LLAMA: LEVERAGING COLUMNAR STORAGE FOR SCALABLE JOIN PROCESSING IN MAPREDUCE

Lin Yuting

Bachelor of Science
Sichuan University, China

A THESIS SUBMITTED
FOR THE DEGREE OF MASTER OF SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2011

Acknowledgements

First of all, I would like to thank my supervisor Beng Chin Ooi for guidance and instructions throughout these years. I am fortunate to be his student and I am influenced by his energy, his working attitude, and his emphasis on attacking the real-work problems and building the real system.

I would also like to thank Divyakant Agrawal, who helped me to edit the paper with lots of valuable comments. Collaboration with him has been a true pleasure. I am influenced by his interpersonal skills and his passion.

I thank all members in the database group in School of Computing, National University of Singapore. It is an excellent and fruitful group, providing me a good research environment. Especially to Sai Wu, who took me under his wing from the day I entered the lab, helping me to figure out how to do the research; to Dawei Jiang, who is always willing to share his insightful opinions and giving me constructive feedback; to Hoang Tam Vo, who worked closed with me on the projects, giving me suggestions and assistance in these days.

Finally, and most importantly, I am indebted to my parents, who give me a warm family with never-ending support, care and encouragement in my whole life. All of these help me finish my academic study and complete this research work.

Contents

Abstract

To achieve high reliability and scalability, most large-scale data warehouse systems have adopted the cluster-based architecture. In this paper, we propose the design of a new cluster-based data warehouse system, *Llama*, a hybrid data management system which combines the features of row-wise and column-wise database systems. In Llama, columns are formed into correlation groups to provide the basis for the vertical partitioning of tables. Llama employs a distributed file system (DFS) to disseminate data among cluster nodes. Above the DFS, a MapReduce-based query engine is supported. We design a new join algorithm to facilitate fast join processing. We present a performance study on TPC-H dataset and compare Llama with Hive, a data warehouse infrastructure built on top of Hadoop. The experiment is conducted on EC2. The results show that Llama has an efficient load performance and its query performance is significantly better than the traditional MapReduce framework based on row-wise storage.

List of Tables

List of Figures

Chapter 1

Introduction

In the era of petabytes of data, processing analytical queries on massive amounts of data in a scalable and reliable manner is becoming one of the most important challenges for the next generation of data warehousing systems. For example, about 16 petabytes of data are transferred through AT&T's networks everyday [1], and a petabyte of data are processed in Google's servers every 72 minutes [11]. In order to store such big data and response various of online transaction queries, several "NoSQL" data stores [27] are developed by different web giants, such as BigTable [28] in Google, Dynamo [36] in Amazon, and PNUTS [33, 66] in Yahoo!. HBase [5] and Cassandra [51, 52] are the open source implementations of BigTable and Dynamo under the umbrella of Apache. At the same time, special-purpose computations are usually needed for extracting valuable insights, information, and knowledge from the big data collected by these stores. Given the massive scale of data, many of these computations need to be executed in a distributed and parallel manner over a network of hundreds or even thousands of machines. Furthermore, often these computations involve complex analysis based on sophisticated data mining and machine learning algorithms which require multiple datasets to be processed simultaneously. When multiple datasets are involved, a most common operation that is used to combine and collate information from multiple datasets is what is referred to as the *Join* operation used widely in relational database management systems. Although numerous algorithms have been discussed for performing database joins in centralized, distributed, and parallel environments [37, 40, 68, 71, 42, 69, 41, 64, 65], joining datasets that are distributed and are extremely large (hundreds of terabytes to petabytes) poses unprecedented research challenges for scalable join-processing in contemporary data warehouses.

The problem of analyzing petabytes of data was confronted very early by large Internet search companies such as Google and Yahoo!. They analyze large number of crawled web pages from the Internet and create an inverted index on the collection of crawled Web documents. In order to be able to carry out such type of data-intensive analysis in a scalable and fault-tolerant manner in a distributed environment, Google introduced a distributed and parallel programming framework called MapReduce [35]. From a programmer's perspective, the MapReduce framework is highly desirable since it allows the programmer to specify the analytical task and the issue of distributing the analysis on multiple machines is completely automated. Furthermore, MapReduce achieves its performance by exploiting parallelism among the compute nodes and uses simple scan-based query processing strategy. Due to its ease of programming, scalability, and fault tolerance, the MapReduce paradigm has become extremely popular for performing large-scale data-analysis both within Google as well as in other commercial enterprises. In fact, under the umbrella of Apache, an open source implementation of the MapReduce framework, referred to as Hadoop [4] is freely available to both commercial and academic users.

Given its availability as open source and wide acknowledgement of MapReduce parallelism, Hadoop has become a popular choice to process big data produced by the web applications and business industry. Hadoop has been widely deployed in performing analytical tasks on big data, and there is a significant interest in the traditional data warehousing industry to explore the integration of the MapReduce paradigm for large-scale analytical processing of relational data. There are several efforts to provide a declarative interface on top of the MapReduce run-time environment that will allow the analysts to specify their analytical queries in SQL-like queries instead of C-like MapReduce programs. The two major efforts are the Pig project from Yahoo! [12] and the Hive project from Apache [7], both of which provide a declarative query language on top of Hadoop.

The original design of MapReduce was intended to process a single dataset at a time. If the analytical task required processing and combining multiple datasets, this was done as a sequential composition of multiple phases of MapReduce processing. As we consider the adaptation of the MapReduce framework in the context of analytical processing over relational data in the data warehouse, it is essential to support *join* operations over multiple datasets. Processing multiple join operations via a sequential composition of multiple MapReduce processing phases is not desirable since it would involve storing the intermediate results between two consecutive phases into the underlying file-system such as HDFS (Hadoop Distributed File System).

As can be seen in Figure 2.1, each join has to write the intermediate results into HDFS. This approach not only incurs very high I/O cost, it also introduces heavy workload to the NameNode, because NameNode has to maintain various of meta information for the intermediate outputs. When the selectivity is extremely low in a sequence of joins, most of these intermediate files would be empty, but Namenode still has to consume much memory to store the meta information of a large number of such files [70]. Furthermore, if the system has to periodically execute some particular sequences of joins as routine requirements, the situation could further deteriorate.

Recently, several proposals, such as [20] and [48], have been made to process multi-way join in a single MapReduce processing phase. The main idea of this approach is when the filtered (mapped) data tuples are shuffled from the mappers to the reducers, instead of shuffling a tuple in a one-to-one manner, the tuples are shuffled in a one-to-many manner and are then joined during the reduce phase. The problem with this approach, however, is that as the number of datasets involved in the join increases, the tuple replication during the shuffling phase increases exponentially with respect to the number of datasets.

In order to address the problem of multi-way joins effectively in the context of the MapReduce framework, we have developed a system called *Llama*. Llama stores its data in a distributed file system (DFS) and adopts the MapReduce framework to process analytical queries over those data. The main design objectives of Llama are: (i) Avoidance of high loading latency. Most customers who were doing traditional batch data warehousing expect a faster-paced loading schedule rather than daily or weekly data loads [2]. (ii) Reduction of scanning cost by combining the column-wise techniques [34, 56, 67]. In most analytical queries processed over large data warehouses, users typically are interested in aggregation and summarization of attribute values over only a few of the column attributes. In such a case, column-wise storage can significantly reduce the I/O overhead of the data being scanned to process the queries. (iii) Improvement of the query processing performance by taking advantage of the column-wise storage. Llama balances the trade-off between the load latency and the query performance. Llama is part of our *epiC* project [29, 3] which aims to provide an elastic, power-aware, data-intensive cloud computing platform. It is based on an elastic storage system(ES^2) for supporting both OLTP and OLAP [26]. We discuss Llama in the context of MapReduce in this thesis and then discuss how it is implemented on top of ES^2.

For each imported table, Llama transforms it into column groups. Each group contains a set of files representing one or more columns. Data is partitioned vertically based on the granularity of column groups. To this end, Llama intentionally sorts the data based on some orders when importing them into the system. The ordering of data allows Llama to push operations such as *join* and *group by* to the map phase. This strategy improves parallelism and reduces shuffling costs. The contributions of this thesis are as follows:

- We propose *Concurrent Join*, a multi-way join approach in the MapReduce framework. The main objective is to push most operations such as join to the map phase, which can effectively reduce the number of MapReduce jobs with minimal network transfer and disk I/O costs.
- We present a plan generator, which generates efficient execution plans for complex queries exploiting our proposed concurrent join technique.
- We study the problem of data materialization and develop a cost model to analyze the cost of data access.
- We implement our proposed approach on top of Hadoop. It is compatible with existing MapReduce program running in Hadoop. Furthermore, our technique could be adopted in current Hadoop-based analytical tools, such as Pig [12] and Hive [7].

- We conduct an experimental study using the TPC-H benchmark, and compare Llama's performance with that of Hive. The results demonstrate the advantages of exploiting a column-wise data-processing system in the context of the MapReduce framework.
- We present our design of Llama on top of ES^2 with some preliminary results.

The remainder of the thesis is organized as follows. In Section 2 we review the prior work on the integration of database join processing in the MapReduce framework. It contains the joining process and the column-wise storage in MapReduce framework. In addition, we we depict the fundamental single join approaches in MapReduce, including the reduce join, the fragment-replication join and the map-merge join in this chapter. In Chapter 3 we describe the column-wise data representation used in Llama, which is important for processing concurrent joins efficiently. Moreover, we exploit the issue of data materialization and develop a cost model to analyze the materialization cost for the column-wise storage. In Chapter 4 we illustrate the detailed design of concurrent join. A plan generator is designed to generate efficient execution plans for complex queries. In Chapter 5 we present the detailed implementation of Llama. We customize specific input formats and the join processing in Llama to faciliate variant types of processing procedure in one job. In Chapter 6, we first compare the performance of several widely-used file formats with our CFile in Llama. We then evaluate Llama system by comparing it with Hive on the basis of the TPC-H benchmark. We conclude our work in Chapter 7.

Chapter 2

Related Work

In this chapter we shall review prior work on supporting database join processing in the MapReduce framework. We then describe in detail several popular file formats which are widely used in the Hadoop System.

2.1 Data Analysis in MapReduce

The MapReduce paradigm [35] has been introduced as a distributed programming framework for large-scale data-analytics. Due to its ease of programming, scalability, and fault tolerance, the MapReduce paradigm has become popular for large-scale data analysis. An open source implementation of MapReduce, Hadoop [4] is widely available to both commercial and academic users. Building on top of Hadoop, Pig [12] and Hive [7] provide the declarative query language interface and facilitate join operation to handle complex data analysis. Zebra [15] is a storage abstraction of Pig to provide column wise storage format

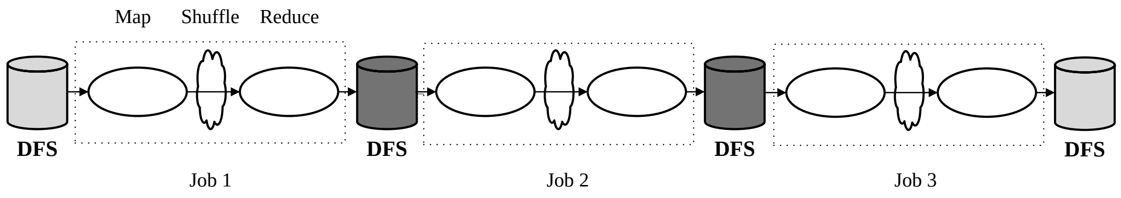


Figure 2.1: Multiple joins in traditional approach in MapReduce

for fast data projection.

To execute equi-joins in the MapReduce framework, both Pig [12] and Hive [7] provide several join strategies in terms of the feature of the joining datasets [9, 8]. For example, [60] proposes a set of optimization strategies for automatic optimization of parallel data flow programs such as Pig. [30] proposes to increase a merge phase after the reduce phase. It improves the performance when there is an aggregation after the join operation, because it saves the I/O without using one more job for the aggregation.

In addition, HadoopDB [19] provides a hybrid solution which uses Hadoop as the task coordinator and communication layer to connect multiple single node databases. The join operation can be pushed into the database if the referenced tables are partitioned on the same attribute. Further detail performance-oriented query execution strategies for data warehouse queries in split execution environments are reported in [22]. Hadoop++ [38] provides a non-invasive index and join techniques for co-partitioning the tables. The cost of data loading of these two systems is quite high. In other words, if there are only a few queries to be performed on the data, the overhead of loading and preprocessing could be too huge. [58] studies the problem of how to map arbitrary join conditions to Map and Reduce functions. It also derives a randomized algorithm for implementing arbitrary joins (theta-joins).

A comprehensive description and comparison of several equi-join implementations for the MapReduce framework appears in [24, 47]. However, in all of the above implementations, one MapReduce job can only process one join operation with a non-trivial startup and checkpointing cost. To address this limitation, [20, 48] propose a one-to-many shuffling strategy to process multi-way join in a single MapReduce job. However, as the number of joining tables increases, the tuple replication during the shuffle phase increases significantly.

In another recent work [50], an intermediate storage system of MapReduce is proposed to augment the fault-tolerance while keeping the replication overheads low. [72] proposes a Hadoop based approached to distributed loading the data to parallel data warehouse. [32] presents a modified version of the Hadoop MapReduce framework that supports online aggregation by pipelining. [59] further describes a work flow manager developed and deployed at Yahoo called Nova, which pushes continually arriving data through graphs of Pig programs executing on Hadoop clusters. [63] presents a system for allocating resources in shared data and compute clusters that improves MapReduce job scheduling in three ways. Facebook is exploring the enhancements to make Hadoop a more effective real time system [25]. [54] examines, from a systems standpoint, what architectural design changes are necessary to bring the benefits of the MapReduce model to incremental one-pass analytics. However, all of the above approaches do not essentially improve the performance of MapReduce based multi-way join processing.

To overcome the limitation of MapReduce framework and improve the performance of multi-join processing, we propose the concurrent join in the following chapter. In this approach, several join can be performed in one MapReduce job, without any redundant data transformation incurred in [20, 48]. A similar approach was also recently proposed in [53].

2.2 Column-wise Storage in MapReduce

The fundamental idea of the column-wise storage is to improve I/O performance in two ways: (i) Reducing data transmission by avoiding to fetch unnecessary columns; and (ii) Improving the compression ratio by compressing the data blocks of individual columnar data [16, 46, 62]. [43, 17] compares the performance of column-wise store to several variants of a commercial row-store system. [45] further discusses the factors that can affect the relative performance of each paradigm.

Although vertical partitioning of the table has been around for a long time [34, 23], it has only recently gained wide-spread interest as a possibly efficient technique for building columnar analytic databases [56, 67, 14, 55] primarily for data warehousing and online analytical processing.

Column-wise data model is also suitable in the MapReduce and distributed data storage systems. HadoopDB [19] can use columnar database like C-store [67] as its underlying storage component. Dremel [57] proposed a specific storage format for nested data along with the execution model for interactive queries. Bigtable [28] proposed column family to group one or more columns as a basic unit of access control. HBase [5], an open source implementation of BigTable, has been developed as the Hadoop [4] database. HFile [6] is its underlying column-wise storage. Besides, TFile [13] and RCFile [44] are the other two popular file structures that have been used in Zebra [15] and Hive [7] projects for large scale data analysis on top of Hadoop. Each of these files represents one column family and contains one or more columns. Their records are presented as a key-value pair. Recently another Pax-like format called Trojan Data Layouts [49] is also proposed for MapReduce. In this approach, one file is replicated into several different data layouts.

HFile provides a similar feature as SSTable in Google BigTable [28]. In HFile, each record contains detailed information to indicate the key by (*row:string, column-qualifier:string, timestamp:long*), because it is specifically designed for storing sparse and real-time data. On the other hand, storing all the detailed information is a non-trivial overhead, because all the records have to contain certain redundant information such as column-qualifier. This makes HFile a wasteful of storage space and thus ineffective in large scale data processing, especially when the table has a fixed schema.

TFile, on the other hand, does not store such meta data in each record. Each record is stored in the following format:(*keyLength, key, valLength, value*). Both key and value could be either null or a collection of columns. The length information is necessary to record the boundary of the key and value in each record. Similar to TFile, RCFile [44] stores the data of the columns in the same block. However, within each block, it groups all the values of a particular column together into a separate mini block, which is similar to PAX [21]. RCFile also uses the key-values pair to represent the data, whereas the key contains the length information for each column in the block, and the value contains all the columns. This file format can leverage a lazy decompression callback implementation to avoid the unnecessary decompression of the uncorrelated columns in particular query.

The above file formats store the columns in a column family on the same block within a file. This strategy provides a good data locality when accessing several columns in the same file. However, it requires reading the entire block even if some columns are not needed in the query, resulting in wasted I/Os. Even though each file stores only one column, the file format is wasteful of storage space, because the length information of both key and value for each record incur non-trivial overhead, especially when that column is small such as being an integer type. Furthermore, these files are only designed to provide the I/O efficiency. There is no effort to leverage the file formats to expedite query processing in MapReduce. In this aspect, Zebra [15] and Hive [7] could not be treated as a truly column-wise data warehouse.

2.3 Single Join in MapReduce

To extract valueble insights and knowledge, complex analysis and sophisticated data mining are usually processing on multiple datasets. A most common operation that is used to collate information from different datasets is the *EquiJoin* operation. An equijoin between table A and B on one or more columns can be presented as $A \bowtie_{A.x=B.x} B$. Given the massive scale of data, these two tables as well as the joined results are usually stored in DFS. There may be predicates or projections on either table. In the MapReduce system, any nodes in the cluster are able to access both A and B. In other words, any mappers and reducers can access any partitions of these tables according to the proper reader, which is provided by the corresponding file format in the MapReduce framework.

Single join has been well studied in the MapReduce environment [9, 8, 24]. Both Pig [12] and Hive [7] provide several join strategies and are widely available to both commercial and academic users. In general, there are three common approaches for processing a single join in the MapReduce framework:

Table 2.1: Single Join in MapReduce framework

Join Strategy	Phase	Requirement
Reduce Join	Reducer	Shuffle from mapper to Reducer by the join key
Fragment-Replication Join	Mapper	One table is small enough
Map-Merge Join	Mapper	Both table are sorted by the join key

1. Reduce Join. It is the most common approach to processing the join operation in the MapReduce framework. The two tables that are involved in the join operation are scanned during the map phase. After the filtering and the projection operations performed in the mapper, intermediate tuples are shuffled to the reducers based on the join key and are joined at the corresponding reduce nodes during the reduce phase. This method is similar to the hash-join approach in the traditional databases. It requires shuffling data between the mappers and the reducers. This approach is referred to as *Standard Repartition Join* in [24].

2. Fragment-Replication Join. This approach is applied when one of the tables involved in the join is small enough to be stored in the local memory of the mapper nodes. In the map phase, all the mappers read the small table from DFS and store it in the local memory. Then each mapper reads a fragment of the large table and performs the join in the mapper. Note that neither of the tables is required to be sorted in advance. This approach is referred to as *Broadcast Join* in [24].

3. Map-Merge Join. This approach is used when both tables are sorted already based

on the join key. Each mapper joins the tables by sequentially scanning the respective partitions of these two tables. This approach avoids the shuffle and the reduce phase in the job and is referred to as *Pre-processing for Repartition Join* in [24]. Fragment-replication join and map-merge join are referred to as map-side join, as they are performed in the map phase.

These single joins are important to Llama, because they constitute the basis of our concurrent join. In the following sections, we will present how we leverage the column wise storage to make the join processing more efficient in the MapReduce framework.

Chapter 3

Column-wise Storage in Llama

In this section, we describe a new column-wise file format for Hadoop called *CFile*, which provides better performance than other file formats in data analysis. We then present how to manage the columns into vertical groups in different orders. Such groups are important in Llama, because it facilitates certain operations such as the map-merge join. In addition, under the column-wise storage, the cost of scanning and creating different groups is acceptable.

3.1 CFile: Storage Unit in Llama

To specify the structure of a CFile, we use a notion of *block* to represent the logical storage unit in each file. Note that the notion of block in CFile format is logical in that it is not related to the notion of disk blocks used as a physical unit of storage of files. In CFile, each block contains a fixed number of records, says k. The size of each logical block may

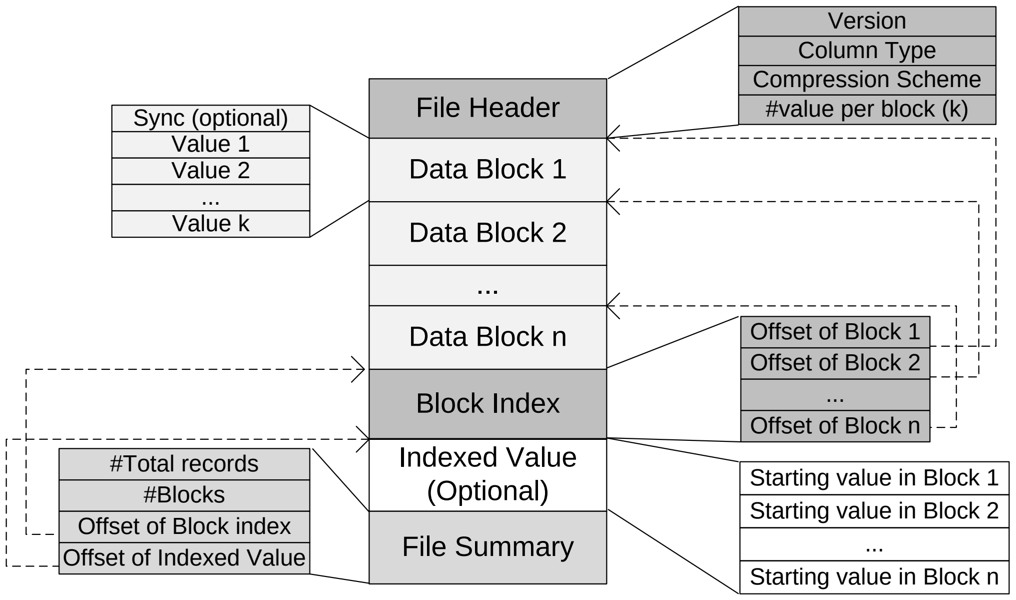


Figure 3.1: Structure of CFile

vary since records can be variable-sized. Each block is stored in the *buffer* before flushing. The size of buffer is usually 1 MB. When the size the buffer is beyond a threshold or the number of records in the buffer become *k*, the buffer is flushed to DFS. The starting offset of each block is recorded. In addition, we use *chunk* to represent the partitioned unit in the file system. One file in HDFS is chopped into chunks and each chunk is replicated in different data nodes. A default chunk size in HDFS is 64 MB. One chunk contains multiple blocks depending on the number of records *k* and the size of each record.

CFile is the storage unit in Llama to store the data of a particular column. In contrast to the other file formats that store the records as a collection of key-value pairs, each record in CFile contains only a single value. As illustrated in Figure 3.1, one CFile contains multiple data blocks and each block contains a fixed number (*k*) of values. *Sync* may be contained in the beginning of the block for checkpoint in case of failure. The block index, which is stored after all the blocks are written, stores the offsets of each block and is used to locate a specific block. For example, if we need to retrieve the n-th value of an attribute in a CFile of the attribute, we obtain the offset of the n/k-th block, read the corresponding block, and retrieve the $n\%k$-th value in that block. If the column is already sorted, its CFile could also store all the starting values of each block as the indexed value. A look up via a certain value is thus supported by performing a binary-search in the indexed value. As the block is located, Llama scans the block and calculates the position of that value. This position is further used to retrieve other columns of the same tuple but in the different CFile. Both the block index and the indexed value are loaded only when random access is required.

In order to achieve storage efficiency, we can use block-level compression by using any of the well-known compression schemes. Any of the available compression schemes in Hadoop could be easily deployed for compressing CFiles. Some column-specific compression algorithms such as *run-length encoding* and *bit-vector encoding* will be implemented in our next step. As the block index and indexed value are stored immediately after all blocks are flushed to DFS, they are stored in the memory before they are written to DFS. If these intermediate values are too large to be stored in the memory, they are flushed to a temporary file and are finally copied into the appropriate location. In most cases the block index will not exhaust the memory because the index contains only the offset information. In the tail of the CFile, summary information is provided. For instance, the offsets of the block index and the indexed value are included in the summary.

The number of tuples in each block needs to be carefully tuned. A smaller block size is good for random access, but it uses a larger amount of memory to maintain the block index for efficient search. It also incurs high overhead when flushing too many blocks to DFS. A larger block size, on the other hand, is superior when data are accessed primarily via sequential scan, but this incurs inefficient random accesses since more data are decompressed for each access. Based on our experiment results, we have found that

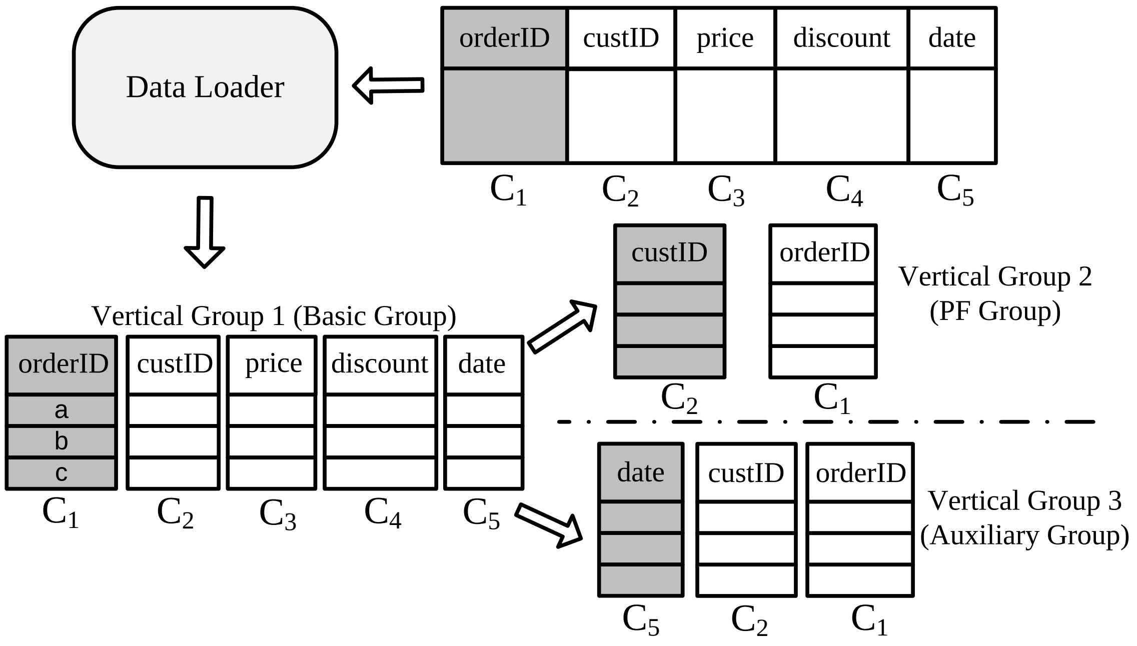
- For frequent random access, even applying index cannot provide a satisfactory performance.
- In the map-merge join, the mapper only incurs very few random I/O for a table to locate the starting block for join, which is not a performance bottleneck.

As an alternative, [39] proposes a skip list format to store the data within each column file. That is, each column file contains the skip blocks, which indicate the information about byte offsets to enable skipping the next N records. Here N is typically configured for 10, 100, and 1000 records. Using the skip/seek function, this format can reduce the serialization overhead without materializing all the records. However, this alternative suffers a potential disadvantage. To locate a specific position in one column file, it needs to perform several skip operations, which incurs a non-trivial overhead as random access. In our CFile, each block has a fixed number of records and the offset information is stored as the index in the beginning of CFile. As a result, it only needs one seek operation and thus incurs less overhead of random access and late materialization.

Compared to TFile of Pig [12] or RCFile [44] of Hive [7], the primary benefit of CFile is that it significantly reduces the I/O bandwidth during the data analysis since only the necessary columns are accessed rather than a complete data tuples. In addition, increasing a column in certain column family to the existing table is much cheaper than RCFile or TFile. In CFile, this can be done by simply increasing an additional CFile in the proper directory and update the meta information. Instead, TFile and RCFile have to read and rewrite the whole column family. This flexibility make CFile competent in the scenario in which the schema is frequently updated. In addition, since each column is stored in a separated CFile in the HDFS, it is possible to read each CFile by an individual thread and then store the data in the buffer when processing. This approach can further improve the I/O performance significantly and maximize the utility of the network bandwidth. The experimental results will be presented in Chapter 6.

On the other hand, HDFS individually distributes each file to different data nodes without considering their data model. This may potentially give rise to the problem of the loss of data locality in HDFS. Data from the same column family may need to be read from different data nodes. It is possible to modify the HDFS so that it assigns the same node for a set of CFiles corresponding to the attributes in the same column family. Meanwhile, to maintain the load balance, we can add the meta data in the namenode to gather the statistical information of CFile distribution. Although this approach solves the data locality issue, it violates the design philosophy that the data model should be independent of the underlying file system. To comply with this rule, our experiments in data analysis use the original HDFS.

Recently, [39] proposes a similar approach to co-locate the related columns in the same data nodes by replacing the default HDFS block placement policy. They show that the performance can be significantly improved without remotely accessing any column files.



Data Loader
orderID
custID
price
discount
date
C1
C2
C3
C4
C5
Vertical Group 1 (Basic Group)
a
b
c
Vertical Group 2 (PF Group)
Vertical Group 3 (Auxiliary Group)

Figure 3.2: Data Transformation

3.2 Partitioning Data into Vertical Groups

One challenge of table join is to efficiently locate the corresponding data according to the joining key. If the columns are sorted in different ways, we may need to scan the entire columns to produce the final results. To address this problem in Llama (specifically to facilitate the processing of star-join and chain-join), we create multiple vertical groups similar to the *projection* in C-store [67].

Definition 1 ***Vertical Group.*** *A vertical group V_g of table T is represented as (G, c), where G is a collection of columns and c is the sorted column in the group. V_g is created by projecting T on G and sorting via c.*

A vertical group is physically maintained as a set of CFiles, which are built for columns in G. Initially when a table T is imported into the system, Llama creates a *basic* vertical group which contains all the columns of the table. The basic group is sorted by its primary key by default. A *basic* group can be represented as v_g=($\{c_i \mid 0 \leq i \leq n\}$, primary_key), where n is the number of columns in table T. To build the basic group of T, we actually create n CFiles. Each CFile refers to a column in T and all CFiles sort the data by the primary key. For example, Group 1 in Figure 3.2 is the *basic* group of table *Orders*.

In addition, Llama creates another vertical group called *PF* group to facilitate the map-merge join in our concurrent join approach. A *PF* group is represented as v_p=($\{$primary_key, foreign_key, $c_i\}$, foreign_key). Tuples in a *PF* group are sorted by the foreign key. Besides the primary key and foreign key columns, *PF* group can contain other columns. New columns are inserted into *PF* group during the data processing for better performance. If there are k foreign keys in one table, Llama could build k *PF* groups based on different foreign keys. For instance, Group 2 in Figure 3.2 shows a simple *PF* group of table *Orders*. There are two columns *custID* and *orderID* in the group and data are sorted by *custID*. In Llama, a column can be included into multiple groups in different sorted orders. For example, column *custID* in Figure 3.2 appears in both basic group and *PF* group with different sort orders. In addition, it is not necessary to build the *PF* group when the table is being imported, since the *PF* group is built when the table needs to perform the map-merge join on that foreign key or for some ad-hoc queries.

Based on the statistics of query patterns, some auxiliary groups are dynamically created or discarded to improve query performance. For example, if Llama monitors that many queries compute the order statistics in some date ranges, it creates a vertical group v_p=($\{date$, *custID*, *orderID*$\}$, $date$) at runtime which is sorted by $date$. sorted by $date$ at runtime. In this way, Llama can answer the queries without scanning the entire datasets of the corresponding basic groups.

We use a variant of Log-Structured Merge-Tree (LSM-Tree) [61] to insert and update the tuples in our system. In this idea, its basic data structure consists of a base tree, and a number of incremental trees to the base. The newly updated records are stored in the incremental trees. While the number of the incremental trees reaches a certain threshold, the system merge them together and delete the out-of-date records. When the size of the base tree is large enough, it would be split into two individual trees. In our system, to update a particular vertical group when there are a batch of new records, Llama first extracts the corresponding columns of those new records. It then sorts the records via the specific column and writes the sorted results to a temporary group. This temporary delta group is periodically compacted with the original group. If a physical group is larger than a threshold, it is split into different physical files for better load balance. This approach is similar to the compact operation in BigTable [28].

3.3 Data Materialization in Llama

In the column-wise data model, there are two possible materialization strategies during the query processing [18]: Early Materialization (EM) and Late Materialization (LM). EM materializes the involved columns as the job begins, whereas LM delays the materialization until it is necessary. Both materialization strategies can be used in Llama and implemented in the MapReduce framework, but they incur different processing costs depending on the underlying queries. Because most jobs in Llama are I/O intensive, the I/O cost is our primary concern in selecting a proper materialization strategy. Before we analyze these materialization strategies, we first analyze the processing flow in Llama to have a clear understanding of the overhead.

Similarly to the MapReduce framework, records from DFS are read by a specific reader and then pipelined to the mapper. After being processed by the mapper, the records are optionally combined and then partitioned to the reducer. To simplify the relational data processing, Llama introduces an optional joiner before the reducer in the reduce phase, which is similar to the Map-Join-Reduce approach described in [48]. In addition, we push the predicate operations to the reader to prune the unsatisfied records.

Since records from the reader are passed to the mapper and combiner by reference with zero-copy, the major I/O cost in map phase is to read the data from DFS. Another obvious I/O cost is the data shuffling from the mapper to the reducer. Here we include the overhead of spilling after map() and the merging before reduce() in the shuffling cost, as they run in the sequential overlapping phases in the job and are proportional to the size of the shuffled data.

EM materializes all the tuples as the job initializes. LM delays the materialization until the column is necessary. That is, LM can be processed in either map or reduce phase. As a result, it reduces the scan overhead of early materialization. If LM is processed in reducer, it further reduces the shuffling overhead of certain columns. On the other hand, these columns have to be materialized by random access in the DFS whenever they are needed. As random access contains the seeking cost and the I/O cost through network, it is a non-trivial overhead for LM. We use the following query as an example to show the difference of these two materialization strategies in terms of their executions and overheads in the MapReduce framework:

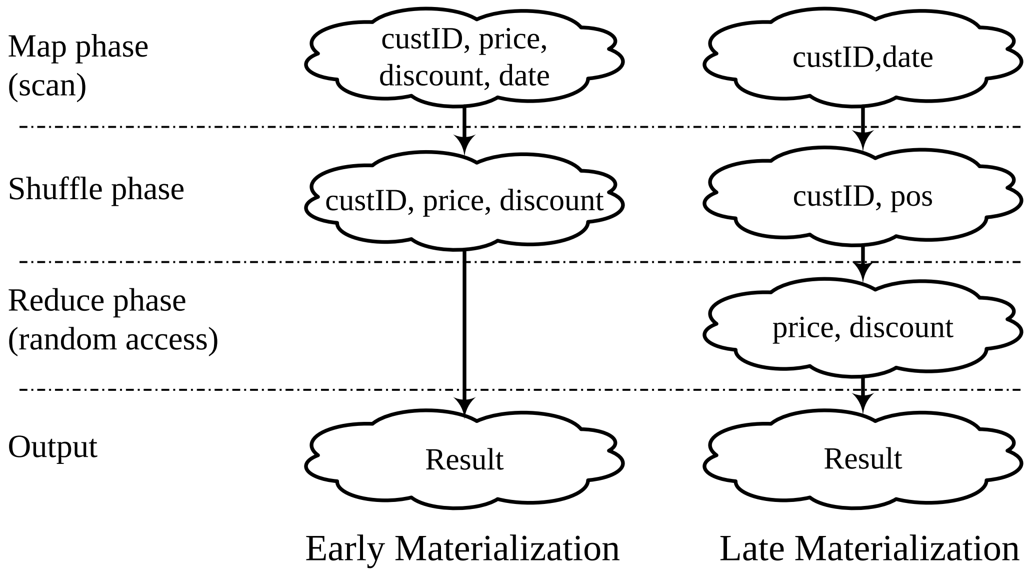


Figure 3.3: Example of Materializations

```
SELECT custID, SUM(price * (1 - discount)) as Revenue
FROM Orders
WHERE date < X
GROUP BY custID;
```

As illustrated in Figure 3.3, EM materializes four involved columns as the job begins. *date* is used to filter unqualified records by a specific reader. After pruned, the remaining three columns are shuffled to the reducers via *custID*. In contrast, LM only materializes *custID* and *date*, as *date* is used for predicate and *custID* is used for partitioning. Other columns are delayed to be materialized in the reduce phase in this example. The position information *pos* is maintained by the reader during materialization without consuming the I/O bandwidth in the map phase. *pos* is further shuffled with *custID* to the reducer and is used to late materialize the corresponding *price* and *discount* by random access. Comparing these two strategies, the major overhead of EM is the input and shuffle overhead, while LM reduces the input and shuffle overhead at the cost of random access in the DFS.

Given a table $T = (c_0, c_1, ..., c_k)$, S ($S \subseteq \{c_0, ..., c_k\}$) is the column set and r_{op} is the cost ratio for specific operation op in the MapReduce job. op could be sequential scan, shuffle or random access. For example, r_{scan} denotes the cost ratio of sequential scan in the DFS. In addition, $|T|$ is the number of tuples in table T. f denotes the selectivity of filters for the predicate. $Size(c_i)$ is the average size of column c_i. To early materialize certain column set S in initialization, the I/O cost is

$$C_{scan}(S) = |T| \times r_{scan} \times \sum_{\forall c_i \in S} Size(c_i) \tag{3.1}$$

To shuffle the pruned records to the reducers, the I/O cost is

$$C_{shuffle}(S) = f \times |T| \times r_{shuffle} \times \sum_{\forall c_i \in S} Size(c_i) \tag{3.2}$$

To random access certain columns via their positions in late materialization, the I/O cost is

$$C_{random}(S) = f \times |T| \times r_{random} \times \sum_{\forall c_i \in S} Size(c_i) \tag{3.3}$$

While selecting a materialization strategy, we compare the random access overhead to the saving of I/O from input and shuffle. We use ΔS to indicate the column set that uses late materialization. If LM is processed in the map phase, the different cost of LM over EM is mainly related to random access and sequential scan:

$$\Delta C = C_{random}(\Delta S) - C_{scan}(\Delta S) \tag{3.4}$$

If LM is processed in the reduce phase, the different shuffling cost should be included:

$$\Delta C = C_{random}(\Delta S) + C_{shuffle}(\{pos\}) - C_{scan}(\Delta S) - C_{shuffle}(\Delta S) \tag{3.5}$$

To compute ΔC with different shuffling cost, we need to take the additional information *pos* into consideration, because *pos* is necessary to late materialize those missing columns. If ΔC is larger than zero, EM is chosen for less overhead. Otherwise LM is chosen. Since *pos* is of long type, $Size(\{pos\})$ is 8 bytes in Java. We can further simplify the above equation and draw the following conclusion:

$$materialization = \begin{cases} EM, & \text{if } f > \frac{r_{scan} \times L}{r_{random} \times L + m \times r_{shuffle} \times (8-L)} \\ LM, & \text{else} \end{cases} \tag{3.6}$$

Here L is the average length of the late materialized records and is equal to $\sum_{\forall c_i \in \Delta S} Size(c_i)$. In addition, m denotes in which phase LM is processed. If LM is processed in mapper, m is equal to 0. Otherwise m is equal to 1, meaning that the shuffling differences should be considered. As will be seen in Section 6.5, r_{scan}/r_{random} is usually much smaller than $r_{scan}/r_{shuffle}$. In this case, we can simply compare f and r_{scan}/r_{random} for fast estimation. More details about the experimental study is provided in Section 6.5. This cost model is easy to be extended for the join situation. If there are n tables involved in one MapReduce job for the join operation, we can separately consider the overhead of these tables and pick the proper materialization strategies for them.

3.4 Integration to epiC

The goal of the *epiC* project [29, 3] is to develop an **e**lastic, **p**ower-aware data-**i**tensive **C**loud computing platform for providing scalable database services on the cloud. It is composed of the Elastic Storage System (ES^2) and the Elastic Execution Engine (E^3). ES^2 is the underlying storage system, which is designed to provide a good performance for both OLAP and OLTP application. E^3, a sub-system of epiC, is designed to efficiently perform large scale analytical jobs on the cloud. In contrast to the concurrent join oriented to overcome the limitation of multi-way join in the specific MapReduce framework, E^3 proposes a more flexible and extensive approach to solving the multi-way join problem. In order not to restrict the advantages of E^3, our integration mainly focuses on the storage layer. In this section, we first present the architectural design of the epiC storage system. We then describe our approach to adapting the columnar storage to epiC as an option to improve its performance with different kinds of workloads.

3.4.1 Storage Layer in epiC

System Overview ES^2 works towards the ultimate objective of managing the data that are accessed simultaneously by both OLTP and OLAP queries. It contains three major components:

- **Data Importer** supports efficient bulk data loading from the external data sources to ES^2. Such external data sources include the traditional database, or plain file data from other applications.

- **Data Access Control** responses the requests from both OLAP and OLTP applications. It parses the requests into internal parameters, estimates the overhead of the different accessing approaches, determine the optimal strategy and finally retrieve the corresponding results from the underlying physical storage.

- **Physical Storage** stores all the data in the system. It is composed of distributed file system (DFS), meta-data catalog, and distributed index system. The DFS is a scalable, fault tolerant and load balanced file system. The meta-data catalog manages the meta information of the table stored in the system. The distributed indexes interacts with the DFS to facilitate a efficient search for a few amount of tuples.

Data Model ES^2 exploits the widely-studied relational data model [31] that all data is represented as mathematical n-ary relations. This model permits the database designer to create a consistent, logical representation of information. Different to the Key-Value data model adopted in the NoSQL data store [27] which oriented to the sparse and flexible data application, using the relational data model facilitates users migrate their database and the corresponding application from the traditional database to our system in a low transferring overhead. Moreover, the user is convenient to switch the database without an expensive learning cost. This meets our objective to provide database-as-a-service and efficiently support of database migration and adaptation.

Data Partitioning ES^2 employs both vertical and horizontal data partitioning schemes. First, the system optionally divide columns of a table schema into several column groups based on the query workload. Each column group contains columns that are often accessed together and will be stored separately in a physical table. Each physical table contains the primary key of the original table, which is used to combine the columns from different groups. This approach is similar to the data partition strategy proposed in Llama, and benefits the OLAP queries with accessing only a subset of columns via particular predicates. Additionally, partitioning the data into vertical groups facilitates to organize the data in different orders. This will significantly improve the speed of certain queries related to the sorted columns. Note that the data is vertically partitioned into tables with different orders based on the queries workload, it may need to re-construct the tuples if the involved columns are distributed in different tables in the worst case.

After vertically partitioning, ES^2 then horizontally distributes the vertical group of a particular table into different data nodes. If one transaction involves several tuples that are distributed in different data nodes, the overhead of communication and the transactional management is extremely high. As a result, the system need to carefully design and tune the horizontal partition strategy to balance the workload and the overhead of transactional managements.

PaxFile Layout ES^2 treats the DFS as a raw byte device, which means that tables are stored in the DFS as binary byte streams. Therefore, ES^2 uses a page-based storage layout, to explicitly interpret the output byte streams from the DFS into records.

ES^2 employs the PAX (Partition Attribute Across) [21], which is essentially a DSM-like [34] organization within an NSM (N-ary Storage Model) page. In general, for a table with n columns, its records are stored into pages. PAX vertically partitions the records within each page into n mini-pages, each of which consecutively stores the values of a single column. While the disk access pattern of PAX is the same as that of NSM and does not have an impact on the memory-disk bandwidth, it does improve on the cache-memory bandwidth and therefore the CPU efficiency. This is because of the fact that the column-based layout of PAX physically isolates values of different columns within the same page, and thus enables an operator to read only necessary columns, e.g. the aggregated columns that are referred in an aggregation OLAP query. The use of PaxFile layout also has great potential for data compression. For each column, we add a compression flag in the header of a Pax page to indicate which compression scheme is utilized.

3.4.2 Columnar Storage Integration

Before we present our approach of adaptation, we first emphasize the assumptions and objectives of epiC when we are integrating the columnar storage to achieve the good performance in both OLTP and OLAP application:

- All the data follows the relational data model that has a fixed schema in certain application.
- The storage system is append-only, to achieve high throughput low latency in the update-heavy applications.
- The insert operation is operated on a tuples with all the attributes, whereas most updates are only operated on one or very few columns.

As a result, we need to carefully consider how to adapt the columnar storage, so that it will not incur any adverse effects in either OLTP or OLAP. In this part, we will present and discuss our approach of the integration. The experimental results will be reported in Chapter 6.

To provide a high throughput low latency append-only storage system for the update operations in OLTP, we suggest to create two kinds of file formats to store the data. Originally, ES^2 uses the *PaxFile* to store a whole tuple in one file in PaxFile format, whereas it uses the *IncrementalFile* to store the incrementally updated information on certain columns in a sparse format. The IncrementalFile is similar to the SSTable in BigTable or the HFile in HBase, which is mainly designed to store the sparse update data.

To integrate the columnar storage to epiC and maintain the high performance of transactional write operation, we would keep the IncrementalFile remained. On the other hand, we manage to replace the PaxFile by CFiles to meet the requirement of daily analytical tasks. We keep the IncrementalFile unchanged because it is designed for the frequent updates on arbitrary sparse columns. All of its entries are formatted as *key-value* pairs. The *key* is the combination of primary key and the column name, and the *value* indicates the exact value of the corresponding column. However, CFiles follows a fixed schema, and each entry represents a materialized tuple containing several columns. Such data structure makes it inappropriate to gracefully complete the arbitrary updates. Supposed we obstinately continue to use the CFiles in a vertical group to represent an arbitrary column, we have to set the other columns NULL, or read the corresponding value from the previous records and initial the other columns in the corresponding CFiles. The first approach results in too much NULL indication in the columns and thus makes the file not compact in contrast to IncrementalFile. The second approach incurs expensive overhead of random access for multiple columns for each write operation. As a result, it is inefficient to replace the IncrementalFile by either PaxFile or CFile.

On the other hand, we consider to take CFiles as an alternative to replace PaxFile in ES^2, because both PaxFile and CFiles employ the relational data model and store the columns in one tuples. Furthermore, CFiles is especially designed for the large scale data analysis, which significantly reduces the I/O in columnar storage. It also provides efficient interface to random access according to the primarily key or the position. All these features convince us that CFile is a competitive alternative to PaxFile to provide the similar functions in our ES^2.

There are two approaches for us to add the support of columnar storage in ES^2. The first one is the offline transformation that transforms the transactional data into the CFiles format. This offline approach can be implemented in one MapReduce job or a distributed job launched by the other processing engines. Such approach is effective, but it has to periodically launched the transformation jobs, which is similar to the conventional strategy to move the data from the database to the enterprise data warehouse. As such, it is suitable for the periodic analysis in batch, or the data minding on the historical data, but it is difficult to execute a real time analysis on the columnar storage since the columnar data can be only generated after certain period.

The other approach is the online transformation that transforms the data into CFiles during the execution of transactional processing, which is similar to compact the data into the PaxFile in ES^2. When an update request arrives, the updated column is first inserted in the buffer. If the buffer is full, the system flushes the buffer to the IncrementalFile in the order of primary key. As the number of IncrementalFile increases, the system will involve the compact operation to filter the out-of-date data, and combine the latest columns into the structured CFiles. After the compact finishes, the IncrementalFile is automatically removed from the file system. In addition, if the amount of data in one region is larger than a certain threshold, the data would be split and stored into separated regions with different CFiles.

If the applications require to perform most real-time analysis on the data, we will use the online approach to transforming the data into the compact columnar format for a faster real-time analysis in OLAP. Otherwise, if most of the analytical queries are periodically executed on the historical in batch, we will prefer the offline transformation, so that more resources of the server are allocated to the transactional processing for a lower latency in

OLTP.

When an access request arrives after we integrate the CFiles in the storage, the system will search and read the corresponding data, which is likely in both the CFiles and the IncrementalFile. It then combines the records and finally returns the latest value. For the analytical tasks that required to scan a large amount of data produced by the OLTP, the system first measures the requirements on performance and the data freshness. If the analytical tasks require to process the latest data, the system will scan both the CFiles and the IncrementalFile and combine the results. Otherwise, the system just scans the structured CFile without any scan and combination of the IncrementalFile for a better performance.

Chapter 4

Concurrent Join

Given a query which involves joins over multiple datasets, we split the single multi-way join query into multiple sub-queries. Each sub-query involves a single join over two datasets. These sub-queries can be executed concurrently by instantiating concurrent but distinct map phases for all the sub-queries. Since each of the concurrent map phases is involved in a join operation, we refer to this approach as *concurrent join*. In previous proposed approaches [9, 8], a MapReduce job only processes a single join and the join result is flushed to DFS. The next job loads the result and continues the join processing with other tables. This strategy incurs a significant network overhead and high I/O costs. In addition,

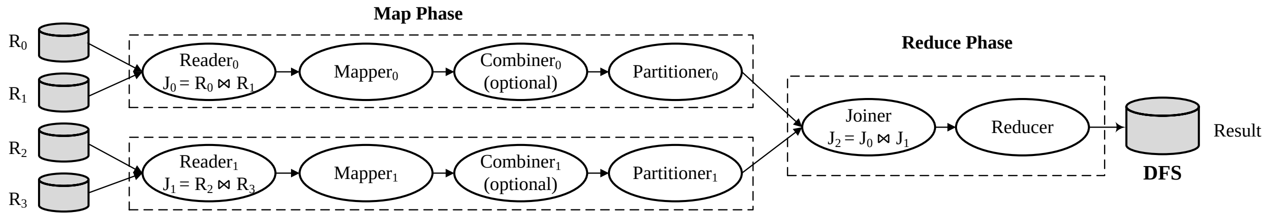


Figure 4.1: Concurrent Join Overview in MapReduce Framework

it puts more memory pressure to the NameNode because the NameNode has to maintain the meta information of these intermediate results. Concurrent join effectively eliminates the need for the sequential composition of multiple instances of MapReduce jobs, hence outperforming existing solutions by a few orders of magnitude.

The intuition of concurrent join is to push as many join operations as possible to the map phase in the MapReduce framework. Its basic idea is to take advantage of the bushy tree plan and the sorted PF groups to facilitate multiple joins in one MapReduce job, by avoiding the expensive data copy in shuffling and reduce the number of MapReduce jobs. In addition, this approach reduces the pressure of the NameNode to maintain a large number of intermediate results in HDFS.

Figure 4.1 illustrates the execution of a concurrent join in the MapReduce framework for an example query $R_0 \bowtie R_1 \bowtie R_2 \bowtie R_3$. As shown in the figure, two distinct types of map phases are generated for sub-query $R_0 \bowtie R_1$ and for sub-query $R_2 \bowtie R_3$, respectively. Specific readers respond to facilitate the map-side join and the joined results are pipelined to mappers. The intermediate results in the map phase are processed further by transferring the appropriate data partitions to the corresponding joiners and reducers by specific partitioners. Each joiner in turn joins the intermediate results from the two types of map phases. These results are further transferred to the reducers for aggregations if necessary. The final results are written to DFS. In Llama, the join operations are respectively completed by Reader and Joiner in the map phase and reduce phase. The implementation details are provided in Section 5.

As discussed in Section 2.3, map-merge join requires that both tables are sorted based on the join key. This is one of the main reasons for us to choose column-wise storage in our system. If we need to push the join operation of two large tables to the map phase but the tables are not sorted as needed to perform the join, re-sorting of certain columns becomes necessary. By leveraging the column-wise storage, we can scan and re-sort specific columns instead of the entire table, which can improve the sorting performance substantially. In this section, we present how concurrent join is implemented in the context of queries that have different join patterns. We consider two most popular query patterns in data warehouse systems, the star join queries and the chain join queries. We then describe how to handle the complex query in hybrid pattern. The query plan of TPC-H Q9 is used as a running example to illustrate the processing logic in the MapReduce framework proposed for Llama.

To search possible query plans, we construct a directed graph, in which each table is denoted as a node, and each join operator is denoted as an edge. The direction of the edge is determined by the foreign-key relationship. If table R_0's foreign key is joining R_1's primary key, we add an edge from R_1 to R_0 in the graph. In this way, we can transform most queries into a directed graph. For example, Figure 4.2(A) shows the graph for a star pattern. Here R_0 is the fact table and R_1, ... ,R_4 are the dimension tables. Moreover, the four dashed eclipses mean four available concurrent map-side joins, namely $R_0 \bowtie R_1$, ..., $R_0 \bowtie R_4$.

4.1 Star Pattern in Data Cube based Analysis

The star join queries have the form $R_0 \bowtie_{R_0.a_1=R_1.a_0} R_1 \bowtie_{R_0.a_2=R_2.a_0} \cdots \bowtie_{R_0.a_n=R_n.a_0} R_n$, where R_0 corresponds to the fact table and R_is $(1 \leq i \leq n)$ correspond to the dimension tables. For notational convenience, we use $R_i.a_0$ as the primary key of the dimension table

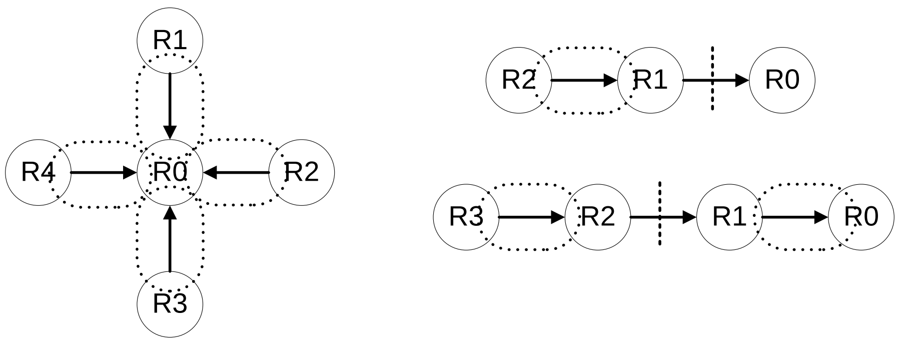


Figure 4.2: Directed Graph of Query Plan

R_i, and use $R_0.a_i$ as the corresponding foreign key of fact table R_0. To handle such queries, Llama generates only one MapReduce job. In its map phase, there are n types of mappers created for processing $R_0 \bowtie R_1$, $R_0 \bowtie R_2$, ..., $R_0 \bowtie R_n$, respectively. Each type of mappers is used to perform a specific map-merge join as described in Section 2.3. All the intermediate results are transformed into key-value pairs and shuffled to the reducer via R_0's primary key. If the involved attributes of R_0 in the query are not covered in the PF groups, Llama uses an additional type of mapper to shuffle R_0's missing attributes to the reducers, or delays the materialization to the reduce phase (the details of data materialization are presented in Section 3.3). In the reduce phase, after pulling and grouping the key-value pairs by the same key, reducers combine the pairs from different types of mappers and finally obtain the results of joining $(R_0 \bowtie R_1) \bowtie (R_0 \bowtie R_2)... \bowtie (R_0 \bowtie R_n)$.

4.2 Chain Pattern

Tables in the query can be linked in a chain manner as $R_0 \bowtie_{R_0.a_1=R_1.a_0} R_1 \bowtie_{R_1.a_1=R_2.a_0} ...R_n$. For notational convenience, we use a_0 as the primary key attribute of table R_i and a_1 as the foreign key attribute of table R_i. In Llama, one chain query is split into a set of sub-chains of a three-way or four-way join. To complete this sub-chain in a single MapReduce job by concurrent join, the chain is split into two parts. Each part refers to a type of mapper for the map-side join or materialization. Their intermediate results are joined by reducers to produce the final results. Figure 4.2(B) shows the examples of three-way join and four-way join queries in the chain pattern. The dashed vertical line indicates how we split the chain.

To process a three-way join $R_0 \bowtie_{R_0.a_1=R_1.a_0} R_1 \bowtie_{R_1.a_1=R_2.a_0} R_2$, at least two types of mappers are needed: one to retrieve R_0's attributes from R_0's basic group, and the other to perform a map-side join for $R_1 \bowtie_{R_1.a_1=R_2.a_0} R_2$. All the intermediate results from the mappers are partitioned by $R_0.a_1$ and $R_1.a_0$.

To perform a four-way chain query $R_0 \bowtie_{R_0.a_1=R_1.a_0} R_1 \bowtie_{R_1.a_1=R_2.a_0} R_2 \bowtie_{R_2.a_1=R_3.a_0} R_3$, two types of mappers are created for $R_0 \bowtie R_1$ and $R_2 \bowtie R_3$. They join R_0's *PF* group with R_1's basic group, and join R_2's *PF* group with R_3's basic group, respectively. All the intermediate results are shuffled via $R_1.a_1$ and $R_2.a_0$ and are finally joined in the reduce phase. In this approach, there may be missing attributes of R_0 and R_2 because we process them via their *PF* groups for the map-merge join. To get the missing attributes of R_2, we could use one more kind of mapper to retrieve them from R_2's basic group, or late materialize them in the reduce phase. For the missing attributes of R_0, only random access is possible. The reason is that, intermediate results from the mappers to the reducers are partitioned by $R_1.a_1$ or $R_2.a_0$. Since R_0 does not contain these attributes, partitioner is unable to partition R_0's tuples individually to the proper reducer.

The primary difference of join strategies between these two patterns is that: In the star pattern, all joins between the fact table and dimension table are performed in the map phase. Their intermediate results are shuffled via the primary key of the fact table and reducers essentially perform the merge-like operation. In the chain pattern, intermediate results from the mappers are shuffled via the join key instead of the primary key of the fact table.

4.3 Hybrid Pattern

Given a complex query, Llama translates it into a set of sub-queries. Each sub-query is composed of a set of joins that can be processed by a single MapReduce job. Basically, sub-queries of star pattern and chain pattern are considered. Algorithm 1 illustrates the plan generation for complex queries in Llama.

As presented in Algorithm 1, if there is only one table in the graph, it returns a materialization operation (Line 4). Otherwise, it iterates all the nodes and generates different execution plans according to the following cases:

- If Table n_i is small enough to be buffered in the memory, we employ the fragment-replication algorithm (Line 9) to join n_i with the remaining graph G'. That is, each mapper reads a replica of n_i and stores it in the local memory. The join is thus capable to be performed in the map phase. The plan of G' is generated by the same algorithm as well.

Algorithm 1 `Plan generatePlan(QueryGraph` G`)`

```
1:  NodeSet $S$ = getAllNode($G$);
2:  PlanCost $cost$ = MaximumCost; Plan $plan$=null;
3:  if $S$.size() == 1 then
4:      return Materialization($S$);
5:  for $\forall$ node $n_i \in S$ do
6:      Plan $tmpPlan$ = null;
7:      if $n_i$ can be buffered in memory then
8:          QueryGraph $G' = G - n_i$
9:          $tmpPlan$ = FragmentReplicationJoin($n_i$, generatePlan($G'$));
10:     else if $n_i$ is a fact table in $G$ then
11:         // process in star pattern
12:         List $list$ = new List();
13:         for $\forall$ (connected subGraph $G_j$) do
14:             $list$.add(Join(generatePlan($G_j$), $n_i$));
15:         $tmpPlan$ = Join($list$);
16:     else
17:         // process in chain pattern
18:         Split $G$ into $G_1$ and $G_2$
19:         $tmpPlan$ = Join(generatePlan($G_1$), generatePlan($G_2$));
20:     if $cost$ > estimateCost($tmpPlan$) then
21:         $plan$ = $tmpPlan$; $cost$ = estimateCost($plan$);
22:     $tmpPlan$ = flattenPlan($tmpPlan$);
23:     if $cost$ > estimateCost($tmpPlan$) then
24:         $plan$ = $tmpPlan$; $cost$ = estimateCost($plan$);
25: return $plan$;
```

- If Table n_i only acts as a fact table in the graph, that is, all its edges are pointing from its connected components, Llama applies the star pattern strategy on table n_i to generate the plan. For each of its connected components, Llama calls this algorithm to respectively generate their sub-plans (Line 14). All of them are finally joined together based on the primary key of the fact table (Line 15).
- If Table n_i is a dimension table joined with table n_j, that is, an edge is pointing from n_i to n_j, Llama splits the graph G into two components G_1 and G_2, each of which contains n_i and n_j respectively (Line 18). They call this algorithm to generate their sub-plans (Line 19) and finally join them by the joined key (n_i's primary key).

The cost estimation during plan generation is focused on the I/O cost incurred in initialization, shuffle and output. The calculation is similar to the overhead analysis in Section 3.3. In addition, the cost estimation also needs to check whether the required PF group exists in the plan. If not, the overhead to generate the proper PF group should be taken into consideration. Note that the $Join()$ in the algorithm which joins decomposed components increases the depth of the plan tree by 1, implying that one more MapReduce job is needed during the query processing. To make the plan tree compact, we call the $flattenPlan()$ function after the original plan is generated. This function combines two MapReduce jobs when the partition key of one job is a subset of its subsequent job. After flattened, their join operations are performed in the same phase to reduce the number of MapReduce jobs and further reduce the intermediate I/O cost. However, it is worth noting that, the flattened plan may not be better than the original one, because generating the proper PF group is also a MapReduce job and its overhead needs thorough consideration.

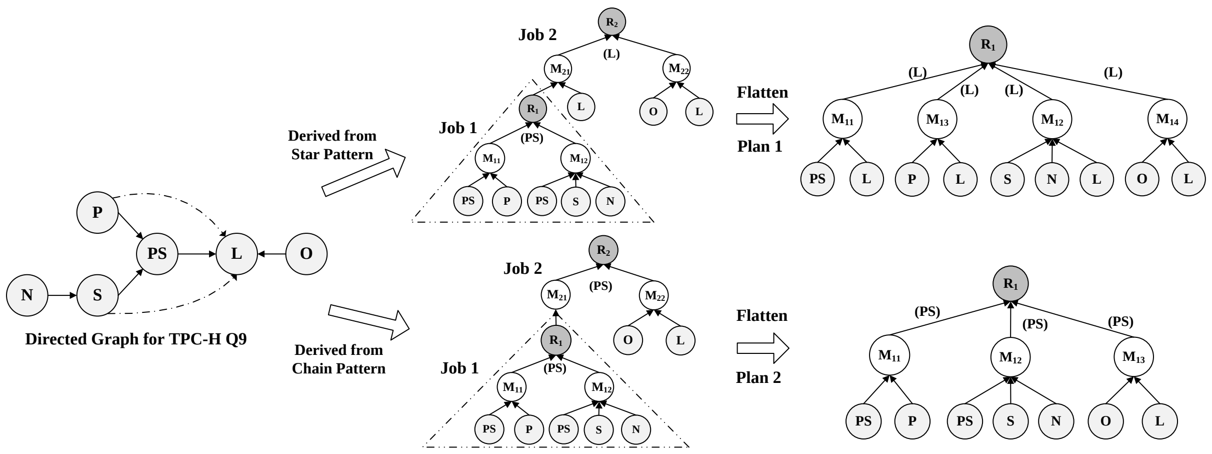

Directed Graph for TPC-H Q9
Derived from Star Pattern
Derived from Chain Pattern
Job 1
Job 2
Flatten
Plan 1
Plan 2
(L)
(PS)

Figure 4.3: Execution Plan for TPC-H Q9

4.4 A Running Example

Taking the query Q9 from the TPC-H benchmark as an example, we illustrate how the query plan is generated and executed in Llama using the proper vertical groups. Q9 determines how much profit is made on a given line of parts, which is expressed as $Lineitem \bowtie Orders \bowtie PartSupp \bowtie Part \bowtie Supplier \bowtie Nation$. For simplicity, we use L, O, PS, P, S and N to represent the corresponding tables. The query graph of Q9 is shown in Figure 4.3.

The size of table N is small enough to be fully buffered in memory and hence can be joined with other tables by using the fragment-replication method. The original query is thus reduced to $L \bowtie O \bowtie PS \bowtie P \bowtie S$. Next, by analyzing the graph, the edges from PS and O point to L. L is thus treated as a fact table. Splitting the graph in L by the star pattern strategy, we obtain two components connected to L: $P \bowtie PS \bowtie S$ and O. The similar approach could be adopted in the first component by using PS as the fact table. Before flattening the plan, there are two MapReduce jobs to complete the query. The first job performs $J_1 = (PS \bowtie P) \bowtie (PS \bowtie S \bowtie N)$ and the second job performs $J_2 = (J_1 \bowtie L) \bowtie$

$(O \bowtie L)$. Since the partitioning key in the first job is *PS*'s primary key, which is the subset of the partitioning key (*L*'s primary key) in the second job, it is possible to flatten the plan by combining the join operations in the two jobs. Each dimension table is joined with L directly. Therefore, $L \bowtie J_1$ can be transferred into $(PS \bowtie L) \bowtie (P \bowtie L) \bowtie (S \bowtie N \bowtie L)$.

As an optional approach, Llama can use the chain pattern by cutting the edge from *PS* to *L* to split the graph into two separate components: ($P \bowtie PS \bowtie S \bowtie N$) and ($L \bowtie O$). These two components are joined via the primary key of *PS*. For the first component, there are two individual parts, ($P \bowtie PS$) and ($N \bowtie S \bowtie PS$), both of which could be completed in different map-merge joins. Their intermediate results are joined via the primary key of *PS* as well. Because these three parts ($L \bowtie O$), ($P \bowtie PS$) and ($N \bowtie S \bowtie PS$) are joined by the same key (*PS*'s primary key), it is possible to join them in the same reduce phase according to *PS*'s primary key. This makes the plan more compact to be completed in one MapReduce job. In this flattened plan, there are three types of mappers and one type of reducer. The mappers respectively finish $PS \bowtie P$, $PS \bowtie S \bowtie N$ and $L \bowtie O$.

Both of these plans are able to be performed in one MapReduce job if the corresponding *PF* groups are available. Because *L* and *PS* are sorted by the foreign keys from *O* and *P*, the first plan needs to build two additional *PF* groups of table *L*, which are sorted by the foreign key from *P* and *S* respectively. The second plan needs one more *PF* group of table *PS* that is sorted by the foreign key from *S*. If such *PF* group does not exist, the plan generator will globally consider the cost of group creation in addition to the overhead of executing the plan. For the TPC-H Q9, the second plan is preferred because it only needs to initialize one *PF* group of table *PS* with less execution overhead.

In the MapReduce framework, it is generally preferred to choose a bushy tree plan rather than a right-deep tree plan, because results of each job have to be replicated in DFS with a high I/O cost. Bushy tree plan effectively improves the joins parallelism and more joins could be completed in a single MapReduce job.

4.5 Fault Tolerance

Since we rely on the run-time environment of Hadoop, Llama preserves the fault-tolerance properties of the MapReduce framework. As earlier mentioned, Llama tries to push more joins in map phase to reduce the number of jobs. If one of the map tasks fails, job tracker could schedule the task on another machine. The input of each map task is read from the DFS. As is the case in the MapReduce framework, DFS ensures the safety of data from node failures.

In addition, it is easy to control the granularity of each map task by assigning appropriate size of data. That is, the input size of mappers could be adjusted by job tracker, which balances the performance and recovery cost if failures are encountered.

Chapter 5

Implementation of Llama

As discussed in previous sections, the efficiency of Llama lies in its column-wise storage and the concurrent join, which require to materialize the tuples and perform different joins in mappers and reducers. In this section, we first present our customized implementation for data accessing in the map phase. Then we illustrate our approach to facilitating variant types of processing procedure in one job. Finally we show how Llama performs the join on heterogeneous data from different kind of mappers in the reduce phase.

5.1 InputFormat in Llama

Llama is built on top of Hadoop 0.20.2 and employs *InputFormat* to handle various types of input. When a MapReduce job launches, InputFormat is called to sequentially perform the following executions:

- Validate the input specification of the job, such as the existence of file/directory and their access authorization.

- Split the input datasets into logical partitions, each of which corresponds to a portion of data and is later assigned to an individual mapper.
- Provide specific reader to read the corresponding partition according to the record-boundaries and present a record-oriented view of the logical InputSplit to the individual task.

To enable the mappers to handle column materialization and map-merge join, we implement *MaterializeInputFormat* (MIF) and *JoinInputFormat* (JIF) that extend InputFormat of Hadoop.

MaterializeInputFormat (MIF)

To split certain columns of a table into logical partitions, MIF obtains the meta information of these columns, such as the number of records and the corresponding path address. It then creates a list of partitions which is called InputSplit in Hadoop and subsequently assigns these splits to different mappers. Each input split is represented as $(columnset, s, e)$. Here *columnset* denotes the columns to be materialized; s and e denote the start and end sequence number in the column(from the s-th value to the e-th value). A specific reader named *MaterializeReader* is provided to read such input split. Based on the start and end sequence number, the reader first calculates the corresponding block position of each involved CFile in the *columnset*. After seeking to the right position, it begins to scan the values from the corresponding files and merge them into a tuple. The merged tuple will be pipelined to the mappers as input.

Different from the traditional approach that uses the file offsets in the logical input split, sequence number is used in Llama. The reason is that, if Llama uses the file offsets to indicate the logical input split, JobTracker has to read multiple CFile's index to calculate the begin offset for each mapper. Since each table contains multiple columns and each CFile has an individual index, reading the index from different CFiles means to create the connection and transfer the data from different datanodes to JobTracker. This network overhead in JobTracker is significant especially when the number of files increases. In addition, JobTracker has to calculate the offset and generate the split information after reading the index. This makes JobTracker becomes a performance bottleneck during the job. As an alternative, if we use the sequence number to indicate the logical split, and delay the offset calculation performed to the beginning of each maptask, JobTracker is able to analyze and assign the tasks more efficiently. This approach thus facilitates Llama a better scalability for the large data processing.

JoinInputFormat (JIF)

MIF materializes CFiles of the same table, while JIF processes CFiles of different tables. JIF is designed to facilitate the map merge join, namely combining tuples from different tables sorted by the joining key. As a result, this input format requires that the input tables are already sorted by the joining key. Normally, the inputs of JIF are a fact table and a dimension table. JIF uses the fact table as the base table to calculate the logical partition, in which the meta data of the dimension table is included. The tuples of the dimension table are dynamically read in terms of the joining key of the fact table.

The record reader called *JoinReader* is provided by JIF. It not only materializes the tuples but also joins the tuples from different tables. When the reader receives its corresponding partition of the fact table, it first seeks to the start position of the fact table, and reads the first tuple in the current position. According to the join key of this tuple, it locates the start position of the corresponding values in the dimension table by a binary search in the index. Subsequently, it performs a merge join by sequentially scanning both the fact and dimension tables. The result tuples are pipelined to the mappers. If the join selectivity is high, Llama exploit the index of the dimension table to seek to the corresponding block, which further reduces the scan cost.

To privide an efficient access in a MapReduce job, Hadoop tries to assign the input split to the proper nodes, in which stores a corresponding replication of the data of that split. The data locality can reduce the data transformation through network for better read performance. It is worth noting that, JoinReader has to read and join multiple CFiles, which are likely stored in different data nodes. In this case, our splits assignment strategy only considers the locality of the fact table. In order words, we may have to remotely access the dimension table during the join processing. Although this causes certain network overhead, but the overall performance is still better than the traditional Reduce-Join approach. The reason is that, the Reduce Join approach still needs the network overhead in the shuffle phase. Moveover, it suffers additional I/O cost to write the intermediate result in the map phase before shuffling to the reducers.

According to modify the strategy of distributing data block in HDFS, it is possible to make most data block of the dimension table store in the same node of the corresponding fact table. However, this approach is not recommended for two reasons:

1. It violates the design philosophy that the data model should be independent of the underlying file system. That is, the implementation details of underlying data storage

system should be transparent to the upper applications.

2. A small fact table may be associated with several small dimension tables. In this case, it is difficult to place all these blocks in one data nodes when considering the fault tolerance and load balance at the same time.

5.2 Heterogeneous Inputs

To facilitate different types of processing procedure in one MapReduce job, we implement *LlamaInputs* which specifies particular inputformat, mappers, combiner and partitioner in terms of different datasets. Since it provides more flexible user defined configurations to handle heterogeneous data input, it is superior to *MultipleInputs* in Hadoop [4] and *TableInputs* in [48]. To simplify the deployment of customized interfaces, it is defined as follows:

```
  int LlamaInputs.addInput(configuration, dataset,
inputformat, mapper, combiner, partitioner);
```

In this interface, $configuration$ describes the job configuration in the MapReduce environments. *dataset* indicates the table(s) to be processed in this specific mapper and is presented as String. If the dataset is a row-wise table, it is the corresponding path. Otherwise it is a combination of the paths information from the related CFiles that need to be accessed. $inputformat$ is the particular input format to handle different data sources. It can be MIF or JIF for the columnar storage as previously mentioned, or the traditional input formats in Hadoop such as the TextInputFormat and SequentialFileInputFormat. *mapper*, *combiner* and $partitioner$ are respectively used to filter particular tuples, combine the intermediate results, and define the partition strategy in the shuffle phase. It is noted that the *partitioner* inherits the one-to-many capability as presented in [48]. That is, each record in the mapper is able to partition to one or more different reducers by specific partitioner. This feature thus facilitates the function of map join reduce as described in [48].

The return value of this interface is an integer, which binds the input dataset with the customized processing. Once this function is called, the customized details associated with that integer are recorded in the *configuration*, which is further transferred to different mappers and reducers. when the MapReduce job is launched. According to this integer, Llama can retrieve the customized process for the particular dataset. The proper input format is used to materialize or join the tables. Then it filters the materialized tuples, combines the qualified tuples, and shuffles these intermediate results to the reducers by the specific mapper, combiner and partitioner.

5.3 Joiner

To join heterogeneous data sources in the reduce phase, we introduce a MergeJoiner before the reducer as the Map-Join-Reduce proposal [48] as presented in Figure 5.1. As can be seen, the ReduceCopier is responsible to pull the intermediate results from all the mappers through the network. The MergeJoiner is an optional component in the reduce phase and may contain one or more join processing. Each join processing requires a merger and a joiner. The merger is to merge and sort the dataset. The sorted data is passed to the following joiner as iterator. In the joiner, since the data is already sorted in different datasets, it is easy to define and perform the join process in the joinner. After all the join operations

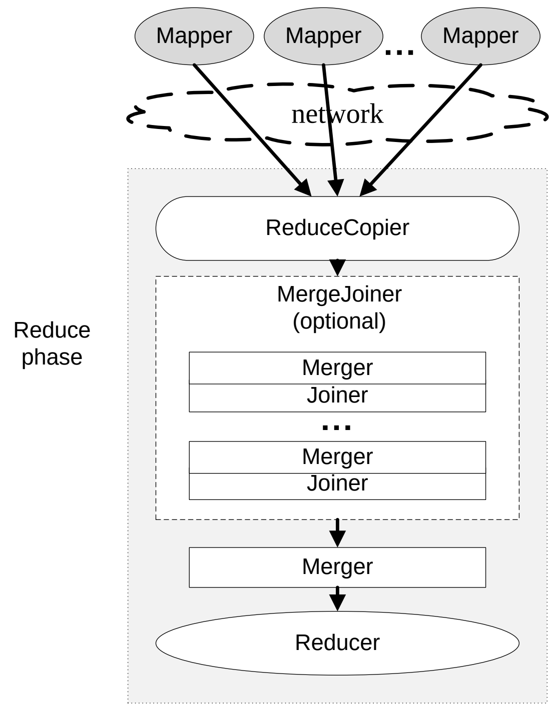


Figure 5.1: Execution in Reduce Phase

is finished, the joined results are sorted by the merger and are further transferred to the reducer for the final aggregation.

The join processing logic is specified by creating a joiner class and implementing specific join() functions. Joiners are registered to the system by the following function:

```
int LlamaInputs.addJoiner(int[] tableID, Joiner joiner);
```

In this function, *tableID* indicates two or more tables to be joined by the same joining key. They can be either the return integer from LlamaInputs.addInput() or the intermediate join result from LlamaInputs.addJoiner(). *joiner* provides the join() function to join the indicated tables before the reduce phase. Before the join processing, data from these tables is sorted by the joining key. The return value of this function is the intermediate join result. Therefore, the joiners can be chained and each reducer can perform one or several join operations in sequence. It is worth noting that in one join operation, there may be more than two table sets to be joined if these table sets share the same join key. For example, *Lineitem*, *Part* and *PartSupp* can be joined by the *partID* in one joiner.

As Llama inherits and extends the joiner from the previous proposal [48], it is able to complete multiple joins of different join keys in the reduce phase. To enable this function, we also slightly modify the structure of ReduceTask. It holds an array of counters for the join operation and store the table id that required to join in the reduce phase. When all the needed partition for a table is read, it sorts these partitions and then perform the join function.

Chapter 6

Experiments

In this section, we study the performance of Llama. We first compare the data storage with the existing file formats in Hadoop. Then we study the column materialization in the MapReduce framework. Finally we benchmark our system against Hive with TPC-H queries. We choose Hive for comparison for three reasons: First, Hive is built on top of Hadoop and provides efficient performance on large scale data analysis. Second, Hive has benchmarked itself with Hadoop and Pig using the TPC-H benchmark. It provides scripts with reasonable query plans and execution parameters. Third, Hive provides the column-wise storage format *RCFile*, which could be adopted in the HiveQL and compared with our approach to the column-wise storage.

6.1 Experimental Environment

We conduct our experiments on Amazon EC2 cloud with large instance. Each instance has 7.5 GB memory, 2 virtual cores with 2 EC2 compute units each. There are 850 GB local storage in two local disks for each instance. The operating system is 64-bit Linux. In our configuration, one instance is used as the NameNode and JobTracker, to manage the HDFS cluster and schedule the MapReduce jobs. The others are used as the DataNodes and TaskTrackers to store the files and execute the tasks assigned by the JobTracker.

We implement Llama on top of Hadoop v0.20.2, and use Hive v0.5.0 for comparison purposes. Based on the hardware capacity, we adjust the maximum number of mappers and reducers to 2. We assign 1024 MB memory for each task. Chunk size in the HDFS is set to 512 MB and the replication factor is 3. Following the suggestion of the Hive TPC-H experiments, we set the replication factor of the intermediate results to 1; that is, the results of intermediate jobs are stored in 1 node instead of 3 nodes, to save the replication cost. The number of reducers is automatically configured by Hive's default setting.

Both Hadoop and Llama store the files in HDFS. We use dbgen in TPC-H to generate the synthetic dataset. We benchmark the performance with the cluster size of 4, 8, 16, 32 and 64 nodes. Each node stores 10 GB TPC-H data. The size of TPC-H data are thus 40 GB, 80 GB, 160 GB, 320 GB and 640 GB respectively.

6.2 Comparisons Between Files

In this test, we compare the performance of different file formats implemented on the Hadoop system. They are TFile of Zebra [15], RCFile of Hive [7], HFile of HBase [5] and our CFile. First we compare their compression performance in terms of their compression speed and compression ratio based on the three general compression algorithms, namely BZip2, Gzip and Lzo [10]. Then, we compare their performance on write, sequential scan, and random reads. All these operations are run on the HDFS. The original dataset is the *Orders* table in TPC-H schema (15M tuples with 9 columns, 1.75 GB uncompressed text data when TPC-H scale is 10). The specification of the *Orders* schema are provided in Table 6.1. Our objective here is to demonstrate that the CFile format has distinct advantages in large scale data processing.

Table 6.1: Orders in TPC-H

Column Name	Type	Length (bytes)
OrderID	Variable Length Long	1 to 8
CustID	Variable Length Long	1 to 8
OrderStatus	Fixed Length Text	1
TotalPrice	Double	8
OrderDate	Fixed Length Text	10
ShipPriority	Fixed Length Text	5, 6, 8, 8, 15
Clerk	Fixed Length Text	15
ShipPriority	Fixed Length Text	1
Comment	Variable Length Text	50 in average

In the experiment, we first parse the data into columns and write them by specific writer of these files. HFile and TFile use one column family to store all the columns. In HFile, one tuple is parsed into multiple key-value pairs, and each pair represent a value in one column. Its key is a composite of orderID and the column qualifier, its value is the data in the corresponding column. No timestamp is included in our experiments. In TFile, one tuple is parsed into one key-value pair. Its key is orderID, and its value is a byte array of the other columns, which is in row-wise format. For comparison purposes, we use TFile to simulate the column-wise storage. That is, each file represents one column family. The key is the orderID and the value is the specific value of that column. This key is necessary to combine different columns, when random accessing several columns from different files. In our experiment, we use *TFile-one* and *TFile-multi* to represent these two storage approaches. Both RCFile and CFile provides proper interfaces to write different columns. We try our best to maximize the performance of all these existing file formats. The parsing here is to calculate the offset information of each columns. Pre-created buffer is used to avoid the expensive temporary object creation during the whole I/O processing. The data is block compressed and stored in the HDFS. TFile and HFile only support Lzo and GZip. Therefore, the results of TFile and HFile presented below do not contain BZip2.

Storage Efficiency Figure 6.1 shows the size of files compressed based on different compression algorithms. The results show that BZip2 compressed the data with a higher ratio than Gzip and Lzo. CFile and RCFile are smaller than other file formats, because they both compress the data in columns. On the other hand, since HFile and TFile-multi contain the row-id for each record, their sizes are relatively larger. Moreover, HFile has to store the column qualifier and is thus larger.

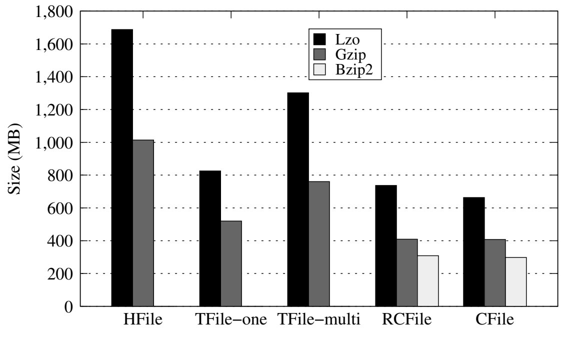


Figure 6.1: Storage Efficiency

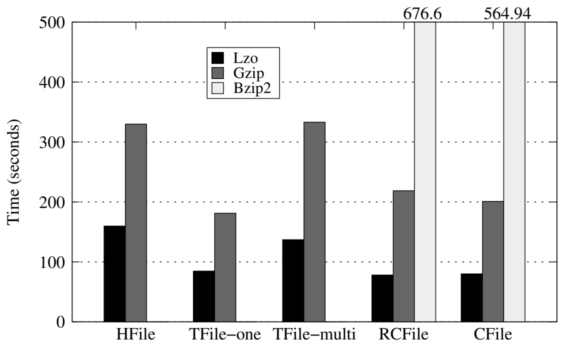


Figure 6.2: File Creation

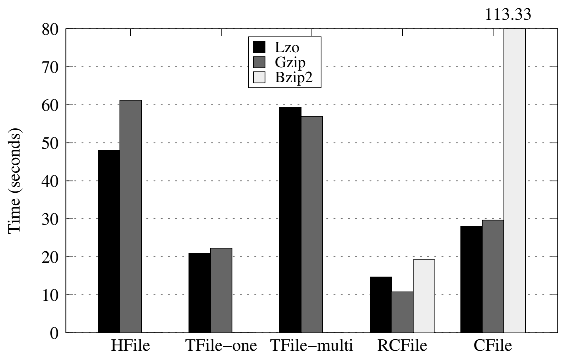


Figure 6.3: All-Column Scan

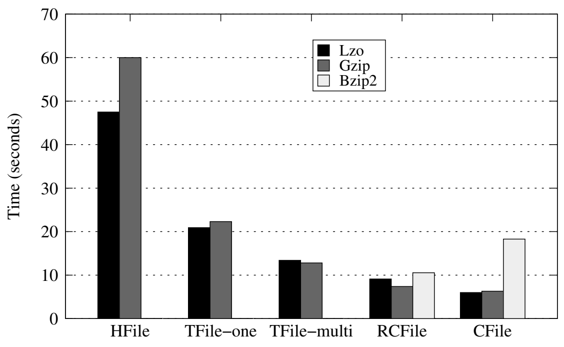


Figure 6.4: Two-Column Scan

Creation Overhead Figure 6.2 presents the creation overhead of different files for table Orders. Although the compression ratio of Lzo is not as good as Gzip and BZip2, its compression speed is much faster. Their write overhead is primarily proportional to the size of the file. As such, the creation of CFile is an efficient process.

Access Performance Figure 6.3 shows the performance of scanning all the columns. Similar to compression, Lzo decompresses faster than the other algorithms. CFile is not as good as RCFile and TFile-one, because CFile has to read all the columns from different files and decompress them individually.

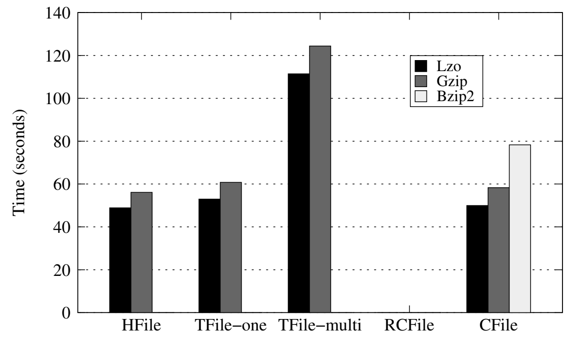


Figure 6.5: Two-Column Random Access

Figure 6.4 shows the speed of scanning the 2nd and the 7th columns in Orders. TFile-one and HFile are essentially the row-wise storage in one column family. Therefore they need to read all the data and decompress them as scanning all the columns. RCFile, on the other hand, groups each column on a separate mini block. This approach avoids decompressing the unnecessary columns but incurs some seek operations. Different with the above formats, TFile-multi and CFile only read the required columns and thus obtain a better performance. Moreover, CFile outperforms TFile-multi because it is designed for column access with more compact storage.

Performance of randomly accessing 10000 records with the two same columns is reported in Figure 6.5. RCFile is designed for sequential scan without providing the interface for random accesses, thus its corresponding results are not captured in the figure. Although CFile has to perform two seeks to random access two columns, its performance is as good as HFile and TFile-one which performs only one seek.

CFile shows its competitive performance than the existing file formats for large scale data processing. In terms of storage efficiency, it is more compact with less storage overhead. In execution, even though it is not as good as RCFile in terms of scanning all the columns,

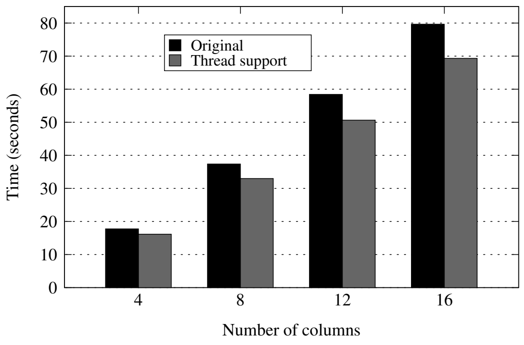


Figure 6.6: Comparison in Read

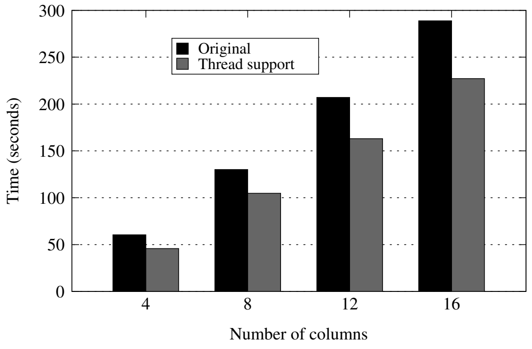


Figure 6.7: Comparison in Write

its speed is fastest when only a few columns have to scanned. Therefore, it is particularly suitable for the analysis that only requires a few columns, but not the whole table. In addition, CFile provides the random access efficiently similar to that of HFile and TFile.

6.3 Experiments on CFile

Impact of Multi-thread Because Llama requires to access multiple CFiles under the column-wise storage, we try to find out whether we can take advantage of multi-thread to improve the performance when accessing multiple CFiles. We maintain the same interfaces such as "*boolean next(Writable)*" in the CFile.Reader or "*boolean append(Writable)*" in CFile.Writer. Meanwhile, we adopt the producer-consumer pattern inside the CFile by leveraging the multi-thread. For the read operation, the read thread is treated as producer, and the user who gets the data via the next() function is treaded as consumer. Each reader has two pieces of input buffer. When buffer *A* is empty, the read thread pre-fetches the data from HDFS. After buffer *A* is filled, it is labeled as *available*. At the same time, the consumer reads the data from the other available buffer *B*. If *B* is exhausted and *A* is available, consumer switches to read from A. If both A and B are empty, the consumer has to wait until one of the buffers is available. Similarly, for the write operation, the user is treated as producer and the write thread is viewed as consumer.

In this experiment, we compare the differences of the file creation and scan in single thread and multi-thread environments. Each column has 10^7 values and is blocked compressed in different CFiles. One column is the long-type primary key, and the other columns are random generated 32-byte strings. The results in different numbers of columns are shown in Figure 6.6 and Figure 6.7. Under the support of multi-thread, the reading and writing performances respectively improve about 14% and 27%. The writing improvement is more obvious than reading. The reason is that, DFSOutputStream synchronizes the dataQueue if a writer is flushing the data into HDFS. When multiple files are being written into HDFS in the scenario of single-thread, each writer has to compete for the synchronization lock. Some writers are thus blocked, and the append operations have to wait until their writers finish flushing. Utilizing multi-thread helps to reduce this contention. For the reading operation, there is no this synchronization conflict. That is why the writing improvement is more obvious than reading.

Data Locality In this experiment we measure the impact of data locality. We replace the original HDFS into our modified version, which distributes the CFiles of the same partition to the same data nodes. Then we run the scan in the nodes that stores the data, ensured that no CFile had to be remotely accessed. We also run the same scan in the name node, meaning that all the files are remotely read.

Figure 6.8 presents the impact of data locality. As the number of columns increases, the

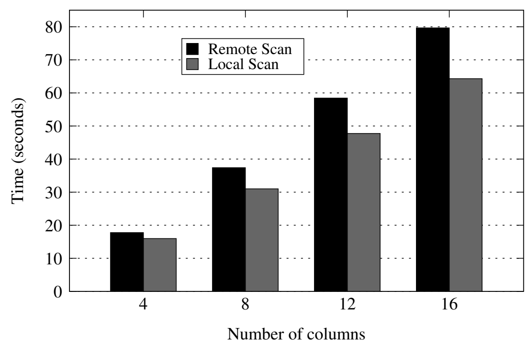


Figure 6.8: Data Locality

overall performance improves from 11.2% to 23.9%, because more data has to be remotely transferred. This ratio is slightly better than multi-thread. On the other hand, although the data locality can improve the performance, we suggest that the underlying file system should be independent to up-level applications.

6.4 Integration to epiC

In this section, we integrate our columnar storage to ES^2 and compare the performance of columnar storage to the existing PaxFile layout in the ES^2. After loading the data to the distributed file system, we study their performance in scan and random access in the distributed environments with a various number of columns. In the experiments, the HDFS cluster contains 9 nodes. 1 node is the NameNode and the other 8 nodes are the DataNodes. 16 mappers are launched to execute the tasks in parallel, and the overall execution time is reported when all the mappers complete the tasks.

Performance of Write In the loading processing, we launch 16 mappers to respectively write 10^6 tuples to the HDFS. As such, the table contains 1.6×10^7 tuples in total. Each

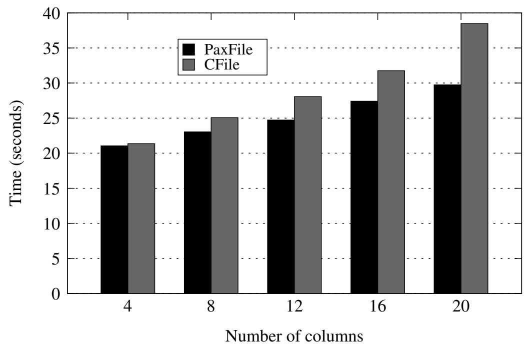


Figure 6.9: Performance of Write

tuple contains 4, 8, 12, 16, and 20 columns. The first column is the primary key of 8-byte string, and the other columns are randomly generated 32-byte strings. The goal of this experiment is to compare the writing performance when the system is flushing the buffer to the HDFS in parallel. According to the results reported in Figure 6.9, we find that the flushing performance of PaxFile is slightly better than CFile as the number of columns increases. One reason is that, each mapper has to write several CFiles into HDFS, requiring more overhead of network communication and maintenance. In addition, when the number of columns increases, the increasing time of PaxFile is less than CFiles. It shows that the overall writing performance of PaxFile is slightly better than CFile especially when the number of columns is large.

Performance of Scan In the scanning processing, each mapper scans a portions of the 20-column table, which is loaded in the previous experiment. This experiment compares the performance of different storage layouts in the large scale data analysis. As illustrated in Figure 6.10, when there are only a few of columns to be access, the advantage of CFile over PaxFile is obvious. One the other hand, this advantage reduces as the number of columns increases. Moreover, PaxFile outperforms CFile when scanning most of the columns.

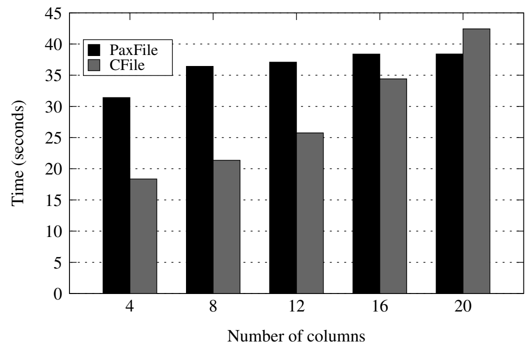


Figure 6.10: Performance of Scan

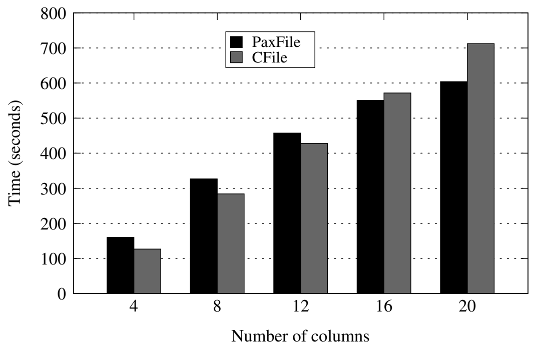


Figure 6.11: Performance of Random Access

Performance of Random Access In this experiment, each mapper randomly accesses 5000 records by the primary key. The original table is previously loaded with 20 columns. Its goal is to investigate the difference when the system responses the queries of random access and the result is shown in Figure 6.11. Similarly, with a small number of access columns, the overall performance of CFile is slightly better than PaxFile, because CFile minimizes the I/O overhead. As the number of columns increases, CFile has to access multiple individual files and thus incurs more transmission overhead. On the other hand, PaxFile stores several columns in one physical block. It thus incurs just a few more overhead especially when the column number increases from 16 to 20.

6.5 Column Materialization

In this experiment, we first estimate the cost ratio discussed in Section 3.3. Then we run a simple aggregation query to study the column materialization within the MapReduce framework. Based on our experiment, we find that LM is preferred when the selectivity is very high.

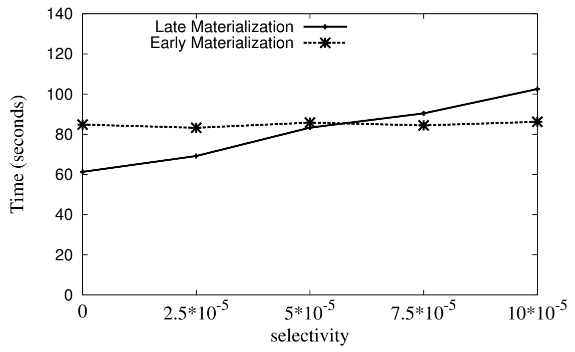


Figure 6.12: Column Materialization

To estimate the cost ratio of $r_1 = r_{random}/r_{scan}$ in the cost model, we compare the average running time of scanning versus random accessing. By examining at the execution time presented in Figure 6.4 and Figure 6.5, r_1 is about 1.5×10^4. To estimate the cost ratio of $r_2 = r_{shuffle}/r_{scan}$, we run two MapReduce jobs that write no results back to the HDFS. The first job is a map only job that scans a large data set without any processing. Its running time t_1 is thus proportional to the scanning overhead. The second job is a MapReduce job that shuffles all the input data to the reducers without any filtering. Its running time t_2 is thus proportional to both scanning and shuffling overhead. Therefore, r_2 is approximately $(t_2 - t_1)/t_1$. Based on the running time of these two jobs on EC2, we estimate that r_2 is about 3. Obviously, the overhead of random access is much larger than scanning and shuffling. If the column size is small, the predicate discussed in Equation 3.6 can be further reduced to $1/r_1$. That is, LM is picked only when the selectivity is smaller than $1/r_1$. To verify this estimation, we run a simple aggregation query to study when LM outperforms EM by adjusting the constant x below:

```
select sum(l_price), sum(l_discount), sum(l_quality), sum(l_tax)
```

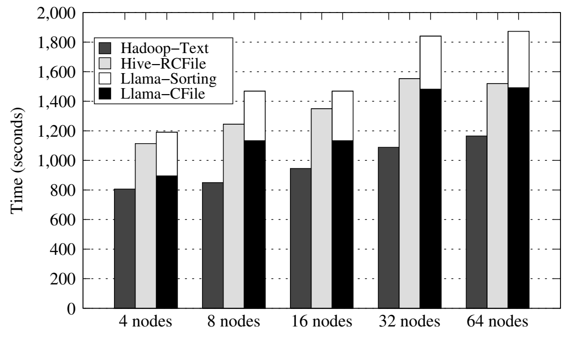


Figure 6.13: Load time

```
from lineitem where l_orderkey < x
group by l_orderkey;
```

The performance of these two strategies are summarized in Figure 6.12. LM is better than EM only when the selectivity is less than 6×10^{-5}. Moreover, as the selectivity grows, the running time of LM increases rapidly. When the selectivity is low, EM is therefore preferred. Since the selectivity in TPC-H is low, Llama mainly employs EM for the following TPC-H queries.

6.6 Data Loading

We report the loading time of all the tables in the TPC-H benchmark in Figure 6.13, and briefly describe our loading procedures here. First we use dbgen to generate the TPC-H data for the different local disks of the cluster. Then we use Hadoop's file utility "copyFrom-Local" to upload unaltered TPC-H text data from the local disk to the HDFS in parallel. It directly copies the text without parsing the data. The HDFS automatically partitions each file into blocks and replicates to three data nodes.

In Hive, we transform the raw-text to RCFile by HiveQL. For example, if there is raw text data located in directory '/*A*' and the schema is (int, int), we could use the following HiveQL to complete the format transformation.

```
CREATE EXTERNAL table A (a1 int, a2 int)
    STORED AS TEXTFILE LOCATION '/A';
CREATE table B (b1 int, b2 int) STORED as RCFILE;
INSERT OVERWRITE table B SELECT a1, a2 FROM A;
```

The first two commands are to declare the meta data information such as the schema and data location. The third command is to execute the specific transformation. In our experiment, RCFile is block compressed by Lzo.

In Llama, we transform the raw-text to CFile. To build the basic group, map-only job is launched to read the data from the HDFS. It parses the text records into columns guided by the delimiters, and writes each column to the corresponding CFile. The black part of the graph in Figure 6.13 indicates the processing time required to build the basic group of TPC-H dataset. The white part of the graph indicates the additional cost for building two *PF* groups: PF group of *Partsupp* sorted by *supplierID*, and the *PF* group of *Orders* sorted by *customerID*. The first *PF* group is to facilitate the map-merge join of TPC-H Q3 while the second one is to facilitate the map-merge join of TPC-H Q9 and TPC-H Q10.

As shown in Figure 6.13, Llama performs slightly better than Hive if we do not take the sorting time into account. On the other hand, even though the transformation cost of Llama is higher than the pure HDFS copy because of the additional overhead of parsing and sorting, this transformation is worthwhile because it significantly reduces the processing time of the analytical task. As will be seen in the following experiments, the accumulated sav-

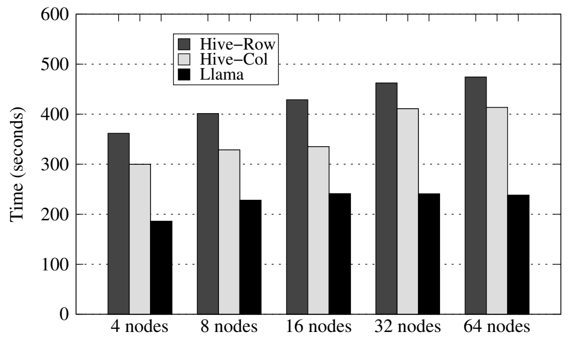


Figure 6.14: Aggregation Task: TPC-H Q1

ings are significantly more than the loading overhead.

6.7 Aggregation Task

We use TPC-H Q1 as our aggregation task to measure the performance improvement gained by adopting the column-wise storage. Q1 is to provide the pricing report for all the lineitems shipped on a given date. Intermediate results have to be exchanged between nodes in the cluster.

The results are shown in Figure 6.14. *Hive-Row* and *Hive-Col* represent the performance of Hive with the same execution plan but on the row-wise and column-wise storage respectively. The results confirm the benefit of exploiting column-wise storage in compression. Under the compressed column-wise storage, both Hive and Llama save the I/O cost. In contrast to the RCFile that stores columns in one file in a record columnar manner, CFile stores each column in an individual file. This guarantees that only necessary data is read, and hence saves more I/O during processing.

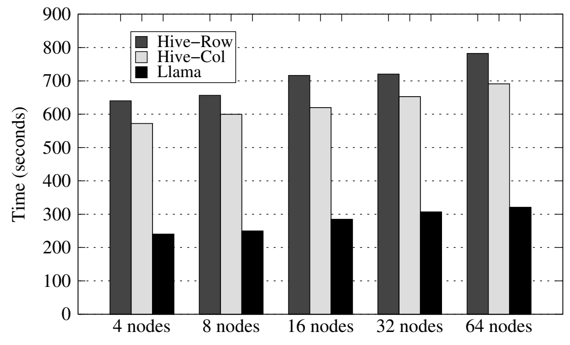


Figure 6.15: Join Task: TPC-H Q4

6.8 Join Task

We choose TPC-H Q4, Q3, Q10 and Q9 as our join tasks. There are one, two, three and five join operations in these queries respectively. They are chosen to study the performance and scalability of the system with respect to the data and cluster size in different join contexts. In Hive, the execution plans for a specific query are the same regardless of the underlying file formats.

TPC-H Q4 Q4 is to determine how well the order priority system is working and gives an assessment of customer satisfaction. It contains one join operation and is compiled into three main MapReduce jobs by Hive. Job 1 creates a temporary table Tmp that contains only distinct qualified key from $Lineitem$. Job 2 joins Tmp with $Orders$ and Job 3 aggregates the results.

Llama processes the query using the similar approach. In the first job, Llama materializes $Lineitem$ and creates a temporary table Tmp with distinct $orderID$. It performs a map-merge join for $Orders$ and Tmp in the second job and aggregates the final results in the last job.

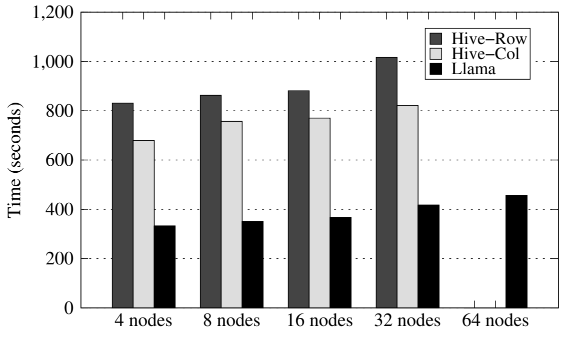


Figure 6.16: Join Task: TPC-H Q3

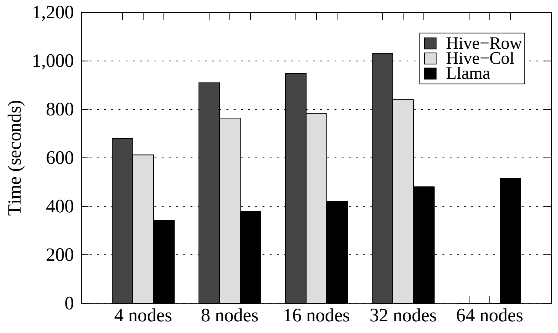


Figure 6.17: Join Task: TPC-H Q10

As shown in Figure 6.15, Llama runs about 2 times faster than Hive. The performance benefit is derived from its column-wise storage. Applying the map-merge join, it further reduces the shuffling cost of the intermediate results. As the query becomes complex in Hive, the benefit of the I/O saving from column-wise storage becomes less obvious because the storage layer only affects the performance in the initial phase.

TPC-H Q3 Q3 is to retrieve 10 unshipped orders with the highest revenue operating on table Lineitem, Orders, and Customer. Hive compiles Q3 into five MapReduce jobs. The first two jobs join the three tables, and other three jobs aggregate the tuples, sort them and get the top 10 results.

Llama processes this job with two jobs. In the first job, there are two types of map tasks, joining *Orders* with *Customer* and materializing *Lineitem*. The intermediate results are shuffled to the reducers by the order key. Reducers perform the reduce-join followed by a local aggregation. The second job combines the partial aggregations for the final answer. To enable the concurrent join, PF group of table *Orders* is built in the loading phase.

Figure 6.16 summarizes the results for different cluster sizes. Concurrent join facilitates a more flexible query plan with fewer jobs, which significantly reduces the job launching time and the intermediate I/O transfer cost, and thus makes it about 2 times faster than Hive. When the data size scales to 640 GB for a cluster size of 64 nodes, one reduce task at the second stage in Hive's job is shuffled more than 30 GB data to process. It ran for a long time and was finally killed after failing to report the status in 600 seconds. Therefore, the execution time of Hive is not reported in the graph for the cluster size of 64.

TPC-H Q10 Q10 is to identify customers who might be having problems with the parts that are shipped to them. This query contains three joins operations on *Customer*, *Orders*, *Nation* and *Lineitem*. Compared to TPC-H Q3, table *Nation* is involved in this query. Hive compiles this query into six MapReduce jobs. The first three jobs perform the joins operations. The subsequent three jobs perform the group, sort and fetch operations.

Instead, Llama transferred this query into three MapReduce jobs. The first job joins the four tables. Two kinds of mappers are launched. One joins table *Customer*, *Orders* with *Nation*, the other one scans table *Lineitem*. Their intermediate results are shuffled via *orderKey*. The second job is to aggregate the previous results by the grouping key. And the last job fetches the first 20 results. Here the aggregation in the second job can not be completed in the first job, because the grouping key is independent to the shuffling key in the first job.

The results are shown in Figure 6.17. Llama is faster than Hive about 2 times. Although there is one more join in this query, the improvement ratio quite similar as in TPC-H Q3. Here are two reasons. First, the grouping key is independent to the joining key in the first MapReduce job. As a result, the group function can not be performed in the first job, and

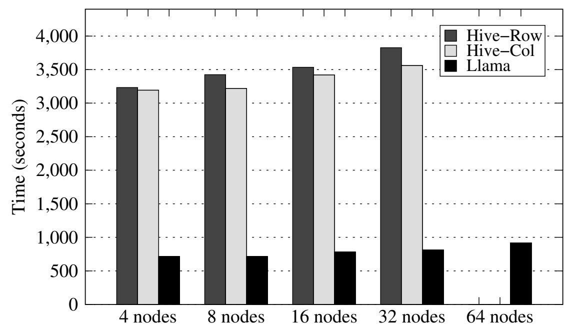


Figure 6.18: Join Task: TPC-H Q9

Llama needs one more job to group the joining results before the fetch operation. Second, the newly increased table is *Nation*, which is too small to make an apparent influence in the hive execution. As a result, the overall execution performance of Hive is quite similar as in TPC-H Q3.

TPC-H Q9 Q9 is to determine how much profit is made on a given line of parts, broken down by supplier's nation and year. Hive compiles Q9 into seven MapReduce jobs. The first five jobs are respectively to join the six tables of the query. After these joins are finished, two additional jobs are launched for aggregation and ordering.

The execution of Q9 in Llama has earlier been presented in Section 4.4. To facilitate the concurrent join, PF group of table *Partsupp* is built during the loading phase.

Figure 6.18 shows the performance of Hive and Llama for Q9 with respect to different cluster sizes. Based on the results, the performance difference between Hive of different storage formats is not very obvious. The main reason is that, Hive configures its number of mappers and reducers by the size of the input dataset. With different file formats, the size of the input varies, which further affects the mappers and reducers in the subsequent jobs. In this case, the overall performance of the subsequent MapReduce jobs slightly deteriorates. Therefore, the benefit of the I/O saving is not apparent in the overall performance. As in TPC-H Q3, Q9 could not be completed in Hive within a specific time frame for the cluster size of 64 nodes.

On the other hand, the concurrent join method is capable of completing all the join in one MapReduce job, which significantly reduce the materialization of intermediate results by the execution plan. As can be observed from Figure 6.18, concurrent join runs nearly 5 times faster than the traditional execution plan.

In summary, Llama achieves a very good scalability when the cluster size increases from 4 nodes (with 40 GB of total data size) to 64 nodes (with 640 GB of total data size). Its scalability is almost linear for large scale data processing.

Chapter 7

Conclusion

In this thesis, we present Llama, a column-wise data management system on MapReduce. Llama applies a column-wise partitioning scheme to transform the imported data into CFiles, a special file format designed for Llama. To efficiently support data warehouse queries, Llama adopts a partition-aware query processing strategy. It exploits the map-side join to maximize the parallelism and reduce the shuffling cost. We study the problem of data materialization and develop a cost model to analyze the cost of data accesses. We evaluate Llama's performance by comparing it against Hive. Our experiments conducted on Amazon EC2 using TPC-H datasets show that, Llama provides the speedup of 5 times compared to Hive. The performance evaluation confirms the robustness, efficiency and scalability of Llama.

Bibliography

[1] At&t news room. `http://www.att.com/gen/press-room?cdvn=news&newsarticleid=26230&pid=4800`.

[2] Customers moving away from batch data warehousing. `http://databasecolumn.vertica.com/database-miscellaneous/batch-data-warehousing-poll/`.

[3] Epic. `http://www.comp.nus.edu.sg/~epiC`.

[4] Hadoop. `http://hadoop.apache.org`.

[5] Hbase. `http://hbase.apache.org`.

[6] Hfile. `http://www.slideshare.net/schubertzhang/hfile-a-blockindexed-file-format-to-store-sorted-keyvalue-pairs`.

[7] Hive. `http://hive.apache.org`.

[8] Hive/tutorial. `http://wiki.apache.org/hadoop/Hive/Tutorial#Joins`.

[9] Join framework. `http://wiki.apache.org/pig/JoinFramework`.

[10] lzo. `http://www.oberhumer.com/opensource/lzo`.

[11] The petabyte age. `http://www.wired.com/science/discoveries/magazine/16-07/pb_intro#`.

[12] Pig. `http://pig.apache.org`.

[13] Tfile. `https://issues.apache.org/jira/secure/attachment/12396286/TFile+Specification+20081217.pdf`.

[14] Vertica. `http://www.vertical.com/`.

[15] Zebra. `http://wiki.apache.org/pig/zebra`.

[16] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[17] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, 2008.

[18] D. J. Abadi, D. S. Myers, D. J. Dewitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, 2007.

[19] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. In *PVLDB*, volume 2, pages 922–933, 2009.

[20] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.

[21] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[22] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *SIGMOD*, 2011.

[23] D. S. Batory. On searching transposed files. *ACM Trans. Database Syst.*, 4(4):531–544, 1979.

[24] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, 2010.

[25] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at facebook. In *SIGMOD*, 2011.

[26] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. Es2:a cloud data storage system for supporting both oltp and olap. In *ICDE*, 2011.

[27] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39:12–27, May 2011.

[28] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, 2006.

[29] C. Chen, G. Chen, D. Jiang, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. Providing scalable database services on the cloud. In *WISE*, 2010.

[30] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, 2007.

[31] E. F. Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[32] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[33] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB.*, 1(2):1277–1288, 2008.

[34] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.

[35] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

[36] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[37] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *VLDB*, 1986.

[38] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1):518–529, 2010.

[39] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. In *PVLDB*, volume 4, pages 419–429, 2011.

[40] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine grace. In *VLDB*, 1986.

[41] G. Graefe and K. Ward. Dynamic query evaluation plans. In *SIGMOD*, 1989.

[42] T. D. Group. Nonstop sql: A distributed, high-performance, high-availability implementation of sql. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, 1989.

[43] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, 2006.

[44] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *ICDE*, 2011.

[45] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. *PVLDB*, 1(1):502–513, 2008.

[46] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD*, 2007.

[47] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[48] D. Jiang, A. K. H. Tung, and G. Chen. Map-join-reduce: Towards scalable and efficient data analysis on large clusters. *IEEE Transactions on Knowledge and Data Engineering*, 2010.

[49] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan data layouts: Right shoes for a running elephant. In *SOCC*, 2011.

[50] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *SoCC*, 2010.

[51] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *PODC*, 2009.

[52] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[53] R. Lee, T. Luo, F. Wang, Y. Huai, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, 2011.

[54] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD*, 2011.

[55] R. MacNicol and B. French. Sybase iq multiplex - designed for analytics. In *VLDB*, 2004.

[56] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB Journal*, 9(3):231–246, 2000.

[57] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[58] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, 2011.

[59] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: continuous pig/hadoop workflows. In *SIGMOD*, 2011.

[60] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, 2008.

[61] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[62] V. Raman and G. Swart. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *VLDB*, 2006.

[63] T. Sandholm and K. Lai. Mapreduce optimization using regulated dynamic prioritization. In *SIGMETRICS*, 2009.

[64] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *SIGMOD*, 1989.

[65] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *VLDB*, 1990.

[66] A. E. Silberstein, R. Sears, W. Zhou, and B. F. Cooper. A batch of pnuts: experiences connecting cloud batch and serving systems. In *SIGMOD*, 2011.

[67] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB*, 2005.

[68] M. Stonebraker, R. H. Katz, D. A. Patterson, and J. K. Ousterhout. The design of xprs. In *VLDB*, 1988.

[69] M. syan Chen, P. S. Yu, and K. lung Wu. Optimization of parallel execution for multi-join queries. *IEEE Transactions on Knowledge and Data Engineering*, 1995.

[70] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *SIGMOD*, 2010.

[71] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst.*, 9(1):133–161, 1984.

[72] Y. Xu, P. Kostamaa, Y. Qi, J. Wen, and K. K. Zhao. A hadoop based distributed loading approach to parallel data warehouses. In *SIGMOD*, 2011.