Tenzing
A SQL Implementation On The MapReduce Framework

Biswapesh Chattopadhyay
biswapesh@

Liang Lin
lianglin@

Weiran Liu
wrliu@

Sagar Mittal
sagarmittal@

Prathyusha Aragonda
prathyusha@

Vera Lychagina
vlychagina@

Younghee Kwon
youngheek@

Michael Wong
mcwong@

Google
*@google.com

ABSTRACT

Tenzing is a query engine built on top of MapReduce [9] for ad hoc analysis of Google data. Tenzing supports a mostly complete SQL implementation (with several extensions) combined with several key characteristics such as heterogeneity, high performance, scalability, reliability, metadata awareness, low latency, support for columnar storage and structured data, and easy extensibility. Tenzing is currently used internally at Google by 1000+ employees and serves 10000+ queries per day over 1.5 petabytes of compressed data. In this paper, we describe the architecture and implementation of Tenzing, and present benchmarks of typical analytical queries.

1. INTRODUCTION

The MapReduce [9] framework has gained wide popularity both inside and outside Google. Inside Google, MapReduce has quickly become the framework of choice for writing large scalable distributed data processing jobs. Outside Google, projects such as Apache Hadoop have been gaining popularity rapidly. However, the framework is inaccessible to casual business users due to the need to understand distributed data processing and C++ or Java.

Attempts have been made to create simpler interfaces on top of MapReduce, both inside Google (Sawzall [19], FlumeJava [6]) and outside (PIG [18], HIVE [21], HadoopDB [1]). To the best of our knowledge, such implementations suffer from latency on the order of minutes, low efficiency, or poor SQL compatibility. Part of this inefficiency is a result of MapReduce's perceived inability to exploit familiar database optimization and execution techniques [10, 12, 20]. At the same time, distributed DBMS vendors have integrated the MapReduce execution model in their engines [13] to provide an alternative to SQL for increasingly sophisticated analytics. Such vendors include AsterData, GreenPlum, Paraccel, and Vertica.

In this paper, we describe Tenzing, a SQL query execution engine built on top of MapReduce. We have been able to build a system with latency as low as ten seconds, high efficiency, and a comprehensive SQL92 implementation with some SQL99 extensions. Tenzing also supports efficiently querying data in row stores, column stores, Bigtable [7], GFS [14], text and protocol buffers. Users have access to the underlying platform through SQL extensions such as user defined table valued functions and native support for nested relational data.

We take advantage of indexes and other traditional optimization techniques—along with a few new ones—to achieve performance comparable to commercial parallel databases. Thanks to MapReduce, Tenzing scales to thousands of cores and petabytes of data on cheap, unreliable hardware. We worked closely with the MapReduce team to implement and take advantage of MapReduce optimizations. Our enhancements to the MapReduce framework are described in more detail in section 5.1.

Tenzing has been widely adopted inside Google, especially by the non-engineering community (Sales, Finance, Marketing), with more than a thousand users and ten thousand analytic queries per day. The Tenzing service currently runs on two data centers with two thousand cores each and serves queries on over 1.5 petabytes of compressed data in several different data sources and formats. Multiple Tenzing tables have over a trillion rows each.

2. HISTORY AND MOTIVATION

At the end of 2008, the data warehouse for Google Ads data was implemented on top of a proprietary third-party database appliance henceforth referred to as DBMS-X. While it was working reasonably well, we faced the following issues:

1. Increased cost of scalability: our need was to scale to petabytes of data, but the cost of doing so on DBMS-X was deemed unacceptably high.

Proceedings of the VLDB Endowment, Vol. 4, No. 12

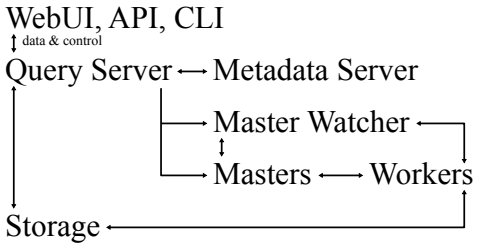

Figure 1: Tenzing Architecture.

2. Rapidly increasing loading times: importing new data took hours each day and adding new data sources took proportionally longer. Further, import jobs competed with user queries for resources, leading to poor query performance during the import process.

3. Analyst creativity was being stifled by the limitations of SQL and lack of access to multiple sources of data. An increasing number of analysts were being forced to write custom code for more complex analysis, often directly against the source (such as Sawzall against logs).

We decided to do a major re-design of the existing platform by moving all our analysis to use Google infrastructure, and specifically to the MapReduce platform. The new platform had to:

1. Scale to thousands of cores, hundreds of users and petabytes of data.

2. Run on unreliable off-the-shelf hardware, while continuing to be highly reliable.

3. Match or exceed the performance of the existing platform.

4. Have the ability to run directly off the data stored on Google systems, to minimize expensive ETL processes.

5. Provide all the required SQL features to the analysts to minimize the learning curve, while also supporting more advanced functionality such as complex user-defined functions, prediction and mining.

We have been largely successful in this effort with Tenzing. With about 18 months of development, we were able to successfully migrate all users off DBMS-X to the new platform and provide them with significantly more data, more powerful analysis capabilities, similar performance for most common scenarios, and far better scalability.

3. IMPLEMENTATION OVERVIEW

The system has four major components: the worker pool, query server, client interfaces, and metadata server. Figure 1 illustrates the overall Tenzing architecture.

The **distributed worker pool** is the execution system which takes a query execution plan and executes the MapReduces. In order to reduce query latency, we do not spawn any new processes,[1] but instead keep the processes running constantly. This allows us to significantly decrease the end-to-end latency of queries. The pool consists of master and worker nodes, plus an overall gatekeeper called the master watcher. The workers manipulate the data for all the tables defined in the metadata layer. Since Tenzing is a heterogeneous system, the backend storage can be a mix of various data stores, such as ColumnIO [17], Bigtable [7], GFS [14] files, MySQL databases, etc.

The **query server** serves as the gateway between the client and the pool. The query server parses the query, applies optimizations and sends the plan to the master for execution. The Tenzing optimizer applies some basic rule and cost based optimizations to create an optimal execution plan.

Tenzing has several **client interfaces**, including a command line client (CLI) and a Web UI. The CLI provides more power such as complex scripting and is used mostly by power users. The Web UI, with easier-to-use features such as query & table browsers and syntax highlighting, is geared towards novice and intermediate users. There is also an API to directly execute queries on the pool, and a standalone binary which does not need any server side components, but rather launches its own MapReduce jobs.

The **metadata server** provides an API to store and fetch metadata such as table names and schemas, and pointers to the underlying data. The metadata server is also responsible for storing ACLs (Access Control Lists) and other security related information about the tables. The server uses Bigtable as the persistent backing store.

3.1 Life Of A Query

A typical Tenzing query goes through the following steps:

1. A user (or another process) submits the query to the query server through the Web UI, CLI or API.

2. The query server parses the query into an intermediate parse tree.

3. The query server fetches the required metadata from the metadata server to create a more complete intermediate format.

4. The optimizer goes through the intermediate format and applies various optimizations.

5. The optimized execution plan consists of one or more MapReduces. For each MapReduce, the query server finds an available master using the master watcher and submits the query to it. At this stage, the execution has been physically partitioned into multiple units of work(i.e. shards).

6. Idle workers poll the masters for available work. Reduce workers write their results to an intermediate storage.

7. The query server monitors the intermediate area for results being created and gathers them as they arrive. The results are then streamed to the upstream client.

4. SQL FEATURES

Tenzing is a largely feature complete SQL engine, and supports all major SQL92 constructs, with some limitations. Tenzing also adds a number of enhancements on core SQL

[1]This option is available as a separate batch execution model.

for more advanced analysis. These enhancements are designed to be fully parallelizable to utilize the underlying MapReduce framework.

4.1 Projection And Filtering

Tenzing supports all the standard SQL operators (arithmetic operators, IN, LIKE, BETWEEN, CASE, etc.) and functions. In addition, the execution engine embeds the Sawzall [19] language so that any built-in Sawzall function can be used. Users can also write their own functions in Sawzall and call them from Tenzing.

The Tenzing compiler does several basic optimizations related to filtering. Some examples include:

1. If an expression evaluates to a constant, it is converted to a constant value at compile time.

2. If a predicate is a constant or a constant range (e.g., BETWEEN) and the source data is an indexed source (e.g., Bigtable), the compiler will push down the condition to an index range scan on the underlying source. This is very useful for making date range scans on fact tables, or point queries on dimension tables, for example.

3. If the predicate does not involve complex functions (e.g., Sawzall functions) and the source is a database (e.g., MySQL), the filter is pushed down to the underlying query executed on the source database.

4. If the underlying store is range partitioned on a column, and the predicate is a constant range on that column, the compiler will skip the partitions that fall outside the scan range. This is very useful for date based fact tables, for example.

5. ColumnIO files have headers which contain meta information about the data, including the low and high values for each column. The execution engine will ignore the file after processing the header if it can determine that the file does not contain any records of interest, based on the predicates defined for that table in the query.

6. Tenzing will scan only the columns required for query execution if the underlying format supports it (e.g., ColumnIO).

4.2 Aggregation

Tenzing supports all standard aggregate functions such as SUM, COUNT, MIN, MAX, etc. and the DISTINCT equivalents (e.g., COUNT DISTINCT). In addition, we support a significant number of statistical aggregate functions such as CORR, COVAR and STDDEV. The implementation of aggregate functions on the MapReduce framework have been discussed in numerous papers including the original MapReduce paper [9]; however, Tenzing employs a few additional optimizations. One such is pure hash table based aggregation, which is discussed below.

4.2.1 HASH BASED AGGREGATION

Hash table based aggregation is common in RDBMS systems. However, it is impossible to implement efficiently on the basic MapReduce framework, since the reducer always unnecessarily sorts the data by key. We enhanced the MapReduce framework to relax this restriction so that all values for the same key end up in the same reducer shard, but not necessarily in the same Reduce() call. This made it possible to completely avoid the sorting step on the reducer and implement a pure hash table based aggregation on MapReduce. This can have a significant impact on the performance on certain types of queries. Due to optimizer limitations, the user must explicitly indicate that they want hash based aggregation. A query using hash-based aggregation will fail if there is not enough memory for the hash table.

Consider the following query:

```
SELECT dept_id, COUNT(1)
FROM Employee
/*+ HASH */ GROUP BY 1;
```

The following is the pseudo-code:

```
// In the mapper startup, initialize a hash table with dept_id
// as key and count as value.
Mapper::Start() {
  dept_hash = new Hashtable()
}

// For each row, increment the count for the corresponding
// dept_id.
Mapper::Map(in) {
  dept_hash[in.dept_id] ++
}

// At the end of the mapping phase, flush the hash table to
// the reducer, without sorting.
Mapper::Finish() {
 for (dept_id in dept_hash) {
   OutputToReducerNoSort(
     key = dept_id, value = dept_hash[dept_id])
 }
}

// Similarly, in the reducer, initialize a hash table
Reducer::Start() {
  dept_hash = new Hashtable()
}

// Each Reduce call receives a dept_id and a count from
// the map output. Increment the corresponding entry in
// the hash table.
Reducer::Reduce(key, value) {
  dept_hash[key] += value
}

// At the end of the reduce phase, flush out the
// aggregated results.
Reducer::Finish() {
  for (dept_id in dept_hash) {
    print dept_id, dept_hash[dept_id]
  }
}
```

4.3 Joins

Joins in MapReduce have been studied by various groups [2,4,22]. Since joins are one of the most important aspects of our system, we have spent considerable time on implementing different types of joins and optimizing them. Tenzing supports efficient joins across data sources, such as ColumnIO to Bigtable; inner, left, right, cross, and full outer joins; and equi semi-equi, non-equi and function based joins. Cross joins are only supported for tables small enough to fit in memory, and right outer joins are supported only with sort/

merge joins. Non-equi correlated subqueries are currently not supported.

We include distributed implementations for nested loop, sort/merge and hash joins. For sort/merge and hash joins, the degree of parallelism must be explicitly specified due to optimizer limitations. Some of the join techniques implemented in Tenzing are discussed below.

4.3.1 BROADCAST JOINS

The Tenzing cost-based optimizer can detect when a secondary table is small enough to fit in memory. If the order of tables specified in the query is not optimal, the compiler can also use a combination of rule and cost based heuristics to switch the order of joins so that the larger table becomes the driving table. If small enough, the secondary table is pulled into the memory of each mapper / reducer process for in-memory lookups, which typically is the fastest method for joining. In some cases, we also use a sorted disk based serialized implementation for the bigger tables to conserve memory. Broadcast joins are supported for all join conditions (CROSS, EQUI, SEMI-EQUI, NON-EQUI), each having a specialized implementation for optimal performance. A few additional optimizations are applied on a case-by-case basis:

- The data structure used to store the lookup data is determined at execution time. For example, if the secondary table has integer keys in a limited range, we use an integer array. For integer keys with wider range, we use a sparse integer map. Otherwise we use a data type specific hash table.
- We apply filters on the join data while loading to reduce the size of the in-memory structure, and also only load the columns that are needed for the query.
- For multi-threaded workers, we create a single copy of the join data in memory and share it between the threads.
- Once a secondary data set is copied into the worker process, we retain the copy for the duration of the query so that we do not have to copy the data for every map shard. This is valuable when there are many map shards being processed by a relatively small number of workers.
- For tables which are both static and frequently used, we permanently cache the data in local disk of the worker to avoid remote reads. Only the first use of the table results in a read into the worker. Subsequent reads are from the cached copy on local disk.
- We cache the join results from the last record; since input data often is naturally ordered on the join attribute(s), it saves us one lookup access.

4.3.2 REMOTE LOOKUP JOINS

For sources which support remote lookups on index (e.g., Bigtable), Tenzing supports remote lookup joins on the key (or a prefix of the key). We employ an asynchronous batch lookup technique combined with a local LRU cache in order to improve performance. The optimizer can intelligently switch table order to enable this if needed.

4.3.3 DISTRIBUTED SORT-MERGE JOINS

Distributed sort-merge joins are the most widely used joins in MapReduce implementations. Tenzing has an implementation which is most effective when the two tables being joined are roughly the same size and neither has an index on the join key.

4.3.4 DISTRIBUTED HASH JOINS

Distributed hash joins are frequently the most effective join method in Tenzing when:

- Neither table fits completely in memory,
- One table is an order of magnitude larger than the other,
- Neither table has an efficient index on the join key.

These conditions are often satisfied by OLAP queries with star joins to large dimensions, a type of query often used with Tenzing.

Consider the execution of the following query:

```
SELECT Dimension.attr, sum(Fact.measure)
FROM Fact
/*+ HASH */ JOIN Dimension USING (dimension_key)
/*+ HASH */ GROUP BY 1;
```

Tenzing processes the query as follows:[2]

```
// The optimizer creates the MapReduce operations required
// for the hash join and aggregation.
Optimizer::CreateHashJoinPlan(fact_table, dimension_table) {
  if (dimension_table is not hash partitioned on
        dimension_key)
    Create MR1: Partition dimension_table by dimension_key
  if (fact_table is not hash partitioned on dimension_key)
    Create MR2: Partition fact_table by dimension_key
  Create MR3: Shard−wise hash join and aggregation

// MR1 and MR2 are trivial repartitions and hence the
// pseudo−code is skipped. Pseudo−code for MR3 is below:

// At the start of the mapper, load the corresponding
// dimension shard into memory. Also initialize a hash
// table for the aggregation.
Mapper::Start() {
  fact_shard.Open()
  shard_id = fact_shard.GetCurrentShard()
  dimension_shard[shard_id].Open()
  lookup_hash = new HashTable()
  lookup_hash.LoadFrom(dimension_shard[shard_id])
  agg_hash = new Hashtable()
}

// For each row, increment the count for the corresponding
// dimension_key.
Mapper::Map(in) {
  dim_rec = lookup_hash(in.dimension_key);
  agg_hash(dim_rec.attr) += in.measure;
}

// At the end of the mapping phase, flush the hash table
// to the reducer, without sorting.
Mapper::Finish() {
 for (attr in agg_hash)
   OutputToReducerNoSort(
     key = attr, value = agg_hash[attr]);
 }
}
```

[2]Note that no sorting is required in any of the MapReduces, so it is disabled for all of them.

The reducer code is identical to the hash aggregation pseudo-code in 4.2.1.

The Tenzing scheduler can intelligently parallelize operations to make hash joins run faster. In this case, for example, the compiler would chain MR3 and MR1, and identify MR2 as a join source. This has the following implications for the backend scheduler:

- MR1 and MR2 will be started in parallel.
- MR3 will be started as soon as MR2 finishes and MR1 starts producing data, without waiting for MR1 to finish.

We also added several optimizations to the MapReduce framework to improve the efficiency of distributed hash joins, such as sort avoidance, streaming, memory chaining and block shuffle. These are covered in more detail in 5.1.

4.4 Analytic Functions

Tenzing supports all the major analytic functions, with a syntax similar to PostgreSQL / Oracle. These functions have proven to be quite popular with the analyst community. Some of the most commonly used analytic functions are RANK, SUM, MIN, MAX, LEAD, LAG and NTILE.

Consider the following example with analytic functions which ranks employees in each department by their salary:

```
SELECT
  dept, emp, salary,
  RANK() OVER (
    PARTITION BY dept ORDER BY salary DESC)
    AS salary_rank
FROM Employee;
```

The following pseudo-code explains the backend implementation:

```
Mapper::Map(in) {
  // From the mapper, we output the partitioning key of
  // the analytic function as the key, and the ordering
  // key and other information as value.
  OutputToReducerWithSort(
    key = in.dept, value = {in.emp, in.salary})
}

Reducer::Reduce(key, values) {
  // Reducer receives all values with the same partitioning
  // key. The list is then sorted on the ordering key for
  // the analytic function.
  sort(values on value.salary)
  // For simple analytic function such as RANK, it is
       enough
  // to just print out the results once sorted.
  for (value in values) {
    print key, value.emp, value.salary, i
  }
}
```

Currently, Tenzing does not support the use of multiple analytic functions with different partitioning keys in the same query. We plan to remove this restriction by rewriting such queries as merging result sets from multiple queries, each with analytic functions having the same partitioning key. Note that it is possible to combine aggregation and analytic functions in the same query - the compiler simply rewrites it into multiple queries.

4.5 OLAP Extensions

Tenzing supports the ROLLUP() and CUBE() OLAP extensions to the SQL language. We follow the Oracle variant of the syntax for these extensions. These are implemented by emitting additional records in the Map() phase based on the aggregation combinations required.

Consider the following query which outputs the salary of each employee and also department-wise and grand totals:

```
SELECT dept, emp, SUM(salary)
FROM Employee
GROUP BY ROLLUP(1, 2);
```

The following pseudo-code explains the backend implementation:

```
// For each row, emit multiple key value pairs, one
// for each combination of keys being aggregated.
Mapper::Map(in) {
  OutputToReducerWithSort(
    key = {dept, emp}, value = salary)
  OutputToReducerWithSort(
    key = {dept, NULL}, value = salary)
  OutputToReducerWithSort(
    key = {NULL, NULL}, value = salary)
}

// The reducer will do a standard pre-sorted
// aggregation on the data.
Reducer::Reduce(key, list<value>) {
  sum := 0
  for (i = 1 to list<value>.size()) {
    sum += value[i].salary
  }
  print key, sum
}
```

4.6 Set Operations

Tenzing supports all standard SQL set operations such as UNION, UNION ALL, MINUS and MINUS ALL. Set operations are implemented mainly in the reduce phase, with the mapper emitting the records using the whole record as the key, in order to ensure that similar keys end up together in the same reduce call. This does not apply to UNION ALL, for which we use round robin partitioning and turn off reducer side sorting for greater efficiency.

4.7 Nested Queries And Subqueries

Tenzing supports SELECT queries anywhere where a table name is allowed, in accordance with the SQL standard. Typically, each nested SQL gets converted to a separate MapReduce and the resultant intermediate table is substituted in the outer query. However, the compiler can optimize away relatively simple nested queries such that extra MapReduce jobs need not be created. For example, the query

```
SELECT COUNT(*) FROM (
  SELECT DISTINCT emp_id FROM Employee);
```

results in two MapReduce jobs: one for the inner DISTINCT and a second for the outer COUNT. However, the query

```
SELECT * FROM (
  SELECT DISTINCT emp_id FROM Employee);
```

results in only one MapReduce job. If two (or more) MapReduces are required, Tenzing will put the reducer and following mapper in the same process. This is discussed in greater detail in 5.1.

```
message Department {
  required int32 dept_id = 1;
  required string dept_name = 2;
  repeated message Employee {
    required int32 emp_id = 3;
    required string emp_name = 4;
  };
  repeated message Location {
    required string city_name = 5;
  }
};
```

Figure 2: Sample protocol buffer definition.

4.8 Handling Structured Data

Tenzing has read-only support for structured (nested and repeated) data formats such as complex protocol buffer structures. The current implementation is somewhat native and inefficient in that the data is flattened by the reader at the lowest level and fed as multiple records to the engine. The engine itself can only deal with flat relational data, unlike Dremel [17]. Selecting fields from different repetition levels in the same query is considered an error.

For example, given the protocol buffer definition in figure 2, the query

```
SELECT emp_name, dept_name FROM Department;
```

is valid since it involves one repeated level (Employee). However, the query

```
SELECT emp_name, city_name FROM Department;
```

is invalid since Employee and Location are independently repeating groups.

4.9 Views

Tenzing supports logical views over data. Views are simply named SELECT statements which are expanded inline during compilation. Views in Tenzing are predominantly used for security reasons: users can be given access to views without granting them access to underlying tables, enabling row and column level security of data.

Consider the table Employee with fields emp_id, ldap_user, name, dept_id, and salary. We can create the following view over it:

```
CREATE VIEW Employee_V AS
SELECT emp_id, ldap\_user, name, dept_id
FROM Employee
WHERE dept_id IN (
  SELECT e.dept_id FROM Employee e
  WHERE e.ldap\_user=USERNAME());
```

This view allows users to see only other employees in their department, and prevents users from seeing the salary of other employees.

4.10 DML

Tenzing has basic support for DML operations INSERT, UPDATE and DELETE. Support is batch mode (i.e. meant for batch DML application of relatively large volumes of data). Tenzing is not ACID compliant - specifically, we are atomic, consistent and durable, but do not support isolation.

INSERT is implemented by creating a new data set and adding the new data set to the existing metadata. Essentially, this acts as a batch mode APPEND style INSERT. Tenzing allows the user to specify, but does not enforce, primary and foreign keys.

Limited support (no joins) for UPDATE and DELETE is implemented by applying the update or delete criteria on the data to create a new dataset. A reference to the new dataset then replaces the old reference in the metadata repository.

4.11 DDL

We support a number of DDL operations, including CREATE [OR REPLACE] [EXTERNAL] [TEMPORARY] TABLE, DROP TABLE [IF EXISTS], RENAME TABLE, GENERATE STATISTICS, GRANT and REVOKE. They all work as per standard SQL and act as the main means of controlling metadata and access. In addition, Tenzing has metadata discovery mechanisms built-in to simplify importing datasets into Tenzing. For example, we provide tools and commands to automatically determine the structure of a MySQL database and make its tables accessible from Tenzing. We can also discover the structure of protocol buffers from the protocol definition and import the metadata into Tenzing. This is useful for Bigtable, ColumnIO, RecordIO and SSTable files in protocol buffer format.

4.12 Table Valued Functions

Tenzing supports both scalar and table-valued user-defined functions, implemented by embedding a Sawzall interpreter in the Tenzing execution engine. The framework is designed such that other languages can also be easily integrated. Integration of Lua and R has been proposed, and work is in progress. Tenzing currently has support for creating functions in Sawzall that take tables (vector of tuples) as input and emit tables as output. These are useful for tasks such as normalization of data and doing complex computation involving groups of rows.

4.13 Data Formats

Tenzing supports direct querying of, loading data from, and downloading data into many formats. Various options can be specified to tweak the exact form of input / output. For example, for delimited text format, the user can specify the delimiter, encoding, quoting, escaping, headers, etc. The statement below will create the Employee table from a pipe delimited text file input and validate that the loaded data matches the table definition and all constraints (e.g., primary key) are met:

```
CREATE TABLE
  Employee(emp_id int32, emp_name string)
WITH DATAFILE:CSV[delim="|"]:"employee.txt"
WITH VALIDATION;
```

Other formats supported by Tenzing include:

- ColumnIO, a columnar storage system developed by the Dremel team [17].
- Bigtable, a highly distributed key-value store [7].
- Protocol buffers [11] stored in compressed record format (RecordIO) and sorted strings format (SSTables [7]).
- MySQL databases.
- Data embedded in the metadata (useful for testing and small static data sets).

5. PERFORMANCE

One of the key aims of Tenzing has been to have performance comparable to traditional MPP database systems such as Teradata, Netezza and Vertica. In order to achieve this, there are several areas that we had to work on:

5.1 MapReduce Enhancements

Tenzing is tightly integrated with the Google MapReduce implementation, and we made several enhancements to the MapReduce framework to increase throughput, decrease latency and make SQL operators more efficient.

Workerpool. One of the key challenges we faced was reducing latency from minutes to seconds. It became rapidly clear that in order to do so, we had to implement a solution which did not entail spawning of new binaries for each new Tenzing query. The MapReduce and Tenzing teams collaboratively came up with the pool implementation. A typical pool consists of three process groups:

1. The master watcher. The watcher is responsible for receiving a work request and assigning a free master for the task. The watcher also monitors the overall health of the pool such as free resources, number of running queries, etc. There is usually one one watcher process for one instance of the pool.

2. The master pool. This consists of a relatively small number of processes (usually a few dozen). The job of the master is to coordinate the execution of one query. The master receives the task from the watcher and distributes the tasks to the workers, and monitors their progress. Note that once a master receives a task, it takes over ownership of the task, and the death of the watcher process does not impact the query in any way.

3. The worker pool. This contains a set of workers (typically a few thousand processes) which do all the heavy lifting of processing the data. Each worker can work as either a mapper or a reducer or both. Each worker constantly monitors a common area for new tasks and picks up new tasks as they arrive on a FIFO basis. We intend to implement a priority queue so that queries can be tiered by priority.

Using this approach, we were able to bring down the latency of the execution of a Tenzing query itself to around 7 seconds. There are other bottlenecks in the system however, such as computation of map splits, updating the metadata service, committing / rolling back results (which involves file renames), etc. which means the typical latency varies between 10 and 20 seconds currently. We are working on various other enhancements and believe we can cut this time down to less than 5 seconds end-to-end, which is fairly acceptable to the analyst community.

Streaming & In-memory Chaining. The original implementation of Tenzing serialized all intermediate data to GFS. This led to poor performance for multi-MapReduce queries, such as hash joins and nested sub-selects. We improved the performance of such queries significantly by implementing streaming between MapReduces, i.e. the upstream and downstream MRs communicate using the network and only use GFS for backup. We subsequently improved performance further by using memory chaining, where the reducer of the upstream MR and the mapper of the downstream MR are co-located in the same process.

Sort Avoidance. Certain operators such as hash join and hash aggregation require shuffling, but not sorting. The MapReduce API was enhanced to automatically turn off sorting for these operations. When sorting is turned off, the mapper feeds data to the reducer which directly passes the data to the Reduce() function bypassing the intermediate sorting step. This makes many SQL operators significantly more efficient.

Block Shuffle. Typically, MapReduce uses row based encoding and decoding during shuffle. This is necessary since in order to sort the data, rows must be processed individually. However, this is inefficient when sorting is not required. We implemented a block-based shuffle mechanism on top of the existing row-based shuffler in MapReduce that combines many small rows into compressed blocks of roughly 1MB in size. By treating the entire block as one row and avoiding reducer side sorting, we were able to avoid some of the overhead associated with row serialization and deserialization in the underlying MapReduce framework code. This lead to 3X faster shuffling of data compared to row based shuffling with sorting.

Local Execution. Another simple MapReduce optimization we do is local run. The backend can detect the size of the underlying data to be processed. If the size is under a threshold (typically 128 MB), the query is not sent to the pool, but executed directly in the client process. This reduces the query latency to about 2 seconds.

5.2 Scalability

Because it is built on the MapReduce framework, Tenzing has excellent scalability characteristics. The current production deployment runs in two data centers, using 2000 cores each. Each core has 6 GB of RAM and 24 GB of local disk, mainly used for sort buffers and local caches (Note that the data is primarily stored in GFS and Bigtable). We benchmarked the system for the simple query

```
SELECT a, SUM(b)
FROM T
WHERE c = k
GROUP BY a
```

with data stored in ColumnIO format on GFS files (see table 1). Throughput, measured in rows per second per worker, remains steady as the number of workers is scaled up.

Workers	Time (s)	Throughput
100	188.74	16.74
500	36.12	17.49
1000	19.57	16.14

Table 1: Tenzing Scalability.

5.3 System Benchmarks

In order to evaluate Tenzing performance against commercial parallel databases, we benchmarked four commonly used analyst queries (see appendix A) against DBMS-X, a leading MPP database appliance with row-major storage. Tenzing data was stored in GFS in ColumnIO format. The Tenzing setup used 1000 processes with 1 CPU, 2 GB RAM

Query	DBMS-X (s)	Tenzing (s)	Change
#2	129	93	39% faster
#4	70	69	1.4% faster
#1	155	213	38% slower
#3	9	28	3.1 times slower

Table 2: Tenzing versus DBMS-X.

			Throughput		
	Rows	Selected	Vector	LLVM	Ratio
#8	104	99.58%	21	89	4.2
#6	60	100.00%	8.1	25	3.2
#1	19	99.16%	0.66	2.0	3.0
#2	19	0.81%	23	61	2.6
#7	104	58.18%	6.7	17	2.5
#3	57	3.74%	7.4	15	2.0
#5	104	4.28%	13	19	1.5
#4	40	2.49%	12	13	1.1

Table 3: LLVM and Vector Engine Benchmarks.

and 8 GB local disk each. Results are shown in table 2. The poor performance on query #3 is because query execution time is dominated by startup time. The production version of Tenzing was used for the benchmarks; we believe Tenzing's results would have been significantly faster if the experimental LLVM engine discussed in the next section had been ready at benchmark time.

5.4 Experimental LLVM Query Engine

Our execution engine has gone through multiple iterations to achieve single-node efficiency close to commercial DBMS. The first implementation translated SQL expressions to Sawzall code. This code was then compiled using Sawzall's just-in-time (JIT) compiler. However, this proved to be inefficient because of the serialization and deserialization costs associated with translating to and from Sawzall's native type system. The second and current implementation uses Dremel's SQL expression evaluation engine, which is based on direct evaluation of parse trees of SQL expressions. While more efficient than the original Sawzall implementation, it was still somewhat slow because of its interpreter-like nature and row based processing.

For the third iteration, we did extensive experiments with two major styles of execution: LLVM based native code generation with row major block based intermediate data and column major vector based processing with columnar intermediate storage. The results of benchmarking LLVM vs vector on some typical aggregation queries is shown in table 3. All experiments were done on the same dual-core Intel machine with 4 GB of RAM, with input data in columnar form in-memory. Note that the LLVM engine is a work in progress and has not yet been integrated into Tenzing. Table sizes are in millions of rows and throughput is again measured in millions of rows per worker per second. The data suggests that the higher the selectivity of the where clause, the better the LLVM engine's relative performance. We found that the LLVM approach gave better overall results for real-life queries while vector processing was somewhat better for pure select-project queries with high selectivity. In comparison to the production evaluation engine, the vector engine's per-worker throughput was about three times higher; the LLVM engine's per-worker throughput was six to twelve times higher.

Our LLVM based query engine stores intermediate results by rows, even when both input and output are columnar. In comparison to columnar vector processing, we saw several advantages and disadvantages.

- For hash table based operators like aggregation and join, using rows is more natural: composite key comparison has cache locality, and conflict resolution is done using pointers. Our engine iterates on input rows and uses generated procedures that do both. However to our best knowledge, there is no straightforward and fast columnar solution for using hash table. Searching in hash table breaks the cache locality for columns, resulting in more random memory accesses.

- Vector processing engines always materialize intermediate results in main memory. In contrast our generated native code can store results on the stack (better data locality) or even registers, depending on how the JIT compiler optimizes.

- If selectivity of source data is low, vector processing engines load less data into cache and thus scan faster. Because our engine stores data in rows, scans end up reading more data and are slower.

- While LLVM provides support for debugging JITed code [16], there are more powerful analysis and debugging tools for the native C/C++ routines used by most vector processing engines.

The majority of queries in our workload are analytic queries that have aggregations and/or joins. These operations are significant performance bottlenecks compared to other operators like selection and projection. We believe that native code generation engine deals better with these bottlenecks and is a promising approach for query processing.

6. RELATED WORK

A significant amount of work has been done in recent years in the area of large scale distributed data processing. These fall into the following broad categories:

- Core frameworks for distributed data processing. Our focus has been on MapReduce [9], but there are several others such as Nephele/PACT [3] and Dryad [15]. Hadoop [8] is an open source implementation of the MapReduce framework. These powerful but complex frameworks are designed for software engineers implementing complex parallel algorithms.

- Simpler procedural languages built on top of these frameworks. The most popular of these are Sawzall [19] and PIG [18]. These have limited optimizations built in, and are more suited for reasonably experienced analysts, who are comfortable with a procedural programming style, but need the ability to iterate quickly over massive volumes of data.

- Language extensions, usually with special purpose optimizers, built on top of the core frameworks. The ones we have studied are FlumeJava [6], which is built on top of MapReduce, and DryadLINQ [23], which is built on top of Dryad. These seem to be mostly geared towards programmers who want to quickly build large scalable pipelines with relatively simple operations (e.g. ETL pipelines for data warehouses).

- Declarative query languages built on top of the core frameworks with intermediate to advanced optimizations. These are geared towards the reporting and analysis community, which is very comfortable with SQL. Tenzing is mostly focused on this use-case, as we believe, are HIVE [21], SCOPE [5] and HadoopDB [1] (recently commercialized as Hadapt). The latter is an interesting hybrid that tries to approach the performance of parallel DBMS by using a cluster of single-node databases and MapReduce as the glue layer.

- Embedding MapReduce and related concepts into traditional parallel DBMSs. A number of vendors have such offerings now, including Greenplum, AsterData, Paraccel and Vertica.

7. CONCLUSION

We described Tenzing, a SQL query execution engine built on top of MapReduce. We demonstrated that:

- It is possible to create a fully functional SQL engine on top of the MapReduce framework, with extensions that go beyond SQL into deep analytics.

- With relatively minor enhancements to the MapReduce framework, it is possible to implement a large number of optimizations currently available in commercial database systems, and create a system which can compete with commercial MPP DBMS in terms of throughput and latency.

- The MapReduce framework provides a combination of high performance, high reliability and high scalability on cheap unreliable hardware, which makes it an excellent platform to build distributed applications that involve doing simple to medium complexity operations on large data volumes.

- By designing the engine and the optimizer to be aware of the characteristics of heterogeneous data sources, it is possible to create a smart system which can fully utilize the characteristics of the underlying data sources.

8. ACKNOWLEDGEMENTS

Tenzing is built on top of a large number of existing Google technologies. While it is not possible to list all individuals who have contributed either directly or indirectly towards the implementation, we would like to highlight the contribution of the following:

- The data warehouse management, specifically Hemant Maheshwari, for providing business context, management support and overall guidance.

- The MapReduce team, specifically Jerry Zhao, Marián Dvorský and Derek Thomson, for working closely with us in implementing various enhancements to MapReduce.

- The Dremel team, specifically Sergey Melnik and Matt Tolton, for ColumnIO storage and the SQL expression evaluation engine.

- The Sawzall team, specifically Polina Sokolova and Robert Griesemer, for helping us embed the Sawzall language which we use as the primary scripting language for supporting complex user-defined functions.

- Various other infrastructure teams at Google, including but not limited to the Bigtable, GFS, Machines and SRE teams.

- Our loyal and long-suffering user base, especially the people in Sales & Finance.

9. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2:922–933, August 2009.

[2] F.N. Afrati and J.D. Ullman. Optimizing joins in a Map-Reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.

[3] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[4] 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 *Proceedings of the 2010 international conference on Management of data*, pages 975–986. ACM, 2010.

[5] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, August 2008.

[6] C. Chambers, A. Raniwala, F. Perry, S. Adams, R.R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 363–375. ACM, 2010.

[7] 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. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[8] D. Cutting et al. Apache Hadoop Project. http://hadoop.apache.org/.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[11] J. Dean, S. Ghemawat, K. Varda, et al. Protocol Buffers: Google's Data Interchange Format. Documentation and open source release at http://code.google.com/p/protobuf/.

[12] D.J. DeWitt and M. Stonebraker. MapReduce: A major step backwards. *The Database Column*, 1, 2008.

[13] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, 2(2):1402–1413, 2009.

[14] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37:29–43, October 2003.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.

[16] R. Kleckner. LLVM: Debugging JITed Code With GDB. Retrieved June 27, 2011, from http://llvm.org/docs/DebuggingJITedCode.html.

[17] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment*, 3(1), 2010.

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[19] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[20] M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[21] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a Map-Reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[22] H. Yang, A. Dasdan, R.L. Hsiao, and D.S. Parker. Map-Reduce-Merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.

[23] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

APPENDIX

A. BENCHMARK QUERIES

This section contains the queries used for benchmarking performance against DBMS-X. The table sizes are indicated using XM or XK suffix.

A.1 Query 1

```
SELECT DISTINCT dim2.dim1_id
FROM FactTable1_XXB stats
INNER JOIN DimensionTable1_XXXM dim1
  USING (dimension1_id)
INNER JOIN DimensionTable2_XXM dim2
  USING (dimension2_id)
WHERE dim2.attr BETWEEN A and B
AND stats.date_id > some_date_id
AND dim1.attr IN (Val1, Val2, Val3, ...);
```

A.2 Query 2

```
SELECT
  dim1.attr1, dates.finance_week_id,
  SUM(FN1(dim3.attr3, stats.measure1)),
  SUM(FN2(dim3.attr4, stats.measure2)),

FROM FactTable1_XXB stats
INNER JOIN DimensionTable1_XXXM dim1
  USING (dimension1_id)
INNER JOIN DatesDim dates
  USING (date_id)
INNER JOIN DimensionTable3_XXXK dim3
  USING (dimension3_id)
WHERE <fact date range>
GROUP BY 1, 2;
```

A.3 Query 3

```
SELECT attr4 FROM (
  SELECT dim4.attr4, COUNT(*) AS dup_count
  FROM DimensionTable4_XXM dim4
  JOIN DimensionTable5_XXM dim5
    USING (dimension4_id)
  WHERE dim4.attr1 BETWEEN Val1 and Val2
  AND dim5.attr2 IN (Val3, Val4)
  GROUP BY 1
  HAVING dup_count = 1) x;
```

A.4 Query4

```
SELECT attr1, measure1 / measure2
FROM (
  SELECT
    attr1,
    SUM(FN1(attr2, attr3, attr4)) measure1,
    SUM(FN2(attr5, attr6, attr7)) measure2,
  FROM DimensionTable6_XXM
  WHERE FN3(attr8) AND FN4(attr9)
  GROUP BY 1
) v;
```