Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework

Yuting Lin
National University of Singapore
lin36@comp.nus.edu.com

Divyakant Agrawal
University of California, Santa Barbara
agrawal@cs.ucsb.edu

Chun Chen
Zhejiang University, China
chenc@cs.zju.edu.cn

Beng Chin Ooi
National University of Singapore
ooibc@comp.nus.edu.com

Sai Wu
National University of Singapore
wusai@comp.nus.edu.com

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 excellent load performance and its query performance is significantly better than the traditional MapReduce framework based on row-wise storage.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems

General Terms

Design, Performance

Keywords

column store, MapReduce, join

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 data warehousing systems. To deal with data at large scale, special-purpose computations are usually needed for extracting valuable insights and knowledge. Given the massive scale of data, many of these computations need to be executed in a distributed and parallel manner on hundreds or thousands of machines. Furthermore, these computations often involve complex analyzes based on sophisticated data mining algorithms which require multiple data-sets to be processed simultaneously. When multiple datasets are involved, a most common operation that is used to combine 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 proposed for performing database joins in different environments, joining datasets that are widely distributed and are extremely large poses unprecedented research challenges for scalable join-processing in data warehouses.

To carry out data-intensive analysis in scalable and fault-tolerant manner in a distributed environment, Google introduced a distributed and parallel programming framework called MapReduce [21]. The MapReduce framework is highly desirable since it allows the programmer to specify the analytical job and the issue of translating the job into sub tasks on multiple machines is completely automated. Under the umbrella of Apache, an open source implementation of the MapReduce framework, referred to as Hadoop [2] is freely available to both commercial and academic users. Given its easy access, Hadoop has become a popular choice to process big data produced by the web applications and business industry. Furthermore, due to the success of Hadoop and MapReduce, 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. The two major efforts to provide a declarative interface on top of Hadoop run-time environment are the Pig [7] from Yahoo! and the Hive [4] from Facebook.

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 needs a sequential composition of different MapReduce jobs. As we consider the adaptation of the MapReduce framework in the context of analytical processing over relational data in the data warehouse, one of the main extensions that becomes necessary is to support *join* operations over multiple datasets. Processing multiple join operations via a sequential composition of multiple MapReduce jobs is not desirable since it involves storing the intermediate results between two consecutive jobs to the underlying file-system such as HDFS (Hadoop Distributed File System) which would incur very high I/O cost. Recently, several proposals, such as [13], have been made to process multi-way join in a single job. The main idea is when the filtered tuples emitted by mappers are shuffled 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 the

SIGMOD'11, June12–16, 2011, Athens, Greece.

tuple replication during the shuffling phase increases significantly as the number of datasets involved in the query grows.

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. (ii) Reduction of cost of data access by adopting the column-wise techniques [20, 27, 30, 10]. (iii) Improvement of the query processing performance by taking advantage of the column-wise storage. Llama is part of our *epiC* project [18, 1] which aims to provide an elastic, power-aware, data-intensive cloud computing platform. As we develop different components in parallel, we first propose Llama in the context of MapReduce and we will incorporate Llama into epiC later on.

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. 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 paper are as follows:

- We propose a new column-wise file format called *CFile*, which provides better performance than currently available file formats in Hadoop for data analysis.
- We analyze data materialization strategies over this format and develop a cost model to estimate the cost of data access in the MapReduce framework.
- 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 I/O costs.
- 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.

The remainder of the paper is organized as follows. In Section 2 we depict the fundamental single join approaches in MapReduce. In Section 3 we describe the column-wise data representation used in Llama, which is important for processing concurrent joins efficiently. In Section 4 we exploit the issue of data materialization and develop a cost model to analyze the materialization cost for the column-wise storage. In Section 5 we illustrate the detailed design of concurrent join. A plan generator is designed to generate efficient execution plans for complex queries. In Section 6 we present the detailed implementation of Llama. We evaluate Llama system by comparing it with Hive on the basis of the TPC-H benchmark in Section 7. We review the related work in Section 8 and conclude the paper in Section 9.

2. SINGLE JOIN IN MAPREDUCE

Single join has been well studied in the MapReduce environment [6, 5, 16]. Both Pig [7] and Hive [4] provide several join strategies. In general, there are three common approaches for processing a single join in the MapReduce framework:

1. Reduce Join. It is the most common approach to process the join operation in the MapReduce framework. The two tables that are involved in the join operation are scanned during the map phase. 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 in the traditional databases. It requires shuffling data between the map and the reduce phase. This approach is referred to as *Standard Repartition Join* in [16].

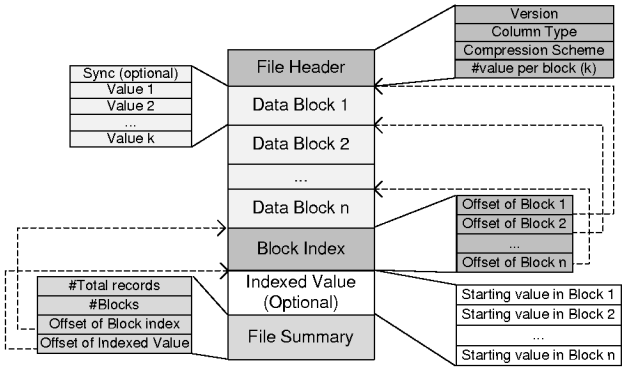


Figure 1: Structure of CFile

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 tables are required to be sorted in advance. This approach is referred to as *Broadcast Join* in [16].

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 [16]. 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.

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 group 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 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 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. 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 (i) For frequent random access, even applying index cannot provide a satisfactory performance. (ii) 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.

Compared to TFile of Pig [7] or RCFile of Hive [4], 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. 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.

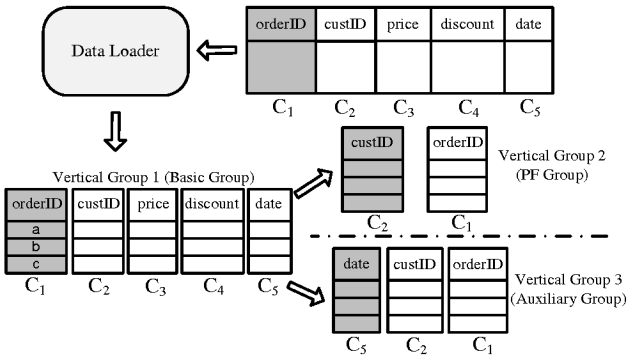


Figure 2: Data Transformation

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 [30].

Each 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. 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. For example, Group 1 in Figure 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 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 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 sorted by *date* at runtime. In this way, Llama can answer the queries without scanning the entire datasets of the corresponding basic groups.

To update a particular vertical group when there are a batch of new records, Llama first extracts the corresponding columns of those new records. Then it sorts the records via the specific column and writes the sorted results to a temporary group. This temporary 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 [17].

4. DATA MATERIALIZATION IN LLAMA

In the column-wise data model, there are two possible materialization strategies during the query processing [11]: 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 [24].

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:

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

As illustrated in Figure 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

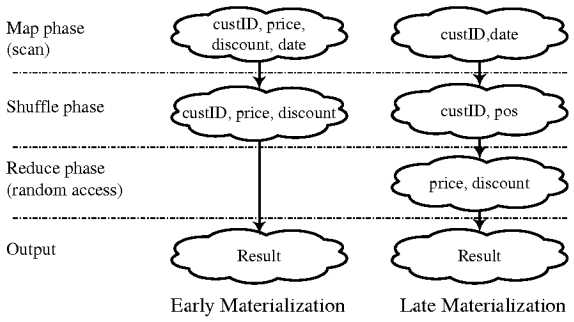


Figure 3: Example of Materializations

$$C_{scan}(S) = |T| \times r_{scan} \times \sum_{\forall c_i \in S} Size(c_i) \quad (1)$$

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

$$C_{shuffle}(S) = f \times |T| \times r_{shuffle} \times \sum_{\forall c_i \in S} Size(c_i) \quad (2)$$

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

$$C_{random}(S) = f \times |T| \times r_{random} \times \sum_{\forall c_i \in S} Size(c_i) \quad (3)$$

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:

$$\Delta C = C_{random}(\Delta S) - C_{scan}(\Delta S) \quad (4)$$

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

$$\Delta C = C_{random}(\Delta S) + C_{shuffle}(\{pos\}) - C_{scan}(\Delta S) - C_{shuffle}(\Delta S) \quad (5)$$

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:

$$materialization = \begin{cases} EM, & \text{if } f > \frac{r_{scan} \times L}{r_{random} \times L + m \times r_{shuffle} \times (8 - L)} \\ LM, & \text{else} \end{cases} \quad (6)$$

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 7.3, 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 7.3. 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.

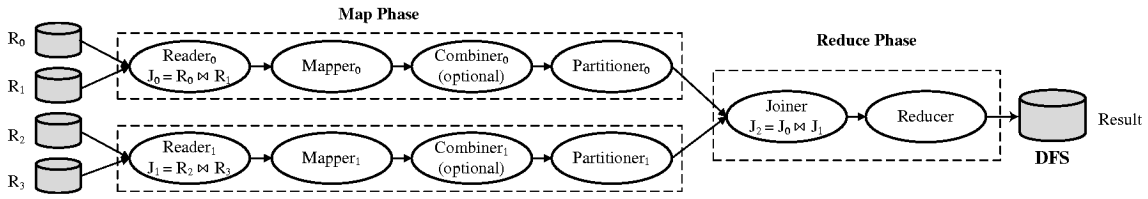


Figure 4: Concurrent Join Overview in MapReduce Framework

5. CONCURRENT JOIN

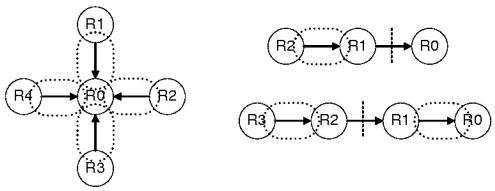
Given a query which involves joins over multiple datasets, we split the single multi-way join query into multiple sub-queries. Each sub-query involves a single join over two datasets. These sub-queries can be executed concurrently by instantiating concurrent but distinct map phases for all the sub-queries. 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 [6, 5], 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. 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.

Figure 4 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 6.

As discussed in Section 2, 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,



(A) Star Pattern (B) Chain Pattern

Figure 5: Directed Graph of Query Plan

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 5(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 $R_0 \bowtie R_1$, ..., $R_0 \bowtie R_4$.

5.1 Star Pattern in Data Cube based Analysis

The star join queries have the form $R_0 \bowtie_{R_0.a_1=R_1.a_0} R_1 \bowtie_{R_0.a_2=R_2.a_0} \ldots \bowtie_{R_0.a_n=R_n.a_0} R_n$, 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 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. 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 4). 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 $(R_0 \bowtie R_1) \bowtie (R_0 \bowtie R_2) \ldots \bowtie (R_0 \bowtie R_n)$.

5.2 Chain Pattern

Tables in the query can be linked in a chain manner as $R_0 \bowtie_{R_0.a_1=R_1.a_0} R_1 \bowtie_{R_1.a_1=R_2.a_0} \ldots R_n$. 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 reduc-

ers to produce the final results. Figure 5(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.

5.3 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.

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 ∀ node n_i ∈ 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 ∀ (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;
```

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.
- 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 4. 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.

5.4 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 6.

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

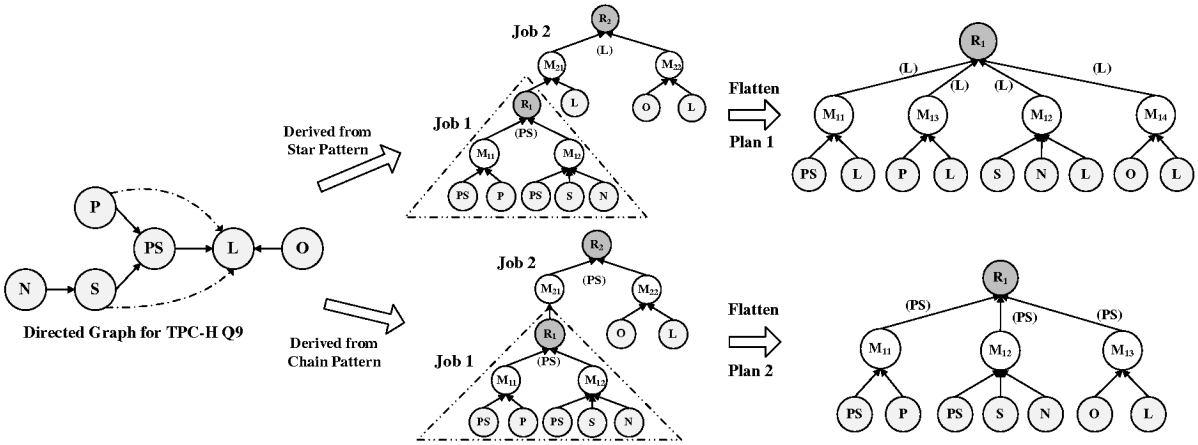


Figure 6: Execution Plan for TPC-H Q9

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.

5.5 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.

6. IMPLEMENTATION

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 facilitate 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.

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 validate the input specification and split the input datasets into logical partitions, each of which corresponds to a portion of data and is later assigned to an individual mapper. Specific reader is provided by the InputFormat to read the corresponding partition. To enable the mappers to handle column materialization and map-merge join, we implement *MaterializeInputFormat* (MIF) and *JoinInputFormat* (JIF) that extend InputFormat of Hadoop.

To split certain columns of a table into logical partitions, MIF obtains the meta information of these columns, such as the number of records, to create a list of partitions 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). Different from the traditional approach that uses the file offsets in the logical partition, sequence number is used as we need to read CFile's index to calculate the offset. This design avoids the heavy calculation in JobTracker for scalability. MIF also provides a reader to read the partition. Based on the sequence number, reader first calculates the corresponding block position of each involved CFile. After seeking to the right position, it begin to scan the values and merged them into a tuple.

MIF handles 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. Normally, the inputs of JIF are a fact table and a dimension tables. 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 reader provided by JIF not only materializes but also joins the tuples from different tables. When the reader receives

its corresponding partition of the fact table, it quickly locates the start position of the corresponding values in the dimension table via the joining key. Then, it can perform 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, we exploit the index of the dimension table to seek to the corresponding block, which further reduces the scan cost.

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 [2] and *TableInputs* in [24]. 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; *inputformat* is the particular input format to handle different data sources; *mapper*, *combiner* and *partitioner* are respectively used to filter particular tuples, combine the intermediate results, and define the partition strategy to the reducers. The return value of this interface is an integer, which binds the input dataset with the customized processing. According to this integer, Llama can use the proper input format to materialize or join the tables and instantiate the specific mapper, combiner and partitioner to process the intermediate tuples.

To join heterogeneous data sources in the reduce phase, we introduce a joiner before the reducer as the Map-Join-Reduce proposal [24]. 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:

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

In this function, *tableID* indicates two or more tables to be joined by the same joining key; *joiner* provides the join() function to join the indicated tables in the reduce phase. Before the join processing, data from these tables is sorted by the joining key. As Llama inherits and extends the joiner from the previous proposal [24], it is able to complete multiple joins of different join keys in the reduce phase.

7. 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.

7.1 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.

7.2 Comparisons Between Files

In this test, we compare the performance of different file formats implemented on the Hadoop system. They are TFile of Zebra [9], RCFile of Hive [4], HFile of HBase [3] 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. 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). Our objective here is to demonstrate that the CFile format has distinct advantages in large scale data processing.

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. 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 7 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.

Creation Overhead.

Figure 8 presents the creation overhead of different files for table

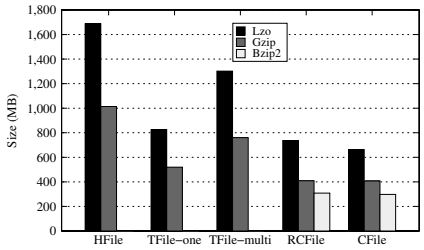


Figure 7: Storage Efficiency

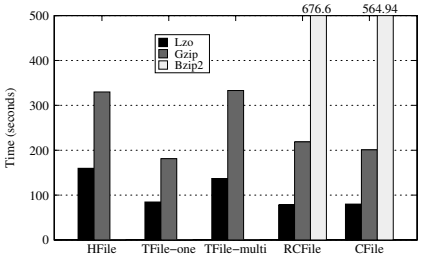


Figure 8: File Creation

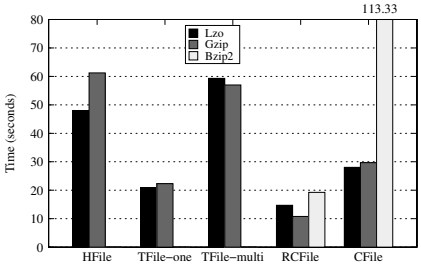


Figure 9: All-Column Scan

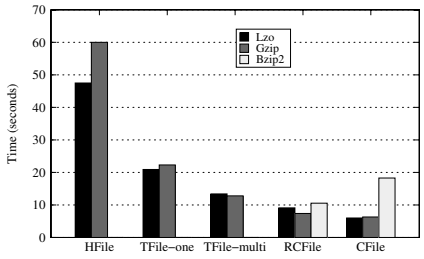


Figure 10: Two-Column Scan

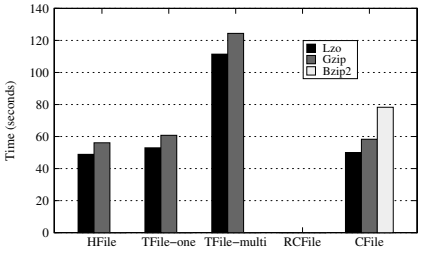


Figure 11: Two-Column Random Access

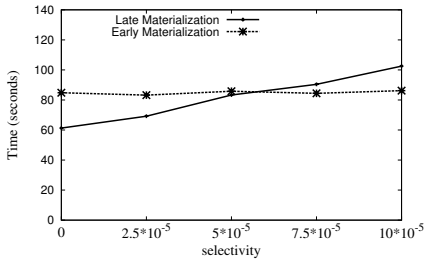


Figure 12: Column Materialization

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 9 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 10 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 it still has to read the whole block. 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 11. 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, 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.

7.3 Column Materialization

In this experiment, we first estimate the cost ratio discussed in Section 4. 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.

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 10 and Figure 11, 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 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)
from lineitem where l_orderkey < x
group by l_orderkey;
```

The performance of these two strategies are summarized in Figure 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.

7.4 Data Loading

We report the loading time of all the tables in the TPC-H benchmark in Figure 13, and briefly describe our loading procedures

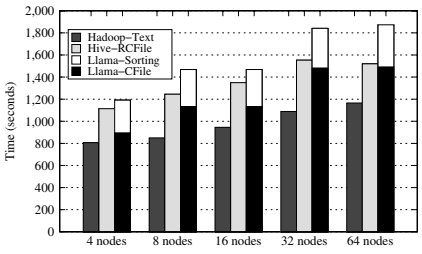


Figure 13: Load time

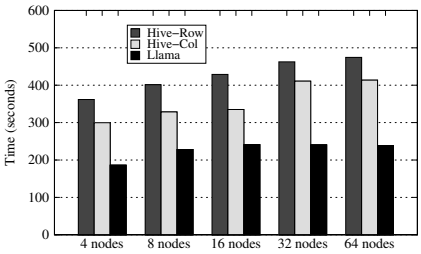


Figure 14: Aggregation Task: TPC-H Q1

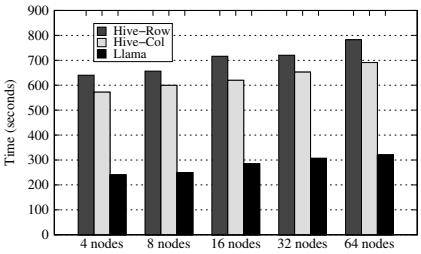


Figure 15: Join Task: TPC-H Q4

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 "copyFromLocal" 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 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.

As shown in Figure 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 savings are significantly more than the loading overhead.

7.5 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 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.

7.6 Join Task

We choose TPC-H Q4, Q3 and Q9 as our join tasks. There are one, two 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 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.

As shown in Figure 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 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

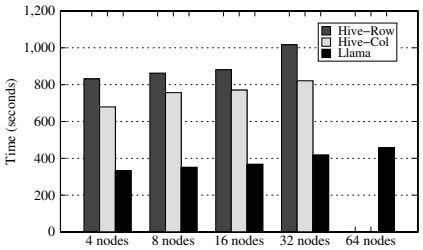


Figure 16: Join Task: TPC-H Q3

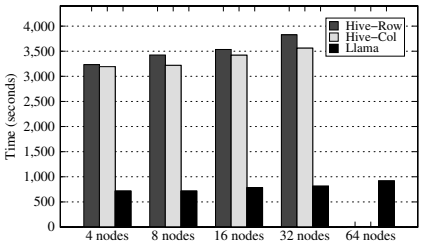


Figure 17: Join Task: TPC-H Q9

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 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 5.4. To facilitate the concurrent join, PF group of table *Partsupp* is built during the loading phase.

Figure 17 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 17, 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.

8. RELATED WORK

8.1 Join Processing in MapReduce

The MapReduce paradigm [21] 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 [2] is widely available to both commercial and academic users. Building on top of Hadoop, Pig [7] and Hive [4] provide the declarative query language interface and facilitate join operation to handle complex data analysis. Zebra [9] is a storage abstraction of Pig to provide column wise storage format for fast data projection.

To execute equi-joins in the MapReduce framework, Pig [7] and Hive [4] provide several join strategies in terms of the feature of the joining datasets [6, 5]. For example, [29] proposes a set of optimization strategies for automatic optimization of parallel dataflow programs such as Pig. On the other hand, HadoopDB [12] 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 involved tables are partitioned on the same attribute. Hadoop++ [22] 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. A comprehensive description and comparison of several equi-join implementations for the MapReduce framework appears in [16, 23]. 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, [13, 24] 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 [25], an intermediate storage system of MapReduce is proposed to augment the fault-tolerance while keeping the replication overheads low. [19] presents a modified version of the Hadoop MapReduce framework that supports online aggregation by pipelining. However, they do not essentially improve the performance of MapReduce based multi-way join processing.

8.2 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. Although vertically partitioning the table has been around for a long time [20, 15], it has only recently gained wide-spread attention to build columnar analytic databases [27, 30, 8, 26] primarily for data warehousing and online analytical processing.

Column-wise data model is also preferred in MapReduce and distributed data storage systems. HadoopDB [12] can use columnar database like C-store [30] as its underlying storage. Dremel [28] proposed a specific storage format for nested data along with the execution model for interactive queries. Bigtable [17] proposed column family to group one or more columns as a basic unit of access control. HBase [3], an open source implementation of BigTable, has been developed as the Hadoop [2] database. HFile is its underlying column-wise storage. Besides, TFile and RCFile are the other two popular file structures that have been used in Zebra [9] and Hive [4] 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.

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. This makes HFile incompact and thus ineffective in large scale data processing. 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*). The length information is necessary to state the boundary of the key and value in each record. Similar to TFile, RCFile stores the same data on each block for given columns. However, within each block, it groups all the values of a particular column together on a separate mini block, which is similar to PAX [14]. 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.

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 while 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 incompact, 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 [9] and Hive [4] could not be treated as a truly column-wise data warehouse.

9. CONCLUSION

In this paper, 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.

10. ACKNOWLEDGEMENTS

The work of Yuting Lin, Beng Chin Ooi, Sai Wu, and Chun Chen are respectively supported in part by the Ministry of Education of Singapore (Grant No. R-252-000-394-112) and the National Natural Science Foundation of China (Grant No. 61070155). We thank Amazon for the research grant of the free usage of AWS. We also thank the anonymous reviewers for their insightful comments.

11. REFERENCES

[1] Epic. `http://www.comp.nus.edu.sg/~epiC`.
[2] Hadoop. `http://hadoop.apache.org`.
[3] Hbase. `http://hbase.apache.org`.
[4] Hive. `http://hive.apache.org`.
[5] Hive/tutorial. `http://wiki.apache.org/hadoop/Hive/Tutorial#Joins`.
[6] Join framework. `http://wiki.apache.org/pig/JoinFramework`.
[7] Pig. `http://pig.apache.org`.
[8] Vertica. `http://www.vertical.com/`.
[9] Zebra. `http://wiki.apache.org/pig/zebra`.
[10] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.
[11] D. J. Abadi, D. S. Myers, D. J. Dewitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, 2007.
[12] 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.
[13] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.
[14] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.
[15] D. S. Batory. On searching transposed files. *ACM Trans. Database Syst.*, 4(4):531–544, 1979.
[16] 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.
[17] 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.
[18] 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.
[19] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.
[20] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.
[21] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.
[22] 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.
[23] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.
[24] 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.
[25] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *SoCC*, 2010.
[26] R. MacNicol and B. French. Sybase iq multiplex - designed for analytics. In *VLDB*, 2004.
[27] 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.
[28] 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.
[29] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, 2008.
[30] 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.