Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce

Songting Chen
Turn Inc.
songting.chen@turn.com

ABSTRACT

Large-scale data analysis has become increasingly important for many enterprises. Recently, a new distributed computing paradigm, called MapReduce, and its open source implementation Hadoop, has been widely adopted due to its impressive scalability and flexibility to handle structured as well as unstructured data. In this paper, we describe our data warehouse system, called Cheetah, built on top of MapReduce. Cheetah is designed specifically for our online advertising application to allow various simplifications and custom optimizations. First, we take a fresh look at the data warehouse schema design. In particular, we define a *virtual view* on top of the common star or snowflake data warehouse schema. This virtual view abstraction not only allows us to design a SQL-like but much more succinct query language, but also makes it easier to support many advanced query processing features. Next, we describe a stack of optimization techniques ranging from data compression and access method to multi-query optimization and exploiting materialized views. In fact, each node with commodity hardware in our cluster is able to process raw data at 1GBytes/s. Lastly, we show how to seamlessly integrate Cheetah into any ad-hoc MapReduce jobs. This allows MapReduce developers to fully leverage the power of both MapReduce and data warehouse technologies.

1. INTRODUCTION

Analyzing large amount of data becomes increasingly important for many enterprises' day-to-day operations. At the same time, the size of data being collected and analyzed is growing rapidly. It's not uncommon to see Petabyte data warehouse nowadays [22, 19]. This trend often makes traditional data warehouse solution prohibitively expensive. As a result, a shared-nothing MPP architecture built upon cheap, commodity hardware starts to gain a lot traction recently.

One type of such systems is MPP relational data warehouses over commodity hardware. Examples include AsterData, DATAllegro, Infobright, Greenplum, ParAccel, Vertica, etc. These systems are highly optimized for storing and querying relational data. However, to date it is hard for these systems to scale to thousands of nodes. Part of the reason is that at this scale, hardware failure becomes common as more nodes are added into the system, while most relational databases assume that hardware failure is a rare event [3]. Also, it is difficult for these systems to process non-relational data. The other type is MapReduce system [8]. Example includes the popular open source MapReduce implementation Hadoop [12]. MapReduce framework handles various types of failures very well and thus can easily scale to tens of thousands nodes. It is also more flexible to process any type of data. On the other hand, since it is a general purpose system, it lacks a declarative query interface. Users have to write code in order to access the data. Obviously, this approach requires a lot of effort and technical skills. At the same time, it may result in redundant code. The optimization of the data access layer is often neglected as well.

Interestingly, we start to see the convergence of these two types of systems recently. On one hand, relational data warehouses, such as AsterData, GreenPlum, etc, are extended with some MapReduce capabilities. On the other hand, a number of data analytics tools have been built on top of Hadoop. For example, Pig [9] translates a high-level data flow language into MapReduce jobs. HBase [14], similar to BigTable [6], provides random read and write access to a huge distributed (key, value) store. Hive [22] and HadoopDB [3] are data warehouses that support SQL-like query language.

Background and Motivation Turn (www.turn.com) is a software and services company that provides an end-to-end platform for managing data-driven digital advertising. Global advertising holding companies and the worlds best-known brands use the Turn platform to more effectively use data to target custom audiences at scale, optimize performance, and centrally manage campaigns as well as data across a range of inventory sources. At Turn, we have the following data management challenges.

- Data: There are frequent schema changes for our relational data due to fast-paced business environment. There are also large portion of data that are not relational at all. Plus, the size of data is growing rapidly.
- Simple yet Powerful Query Language: An important service we provide to our external clients is to allow them to directly access our data warehouse to perform

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.
Proceedings of the VLDB Endowment, Vol. 3, No. 2
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

their own ad-hoc analysis of their advertising data. For this, we need to provide them a simple yet powerful query language.

- Data Mining Applications: Powering the data mining applications with the capability of going through the entire dataset may greatly improve their performance. For this, we need to provide these applications a simple yet efficient data access method.

- Performance: A high performance data warehouse is a must. At the same time, it should also be able to support rapid data growth.

These challenges, in particular, the first three, motivate us to build a data warehouse that is flexible and scalable. In this paper, we describe our custom data warehouse system, called Cheetah, built on top of Hadoop. In summary, Cheetah has the following highlighted features.

- Succinct Query Language: We define virtual views on top of the common data warehouse star/snowflake schema. This design significantly simplifies the query language and its corresponding implementation, optimization and integration. As a testimony, we see clients with no prior SQL knowledge are able to quickly grasp this language.

- High Performance: Cheetah exploits a stack of Hadoop-specific optimization techniques ranging from data compression and access method to multi-query optimization and exploiting materialized views. Each node with commodity hardware in our cluster is able to process the raw data at 1GBytes per second.

- Seamless integration of MapReduce and Data Warehouse: Cheetah provides a non-SQL interface for applications to directly access the raw data. This way, MapReduce developers can take full advantage of the power of both MapReduce (*massive parallelism and scalability*) and data warehouse (*easy and efficient data access*) technologies.

The rest of the paper is organized as follows. Section 2 describes the background of this work. In Section 3, we give an overview of our data warehouse system. The schema design and query language in Cheetah is described in Section 4. The query processing and optimization are included Section 5 and 6. We show how to integrate Cheetah into user program in Section 7. The performance evaluation is presented in Section 8. We conclude in Section 9.

2. BACKGROUND

MapReduce is a programming model introduced by Dean et.al. [8] that performs parallel analysis on large data sets. The input to an analysis job is a list of key-value pairs. Each job contains two phases, namely, the *map* phase and the *reduce* phase.

The map phase executes a user-defined map function, which transforms the input key-value pairs into a list of intermediate key-value pairs.

map(k1,v1) $\rightarrow$ list(k2,v2)

The MapReduce framework then partitions these intermediate results based on key and sends them to the nodes that perform the reduce function. The user-defined reduce function is called for each distinct key and a list of values for that key to produce the final results.

reduce(k2, list(v2)) $\rightarrow$ (k3, v3)

The optional *combiner* function is quite similar to reduce function, which is to *pre-aggregate* the map output so as to reduce the amount of data to be transferred across the network. Many real world data processing jobs can be converted into MapReduce programs using these two simple primitives, such as search engine and machine learning.

While MapReduce system is fairly flexible and scalable, users have to spend a lot of effort writing MapReduce program due to lack of a declarative query interface. Also since MapReduce is just an execution model, the underlying data storage and access method are completely left to users to implement. While this certainly provides some flexibility, it also misses some optimization opportunity if the data has some structure in it. Relational databases have addressed the above issues for a long time: it has a declarative query language, i.e., SQL. The storage and data access are highly optimized as well.

This motivates us to build a hybrid system that takes advantage of both computation paradigms. HadoopDB [3] leveraged PostgreSQL as the underlying storage and query processor for Hadoop. In this work, we build a completely new data warehouse on top of Hadoop since some of our data are not relational and/or have a fast evolving schema. Hive [22] is the most similar work to ours. Instead of building a generic data warehouse system as Hive does, Cheetah is designed specifically for our own applications, thus allowing various custom features and optimizations ranging from schema design and query language to query execution and optimization. Nonetheless, many ideas and even some of the customizations described in this paper can be shared to build a general data warehouse system.

3. SCHEMA DESIGN, QUERY LANGUAGE

3.1 Virtual View over Warehouse Schema

Star or snowflake schema is common data warehouse design [16]. As shown in Figure 1, the central fact table is connected by a number of dimension tables through typically the key and foreign key relationship. Note that such relationship is often stable as a join would only make sense between key and its corresponding foreign key. Based on this observation, we define a *virtual view* on top of the star or snow-flake schema, which essentially joins the fact table with all its related dimension tables [1]. As can be seen in Figure 1, this virtual view contains attributes from fact tables, dimension tables and those pre-defined ones that can be derived from.

These virtual views are exposed to the users for query. At runtime, only the tables that the query referred to are accessed and joined, i.e., redundant join elimination. This abstraction greatly simplifies the query language (Section 3.2).

[1] In this work, we assume only key and foreign key relationship exists in the system. Also the constraints are enforced during the ETL phase.

As a result, users no longer have to understand the underlying schema design before they can issue any query. They can also avoid writing many join predicates repetitively. Note that there may be multiple fact tables and subsequently multiple virtual views. In Section 3.2, we will describe the semantics when querying more than one virtual view.

Handle Big Dimension Tables Unlike the filtering, grouping and aggregation operators, which can be easily and efficiently supported by Hadoop, the implementation of JOIN operator on top of MapReduce is not as straightforward [24, 5].

There are two ways to implement the JOIN operator, namely, either at the map phase or at the reduce phase [22, 24]. The implementation at the reduce phase is more general and applicable to all scenarios. Basically, it partitions the input tables on the join column at the map phase. The actual join is performed at the reduce phase for each partition. Obviously, when the input tables are big, this approach results in massive data redistribution.

The implementation at the map phase is applicable only when one of the join tables is small or the input tables are already partitioned on the join column. For the first case, we can load the small table into memory and perform a hash-lookup while scanning the other big table during the map phase. We can easily generalize this technique to *multi-way join* if there are m small tables to be joined with one big table. For the second case, the map phase can read the same partitions from both tables and perform the join. Unfortunately, we notice that current Hadoop implementation lacks some critical feature that there is no facility in HDFS (Hadoop File System) to force two data blocks with the same partition key to be stored on the same node. As such, one of the join tables still has to transferred over the network, which is a significant cost for big table. Hence it is difficult to do co-located join in Hadoop at this point.

Based on this observation, we choose to *denormalize* big dimension tables. Its dimension attributes are directly stored into the fact table. For example, we store URL data as additional attributes in the fact table instead of creating a huge URL dimension. Note that our big dimension tables are either *insertion-only* or are *slowly changing dimensions* [16], and the queries require to see the snapshot of dimension attributes at the time of event. Hence this denormalization approach works well for our applications.

Handle Schema Changes Frequent schema changes often pose significant challenges to relational databases, especially when the changes are on the huge fact tables. In Cheetah, we allow *schema versioned* table to efficiently support schema changes on the fact table. In particular, each fact table row contains a schema version ID. The metadata store records the columns that are available for that version. This way, adding or removing columns becomes straightforward without touching any prior data.

Handle Nested Tables In Cheetah, we also support fact tables with a *nested relational* data model. For example, in our user profile store, a single user may have different types of events, where each of them is modeled as a nested table [2]. Similarly, we define a *nested relational virtual view* on top of it by joining each nested table with all its related dimensions. We also design a query language that is much simpler than the standard nested relational query [15]. We omit the full details here since it is outside the scope of this paper.

[2]This can be viewed as another form of denormalization to avoid a costly join on different user events at query time.

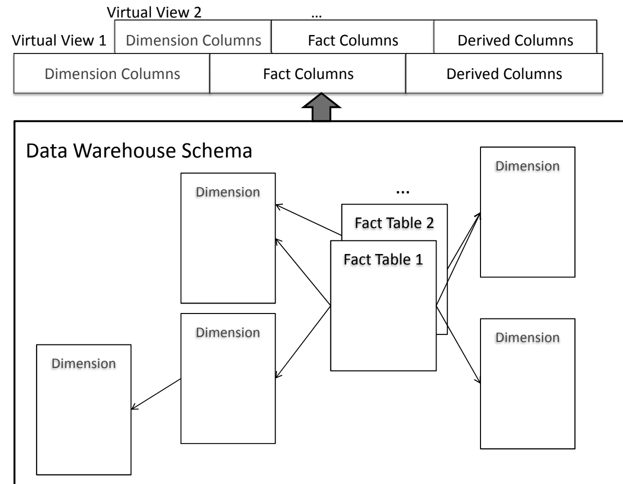

Figure 1: Virtual Views over Warehouse Schema

3.2 Query Language

Currently, Cheetah supports single block, SQL-like query. We describe the query syntax via a sample query below.

```
SELECT advertiser_name,
        sum(impression), sum(click), sum(action)
FROM Impressions, Clicks, Actions
DATES [2010_01_01, 2010_06_30]
WHERE site in ('cnn.com' ,'foxnews.com')
```

First, we provide an explicit DATES clause to make it easier for users to specify interested date ranges. Next, the FROM clause may include one or more virtual views. When there is only one virtual view, its semantics is straightforward. When there are two or more virtual views in the FROM clause, its semantics is defined as follows. Union all the rows from each individual virtual view. If a specific column referred in the query does not exist in the view, treat it as NULL value. To have meaningful query semantics, all group by columns are required to exist in each of the views in the FROM clause. Note that this union semantics still allow the query to be executed in parallel.

In our application, there are three fact tables / virtual views, namely, *Impressions*, *Clicks* and *Actions*. An impression is an event that a user is served an ad when browsing a web page. A click is an event that user clicks on the ad. An action is a conversion event at advertiser website, such as purchase, after receiving at least one impression.

In the above example, *advertiser_name* is a dimension attribute (through join on the ID column). All three virtual views, namely, *Impressions*, *Clicks* and *Actions*, contain that column. The *impression* column only exists in *Impressions* view, the *click* column only exists in *Clicks* view, while the *action* column only exists in *Actions* view. Finally, the SELECT and WHERE clauses have the same semantics as standard SQL. As a simplification, the GROUP BY clause is omitted that all the non-aggregate columns in the SELECT

clause are treated as GROUP BY columns.

As a final remark, this query language is fairly simple that users do not have to understand the underlying schema design, nor do they have to repetitively write any join predicates. For example, an equivalent query written in standard SQL would be much lengthier than the sample query above. In practice, we found that even clients with no prior SQL knowledge were able to quickly grasp this language.

Multi-Block Query Cheetah also supports CREATE TABLE AS (CTAS) statement. To answer a multi-block non-correlated query, user can first create a table and then query over the result table.

3.3 Security and Anonymity

Cheetah supports row level security [23] based on virtual views. For a given user, it may restrict the columns that are queryable by this user. It may also automatically add some additional predicates to remove those rows that should not be accessed by that user. The predicates can be defined on any of the dimension attributes, fact table attributes or derived attributes. Hence the system may join a dimension table for access control purpose even when the user query does not refer to it at all.

Cheetah also provides ways to anonymize data [21], which is done through the definition of *user-dependent* derived attributes in the virtual views. For example, for a given user, a derived attribute can be defined as simple as a case expression over a fact table attribute, which returns a default value under certain conditions. For a different user, the definition of this derived attribute can be completely different. Similarly, joins can be included automatically for the purpose of anonymizing data.

These fine-grained access control mechanisms are fairly important so that external clients are able to access and protect their own data. Our virtual view based approach makes these mechanisms fairly easy to define and fairly easy to implement.

Metadata The metadata store records the table definition, versioned schema information, user and security information etc. It is replicated to all nodes in the cluster and synchronized with the master.

4. ARCHITECTURE

In this section, we provide an overview of our data warehouse system. In particular, we design the system to accomplish the following two goals.

- Simple yet efficient: The core query engine is able to efficiently process the common data warehouse queries, i.e., star joins and aggregations (Section 3.2).
- Open: It should also provide a simple, non-SQL interface to allow ad-hoc MapReduce programs to directly access the raw data.

Figure 2 depicts the architecture of Cheetah. As can be seen, user can issue query through either Web UI, or command line interface (CLI), or Java code via JDBC. The query is sent to the node that runs the *Query Driver*. The main task for Query Driver is to translate the query to a MapReduce job, which includes the map phase plan and the reduce phase plan.

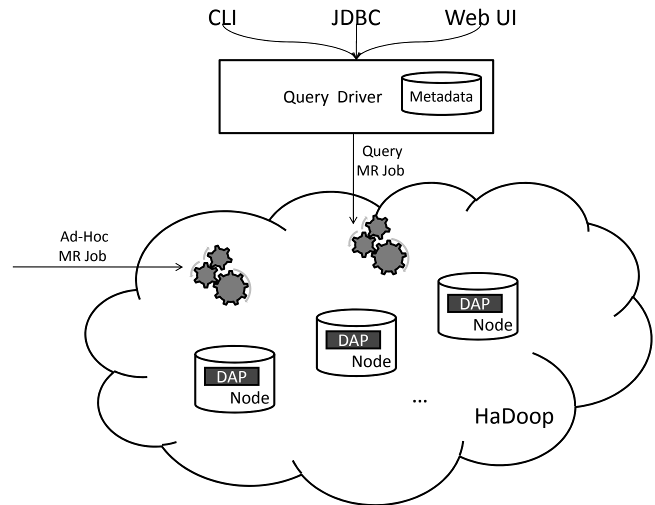

Figure 2: Cheetah Architecture

Several possible optimizations are considered during this query translation phase. For example, it needs to config the query MapReduce job by setting a proper number of reducers. It may also decide to batch process multiple user queries if they are sharable, or to use a pre-defined materialized view if it matches the query. Overall, however, these are quite lightweight optimizations.

As for query execution, each node in the Hadoop cluster provides a *data access primitive (DAP)* interface, which essentially is a *scanner* over *virtual views*. The query MapReduce job utilizes this scanner, which performs SPJ portion of the query. The ad-hoc MapReduce job can issue a similar API call for fine-grained data access as well.

5. DATA STORAGE AND COMPRESSION

5.1 Storage Format

We distinguish between several methods for storing tabular data, namely, *text* (in CSV format), *serialized java object*, *row-based binary array* and *columnar binary array*.

Text is the simplest storage format and is commonly used in web access logs. A java class can implement the serialization method to write its members to a binary output stream, and implement the deserialization method to read from a binary input stream to reconstruct the java object. Row-based binary array is commonly used in row-oriented database systems, where each row is deserialized into a binary array, which is then being written and read as a whole. In the case of read, only interested columns are extracted from the binary array for further processing.

The above three methods conceptually correspond to a row-oriented store. Columnar binary array conceptually is a hybrid of row-column store [4]. That is, n rows are stored in one *cell* [3]. Within one cell, the values of the same column are stored together as a *column set*. As we will show in Section 9, storage format has a huge impact on both compression ratio and query performance. In Cheetah, we store data in columnar format whenever possible.

[3] $n <= N$. We let $N = 200,000$ in our system to balance between better compression ratio and memory usage.

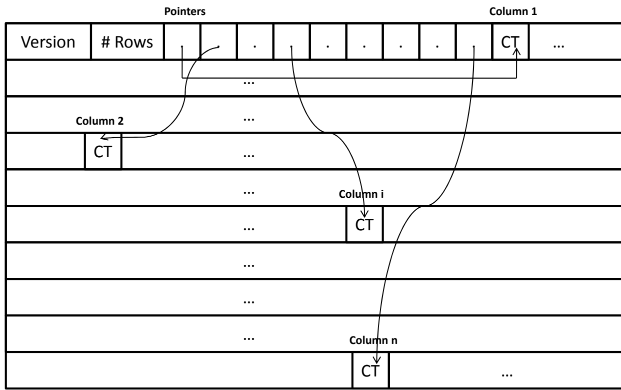

Figure 3: Columnar Storage Format

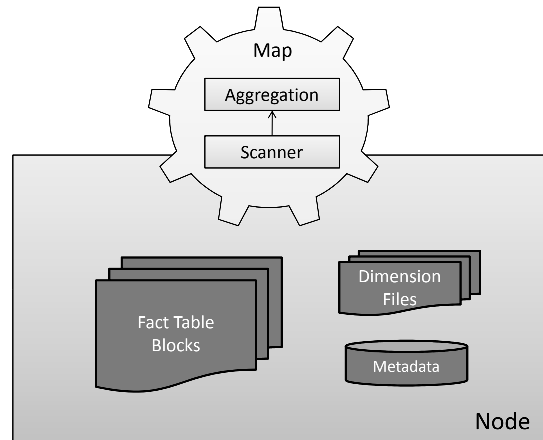

Figure 4: Query Processing: Map Phase

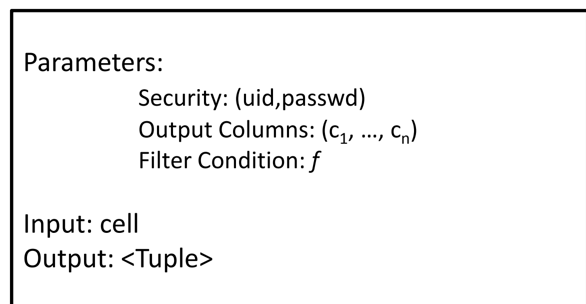

Figure 5: Scanner Interface

5.2 Columnar Compression

Column-oriented DB has shown great potential especially for data compression. The intuition is that the values of the same column are more likely to be similar to each other. Hence the increasing similarity of adjacent data offers better compression opportunity [1].

Figure 3 depicts the storage format of a single cell in Cheetah. The header includes the schema version and the number of rows in this cell. The data portion includes all the column sets and the pointers to the beginning of each column set. Each column set starts with a compression flag (CT), which can be one of the following: *dictionary encoding*, *run length encoding*, *default value encoding* or *no compression* [1, 10, 20].

The choice of compression type for each column set is dynamically determined based on the data in each cell. During ETL phase, the statistics of each column set is maintained and the best compression method is chosen. We also provide heuristics to the ETL process on which columns to sort in order to take full advantage of the run length encoding method, and on how to put similar data into the same cell in order to maximize the compression ratio.

After one cell is created, it is further compressed using GZIP. Put together, we achieve 8X compression ratio compared to when the data is stored in the row-oriented binary array format.

In the case of decompression, the entire cell is first loaded into memory. Only those column sets that are referred in the query are decoded through an iterator interface.

6. QUERY PLAN AND EXECUTION

In this section, we show how to translate the query to a Hadoop MapReduce job. In general, to submit a Hadoop MapReduce job, first we need to specify the input files, which are stored on Hadoop distributed file system (HDFS). We also need to supply query-specific map function and reduce function.

6.1 Input Files

The input files to the MapReduce job are always fact tables. Hadoop framework will schedule map tasks to scan those fact table files in parallel. There is no need to specify any dimension tables as input. They will be picked up automatically during the map phase.

The fact tables are partitioned by date. The DATES clause helps to find related partitions (Section 3.2). Actually the fact tables are further partitioned by a dimension key attribute, referred as D_{ID}. A predicate on D_{ID} in the query will certainly help choose the right partition, which may further reduce the amount of data to be processed. In Cheetah, we also exploit functional dependency to maximize the possibility of using this partition property. For example, assume that we have a functional dependency $D'_A \rightarrow D_{ID}$ and a query predicate $D'_A = a$. By querying related dimensions, we can add an additional predicate $D_{ID} = id$ to the query. Since these dimensions are small, this additional cost is insignificant compared to the potential big performance gain. As a custom data warehouse, we can easily incorporate these domain-specific optimizations into the query engine.

6.2 Map Phase Plan

As can be seen in Figure 4, each node in the cluster stores some portion of the fact table data blocks and (small) dimension files.

The map phase of the query contains two operators, namely, *scanner* and *aggregation*. Externally, the scanner operator has an interface which resembles a SELECT followed by PROJECT operator over the virtual view (Figure 5). That is, a filter condition on the input rows and a list of columns that are interested as output. Here a column can be an expression, such a case statement, as well. The output tuples have an iterator interface such that they can be retrieved by a *getNext* function call. For proper access control (Section 3.3), a user ID and password is also required.

Internally, the scanner operator translates the request to

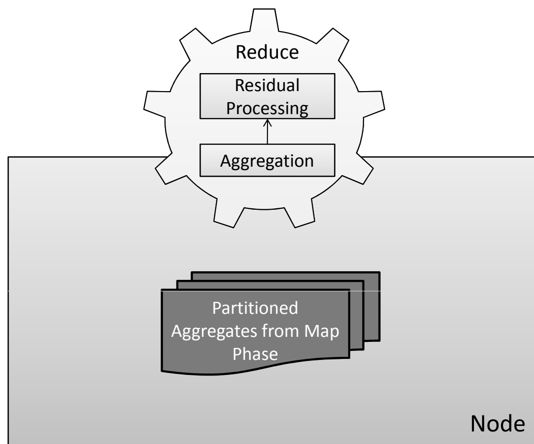

Figure 6: Query Processing: Reduce Phase

an equivalent SPJ query to pick up the attributes on the dimension tables. Note that only those dimension tables referred in the query need to be accessed and joined. We exploit the multi-way join algorithm (Section 3.1) here since the dimensions to be joined are small (big dimension tables are denormalized and thus joins are not required). As an optimization, these dimensions are loaded into in-memory hash tables only once if different map tasks share the same JVM.

Besides selection pushdown, the scanner also explores an optimization that it first retrieves the columns required in the filter condition. If the condition is not satisfied, then it can skip fetching the rest columns. Obviously, this optimization is very useful when the filter condition is selective. Conceptually, it is similar to the late tuple construction strategy in column DB [2].

The aggregation operator is to perform local aggregation of the current block when the aggregate function is algebraic [11]. By default, we use a hash-based implementation of group by operator.

6.3 Reduce Phase Plan

The reduce phase of the query is straightforward as shown in Figure 6. It first performs global aggregation over the results from map phase. Then it evaluates any residual expressions over the aggregate values and/or the HAVING clause.

Lastly, if the ORDER BY columns are group by columns, then they are already sorted by Hadoop framework during the reduce phase. We can leverage that property. If the ORDER BY columns are the aggregation columns, then we sort the results within each reduce task and merge the final results after MapReduce job completes. If in addition there is a LIMIT by n clause, each reduce task only needs to output the top n rows for the final merge.

7. QUERY OPTIMIZATION

In this section, we present some unique optimization opportunities that are specific to our virtual view design and MapReduce framework.

7.1 MapReduce Job Configuration

For a given Hadoop MapReduce job and input files for that job, the number of map tasks are determined by Hadoop framework based on the number of input files and the number of blocks per file. The number of reduce tasks however needs to be supplied by the job itself and has a big impact on performance. When the query output is small, map phase dominates the total cost. Having a large number of reducers is a waste of resources and may slow down the query as well. However, when the query output is large, it is mandatory to have sufficient number of reducers to partition the work.

We use some simple heuristics to automatically config the number of reducers for the query job. First, the number of reducers is proportional to the number of group by columns in the query. Second, if the group by column includes some column with very large cardinality, e.g., URL, we increase the number of reducers as well. These simple heuristics work well for majority of our queries.

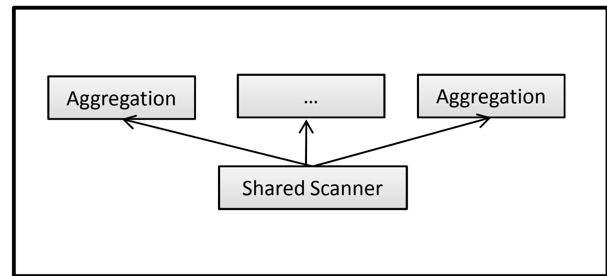

Figure 7: Multi-Query Processing : Map Phase

7.2 Multi-Query Optimization

One of our typical application scenarios is that an analyst picks a particular data range such as month-to-date or quarter-to-date, and issues a number of queries to analyze the data from different perspective. Another scenario is that we have a lot nightly scheduled queries. One important requirement is to have them processed within a specific time window.

In Cheetah, we allow users to simultaneously submit multiple queries and execute them in a single batch, as long as these queries have the same FROM and DATES clauses. For example, the query below can be executed together with the query in Section 3.2.

```
SELECT publisher_name,
       sum(impression), sum(click), sum(action)
FROM Impressions, Clicks, Actions
DATES [2010_01_01, 2010_06_30]
```

As shown in Figure 7, to execute this combined query, the map phase now has a *shared* scanner, which shares the scan of the fact tables and joins to the dimension tables. Note that we take a *selection pushup* approach as in [7] in order to share the joins among multiple queries.

The scanner will attach a query ID to each output row, indicating which query this row qualifies. This ID will also be used as an additional group by column. The output from different aggregation operators will be merged into a single output stream. Hence it is important to have the query ID otherwise reduce phase is not able to distinguish between the rows for different queries.

The reduce phase (Figure 8) will first split the input rows based on their query IDs and then send them to the corresponding query operators.

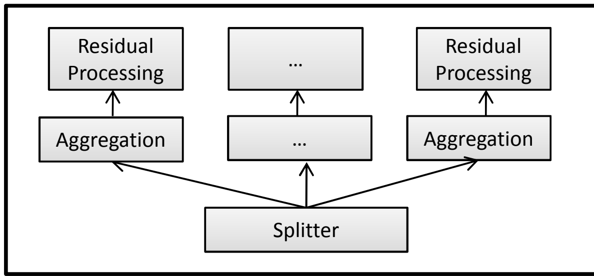

Figure 8: Multi-Query Processing : Reduce Phase

The above scenario assumes user supply a bunch of queries that can be shared. In fact, the system is also able to share the queries across different query workloads. The Query Driver in Figure 2 determines the queries in the process queue that can be shared before submitting them to the Hadoop system. Note that once a query is started, it is difficult to attach another query to it under current Hadoop implementation. Nonetheless, this technology is still useful when the system is under a high load that there are many concurrent queries.

As a final remark, our virtual view abstraction makes this sharing transparent to users. That is, it becomes fairly easy for them to decide what queries can be shared, i.e., same FROM and DATES clauses, while how to create a shared query plan is done automatically by the system. Otherwise users have to modify the query to first create a shared temporary table [22] or add SPLIT/MULTIPLEX operators [9]. In other words, users have to create a shared plan themselves. This is non-trivial work if there are a large number of queries to be batched.

7.3 Exploiting Materialized Views

Exploiting materialized views is a common technique for data warehousing [13]. In this section, we show how to apply this technique based on our virtual view abstraction.

7.3.1 Definition of Materialized Views

First, each materialized view only includes the columns in the face table, i.e., excludes those on the dimension tables. Second, it is partitioned by date as well. That is, we create one materialized view every day. Below is a sample materialized view defined on *Impressions* fact table.

```
CREATE MATERIALIZED VIEW pub AS
SELECT publisher_id,
          sum(impression) impression
FROM Impressions
```

As can be seen, first, both columns referred in the query reside on the fact table, *Impressions*. Second, for simplicity, we do not include any filtering condition in the WHERE clause to simplify the query matching. It also maximizes the matching possibility.

Then we create a virtual view, (for easy description) still referred as *pub*, on top of this materialized view, which joins all related dimension tables. In this case, only *publisher* dimension is related. The resulting virtual view has two types of columns, namely, the group by columns and the aggregate columns. From now on, we will refer this resulting virtual view as the materialized view for easy description.

7.3.2 View Matching and Query Rewriting

To make use of materialized view, two problems need to be addressed, namely, view matching and query rewriting [13]. The following conditions must satisfy for a materialized view to match a query [4].

- The query must refer the virtual view that corresponds to the same fact table that the materialized view is defined upon.
- The non-aggregate columns referred in the SELECT and WHERE clauses in the query must be a subset of the materialized view's group by columns.
- The aggregate columns must be computable from the materialized view's aggregate columns.

For example, the materialized view, *pub*, defined in Section 7.3.1 matches the query in Section 7.2. To rewrite this query, we can simply replace the virtual view in the query with the matching materialized view. Below is the resulting rewritten query.

```
SELECT publisher_name,
          sum(impression), sum(click), sum(action)
FROM pub, Clicks, Actions
DATES [2010_01_01, 2010_06_30]
```

At runtime, the partitions of materialized view *pub* are scanned instead of the *Impression* table. They are joined with publisher dimension to get the name column.

7.4 Low-Latency Query Optimization

The current Hadoop implementation has some non-trivial overhead itself, which includes for example job start time, JVM start time, etc [3]. For small queries, this becomes a significant extra overhead.

To address this issue, during the query translation phase, if the Query Driver detects that the size of the input file is small, e.g., hitting a materialized view, it may choose to directly read the file from HDFS and then process the query locally. This avoids invoking a MapReduce job at all. We pick a file size threshold so that the query can typically finish within 10 seconds. This way, we can improve the responsiveness of the low-latency queries as well as reduce the cluster workload.

8. INTEGRATION

Cheetah provides various ways to connect to user program (Figure 2). First, it has a JDBC interface such that user program can submit query and iterate through the output results. Second, if the query results are too big for a single program to consume, user can write a MapReduce job to analyze the query output files which are stored on HDFS. Lastly, Cheetah provides a non-SQL interface that can be easily integrated into any ad-hoc MapReduce jobs to access the data at the finest granularity.

To achieve this, the ad-hoc MapReduce program needs to first specify the input files, which can be one or more fact table partitions. Then in the Map program, it needs to include a *scanner* in Figure 5 to access individual raw record

[4]After security predicates (Section 3.3) are added to the query.

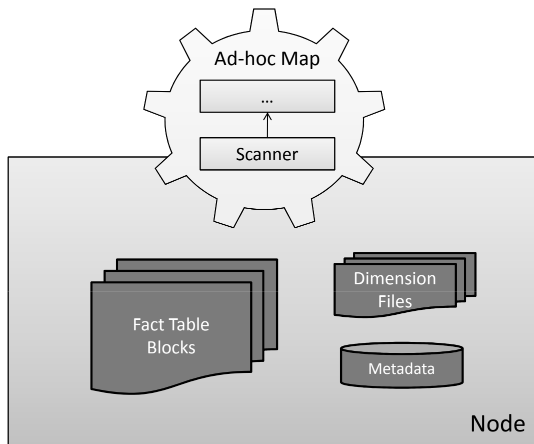

Figure 9: Map Phase for Ad-hoc Job

(Figure 9). After that, user has complete freedom to decide what to do with the data.

There are several advantages to have this low-level, non-SQL interface open to external applications. First, it provides ad-hoc MapReduce programs *efficient* and more importantly *local* data access, which means that there is no data moving across different systems or even different nodes. Second, the virtual view-based scanner hides most of the schema complexity from MapReduce developers. That is, they need not to be aware of how to retrieve the columns, what and where the dimension tables are, how to join them, etc. They can now solely focus on the application itself. Third, the data is well compressed and the access method is fairly optimized inside the scanner operator. In summary, ad-hoc MapReduce programs can now automatically take full advantage of both MapReduce (massive parallelism and scalability) and data warehouse (easy and efficient data access) technologies.

9. PERFORMANCE EVALUATION

9.1 Implementation

The Cheetah system has been built completely from scratch in Java. One important experience we learned is that the query engine must have low CPU overhead. This includes for example choosing the right data format (Section 9.2) and efficient implementation of various components on the data processing path. For example, we implemented an efficient hashing method to support multi-column group by.

The system has been deployed to production on Dec. 2008. Today, it has become the primary data platform that supports all data warehouse analytics and machine learning applications at Turn. In fact, a number of machine learning algorithms have been developed or strengthened by having the ability to go through the entire dataset, while external clients were able to discover many interesting custom insights using our simple yet powerful query language.

In this section, we present some experimental results of the system. All the experiments are performed on a cluster with 10 nodes. Each node has two quad core, 8GBytes memory and 4x1TBytes 7200RPM hard disks. We also set big readahead buffer at OS level in order to support multiple concurrent sequential reads, which is typical for Hadoop workload.

We use Cloudera's Hadoop distribution version 0.20.1. The size of the data blocks for fact tables is 512MBytes. The main reason for having such a big block size is that since the query map task is quite efficient, it finishes quickly for smaller data blocks. In this case, a small data block would make it harder for job scheduler to keep pace with the map tasks. A big data block thus allows the system to exploit all available system resources.

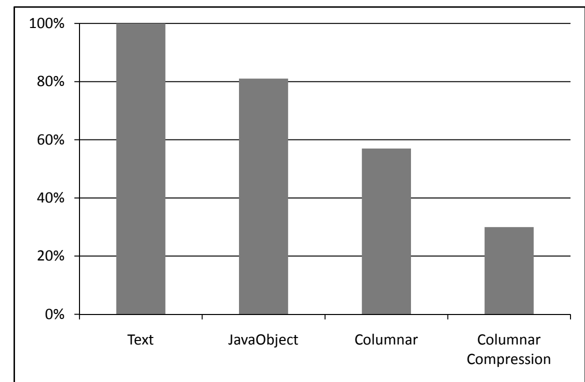

Figure 10: GZIP on Different Storage Format

9.2 Storage Format

The first set of experiments study the difference between different storage formats (Section 5.1) in terms of compression ratio as well as query performance.

First, we store one fact table partition into four files, which correspond to four different data formats, namely, *Text* (in CSV format), *Java object* (equivalent to row-based binary array), *column-based binary array*, and finally *column-based binary array with compressions* described in Section 5.2. Each file is further compressed by Hadoop's GZIP library at block level.

Figure 10 depicts the comparison of file size after GZIP compression. We use the compressed Text file as baseline for comparison. As can be seen, binary format consumes 20% less storage compared to text format after GZIP compression. There is another 25% storage reduction by simply storing the rows in columnar format. This is due to the fact that the values of the same column are more likely to be similar to each other, which offers better compression opportunity if they are stored adjacent to each other. Lastly, our compression method in Section 5.2 provides an additional significant lift in the compression ratio. Overall, it is more than 3X better than Text based compression.

Next, we show that storage format has a significant impact on query performance as well. We run a simple aggregate query over three different data formats, namely, Text, Java Object and Columnar (with compression). We use iostat to monitor the CPU utilization and IO throughput at each node.

Figure 11 and 12 record the average CPU utilization and IO throughput of a single node after the system resources are fully utilized, i.e., when each node runs the maximum number of map tasks. This is a good indicator of system's query processing capability.

As can be seen, both Text and Java Object formats incur a very high CPU overhead (above 90%). The disk IO

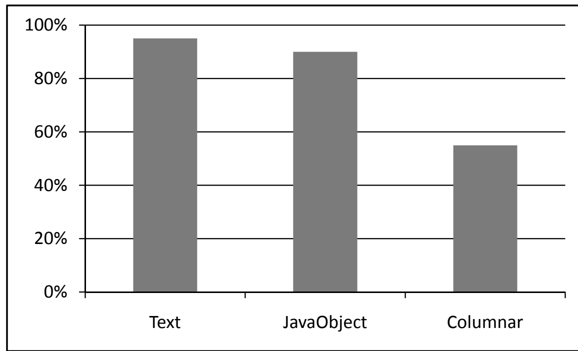

Figure 11: CPU Utilization

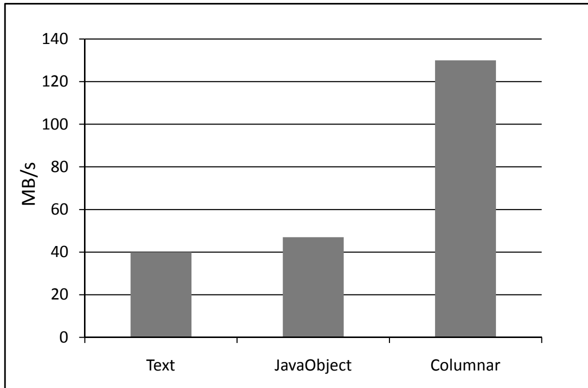

Figure 12: Disk IO Throughput

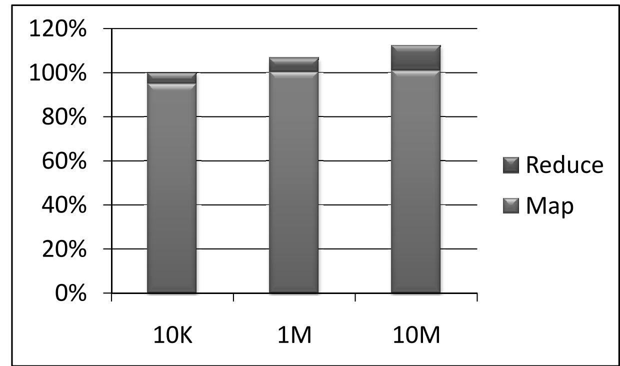

Figure 13: Small .vs. Big Queries

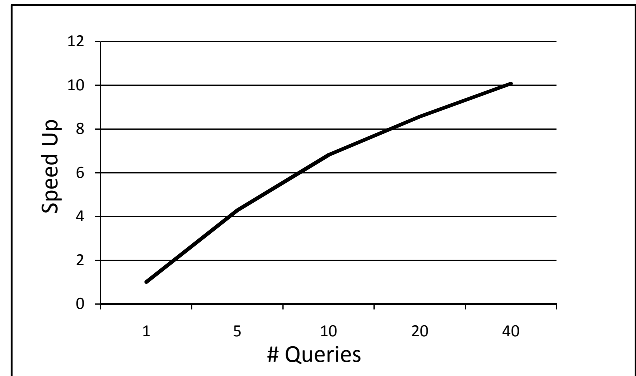

Figure 14: Multi-Query Optimization

throughput is merely 40MBytes per second, since the query essentially is CPU bound. In comparison, the columnar format has much less than CPU overhead (less than 60%) and much higher disk IO throughput (130MBytes per second, or about 1GBytes per second over uncompressed data). The same query now becomes IO bound (we observe an average 20% IO wait time from iostat).

The main reason for such a big difference is that the *deserialization* for both Text and Java Object format is fairly CPU-intensive. It extracts every single column even when the query does not ask for it. In comparison, the columnar format reads the entire binary as a whole and then only enumerates those columns that are interested.

In summary, columnar storage format is more preferred method for storing and accessing data than the common Text or Java Object format used on Hadoop.

9.3 Small .vs. Big Queries

In this section, we study the impact of query complexity on query performance. We create a test query with two joins [5], one predicate, 3 group by columns including URL, and 7 aggregate functions. By varying the selectivity of the predicate, we are able to control the number of output groups.

In Hadoop, the reduce phase of a MapReduce job contains three phases, namely, shuffle (copy partial aggregates from map tasks), sort and reduce. The shuffle task overlaps well with the map phase, while sort and reduce start only when the map phase completes. For this experiment, we measure not only the total query evaluation time, but also the time spent on processing the map tasks and the time spent on sort and reduce portion of the reduce tasks.

Figure 13 depicts the results, where x-axis is the number of output groups and y-axis is the relative performance between these three queries. As can be seen, the total evaluation time increases merely 10% for a query that returns 10M groups compared to a query that returns just 10K groups. The difference can be explained by the extra cost spend on the reduce phase. Overall, however, the increase is insignificant since we have sufficient reduce tasks for big queries. Note that if only one reduce task is specified for this MapReduce job, the performance will get much worse.

9.4 Multi-Query Optimization

In this section, we present the experimental results for multi-query optimization (Section 7.2).

We randomly pick 40 queries from our query workload. The number of output rows for these queries ranges from few hundred to 1M. We compare the time for executing these queries in a single batch to the time for executing them separately. As can be seen in Figure 14, at the batch size of 10 queries, the speed up is 7 times. For 40 queries, the speed up is 10 times. However, since the workload becomes quite CPU intensive, adding more queries to the batch no longer offers much gain.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we present our Cheetah data warehouse

[5] In fact, adding more joins has little impact on performance since the tables to be joined are small.

system built on top of the MapReduce technology. The virtual view abstraction plays a central role in designing the Cheetah system. Not only this abstraction fits nicely to the common data warehouse schema design, but also it allows some quite interesting optimization opportunities such as multi-query optimization. Lastly, we show how ad-hoc MapReduce programs can access the raw data by exploiting this virtual view interface.

There are a number of interesting future works. First, the current IO throughput 130MBytes (Figure 12) has not reached the maximum possible speed of hard disks. We suspect this is due to concurrent sequential reads. A native Hadoop implementation of readahead buffer may be helpful here. We also observe that sometimes one of the four disks does not have any reads (we use JBOD setup which is commonly used for Hadoop), while the other three disks are quite busy. Hence, the job scheduler ideally should also be aware of the load on the hard disk in order to have more balanced IO workload. Second, our current multi-query optimization only exploits shared data scan and shared joins. It is interesting to further explore predicate sharing [18] and aggregation sharing [17]. Some other interesting ideas include caching of previous query results for answering similar queries later. This would need a further predicate matching beyond the matching described in Section 7.3.

Acknowledgement We would like to thank Xuhui Shao for many valuable comments to the paper. We also thank many talented engineers at Turn for the feedback to the Cheetah system.

11. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of SIGMOD*, pages 671–682, 2006.

[2] D. J. Abadi, S. Madden, and N. Hachem. Column-Stores .vs. Row-Stores How Different Are They Really. In *Proceedings of SIGMOD*, pages 967–980, 2008.

[3] A. Abouzeid, K. Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectual Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *Proceedings of VLDB*, pages 922–933, 2009.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of VLDB*, pages 169–180, 2001.

[5] S. Blanas, J. Patel, V. Ercegovac, J. Rao, E. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proceedings of SIGMOD*, pages 975–986, 2010.

[6] F. Chang, J. Dean, S. Ghemawat, and et. al. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of OSDI*, 2006.

[7] J. Chen, D. J. DeWitt, and J. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proceedings of ICDE*, page 345, 2002.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI*, page 10, 2004.

[9] A. Gates, O. Natkovich, and S. Chopra et al. Building a HighLevel Dataflow System on top of MapReduce: The Pig Experience. In *Proceedings of VLDB*, pages 1414–1425, 2009.

[10] G. Graefe and L. Shapiro. Data Compression and Database Performance. *ACM/IEEE-CS Symp. On Applied Computing*, pages 22–27, 1991.

[11] J. Gray, S. Chaudhuri, and A. Bosworth et.al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proceedings of ICDE*, pages 29–53, 1997.

[12] Hadoop. *Open Source Implementation of MapReduce. http://hadoop.apache.org/.*

[13] A. Y. Halevy. Answering Queries using Views: A Survey. pages 270–294, 2001.

[14] HBase. *Open Source Implementation of HBase. http://hadoop.apache.org/hbase/.*

[15] Oracle. http://www.orafaq.com/wiki/NESTED_TABLE.

[16] Ralph Kimball. *The Data Warehousing Toolkit*, 1996.

[17] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-Fly Sharing for Streamed Aggregation. In *Proceedings of SIGMOD*, pages 623–634, 2006.

[18] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of SIGMOD*, pages 49–60, 2008.

[19] C. Monash. The 1-Petabyte Barrier is Crumbling. *www.networkworld.com/community/node/31439.*

[20] V. Raman and G. Swart. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *Proceedings of VLDB*, pages 859–869, 2006.

[21] L. Sweeney. K-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, pages 557–570, 2002.

[22] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, and P. Wyckoff. Hive - A Warehouse Solution Over a Map-Reduce Framework. In *Proceedings of VLDB*, pages 1626–1629, 2009.

[23] Q. Wang, T. Yu, and N. Li et.al. On The Correctness Criteria of Fine-Grained Access Control in Relational Databases. In *Proceedings of VLDB*, pages 555–566, 2007.

[24] H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *Proceedings of SIGMOD*, pages 1029–1040, 2007.