-of-band communications can be only generated from a summation of numbers, which significantly limits the cases that can be supported by this facility. For example, the out-of-band communication presented in Fig. 4.2 cannot take advantage of the Counter facility because the transformation function is a prefix sum function. Also, since the Counter facility is not designed as a general-purpose solution for out-of-band communications, there is not a system support to limit its potential negative side effects on the execution of data processing flows when the capability of this facility is abused. Users have to use counters carefully with expert experience [32].

Another work on building a systematic support of out-of-band communications is Inspector Gadget [65], which is a framework for custom monitoring and debugging in Pig [47]. It instruments monitoring agents into an execution flow and setups a coordinator outside of the flow. The coordinator is used to relay and summarize messages received from monitoring agents. Because Inspector Gadget is mainly designed for monitoring and debugging purpose, it does not cover use cases of out-of-band communications other than
monitoring and debugging. Thus, users need other solutions for cases that cannot be supported by Inspector Gadget. While, SideWalk is designed as a general-purpose facility for out-of-band communications. Users can use SideWalk to implement different out-of-band communications for different cases. Also, Inspector Gadget mainly focuses on providing useful functions for users in the monitoring and debugging scenarios and how to avoid negative side effects was not the focus of this work. In SideWalk, we carefully discussed how to avoiding typical negative side effects and implemented SideWalk accordingly.

In Spark [76], accumulators and broadcast variables [33] are shared variables to implement out-of-band communications. A user uses an accumulator by assigning an initial value to it and define a commutative and associative “add” operation to update the value of this accumulator. Tasks that need to update an accumulator will first apply the “add” operation locally and then send the result to the driver to generate the global value. Accumulators can be viewed as an implementation of the Gather and Transform operations defined in our abstraction of out-of-band communications (Section 4.3.1). A broadcast variable is basically a variable that will be broadcasted as a read-only copy to tasks. Broadcast variables can be viewed as an implementation of the Distribute operation defined in our abstraction of out-of-band communications (Section 4.3.1). SideWalk is different from Spark accumulators and broadcast variables in two aspects. First, SideWalk is implemented as a standalone facility that is independent from data processing engines. While, Spark accumulators and broadcast variables are implemented as built-in components in Spark. Second, Spark accumulators and broadcast variables are designed to support out-of-band communications within a live driver session. While, because SideWalk stores initial and desired contents in a storage system (i.e. ZooKeeper), SideWalk can support out-of-band communications
between different driver sessions. Also, because users need to use `clear` to delete an out-of-band communication, destroying a driver session will not delete existing out-of-band communications implemented by SideWalk.

### 4.8 Summary

In this chapter, we have introduced SideWalk, a facility for lightweight out-of-band communications. SideWalk was designed based on our abstraction of out-of-band communications. Using APIs provided by SideWalk, users can easily write reusable programs to implement out-of-band communications. The semantics of these APIs are well defined, which help users to correctly implement out-of-band communications. To minimize negative side effects, we restricted communication patterns that can be done through SideWalk and limit the sizes of data items that can be transferred through SideWalk. Our evaluation has showed that using out-of-band communications can benefit the performance data processing flows and the effectiveness of SideWalk on handling out-of-band communications.
Chapter 5: Conclusions and Future Work

In this dissertation, we have addressed three critical problems of building data analytics systems with our proposed methods and their system implementations. The work on the basic structure and essential issues of table placement methods provides a comprehensive way to understand how to store table data in distributed filesystem. Also, this work provides practical guidelines based on the basic structure of table placement methods and we have seen that ORC and Parquet, two state-of-the-art table placement methods storing data for many companies and organizations providing world wide services have been implemented consistently with our guidelines. The work of generating efficient distributed query plans by reducing unnecessary data movements introduced two practical optimizations to significantly improve the execution time of complex queries. These two optimizations have been adopted by Apache Hive, which is a widely used Hadoop-based data warehousing systems in production. The work of SideWalk provides a system facility for conducting out-of-band communications. SideWalk abstracts common use cases of out-of-band communications. For various use cases, users can use SideWalk to implement out-of-band communications. Also, based on the abstraction of out-of-band communications, SideWalk restricts the communication patterns that users can conduct to prevent them from introducing negative side effects.
Following these three projects, there are three directions of future work. First, based on the study of basic structure and essential issues of table placement methods, a tool can be designed to determine the table placement methods for every table based on its characteristics, such as the number of columns and data types of these columns. Also, data compression is an important technique to improve the data storage efficiency. The impacts of data compression on the design choices of table placement methods is another topic worth studying. Second, following the work of optimizing distributed query plans, for a complex query, it is possible to optimize this query through different schemes of data movements. Considering that datasets processed by data analytics systems are commonly raw data and usually lack of data statistics, how to determine what data movement scheme to choose is an issue that needs to be addressed. Finally, we will work on integrating SideWalk with existing high level programming environments. Through this way, existing high level programming environments can use out-of-band communications in a unified way, and more efficient and advanced data processing flows can be implemented with the help of out-of-band communications.

The data explosion demands that data processing system infrastructure be restructured to be scalable in low cost without changing existing user interface. The first stage of system development has focused on building such systems based on a scale-out model. This dissertation has contributed to this development in both basic research and experimental development. The current and future research and development direction for large scale data processing systems under the scale-out model is to accelerate execution speed with low latency response time. Several promising development tasks are being conducted. One is in-memory computing that enables users to always access data in DRAM memory in order to respond real time data processing demands. Another one is to merge the advanced
storage devices into systems, such as solid state based flash memory, for the same purpose. Researchers are also making efforts to integrate GPU (Graphics Processing Unit) into large scale data processing systems to exploit massive data parallelism at an ultra-high speed. This dissertation has made a foundation for us to continue our research efforts for all above mentioned activities.
Bibliography


