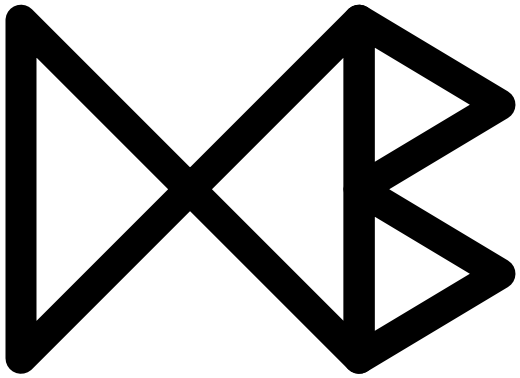

ETLMR: A Highly Scalable Dimensional ETL Framework based on MapReduce

Xiufeng Liu, Christian Thomsen and Torben Bach Pedersen

August, 2011

TR-29

A DB Technical Report

Title	ETLMR: A Highly Scalable Dimensional ETL Framework based on MapReduce
Author(s)	Xiufeng Liu, Christian Thomsen and Torben Bach Pedersen
Publication History	Extended version of: Xiufeng Liu, Christian Thomsen and Torben Bach Pedersen: "ETLMR: A Highly Scalable Dimensional ETL Framework based on MapReduce" in *Proceedings of 13th International Conference on Data Warehousing and Knowledge*, Toulouse, France, August 2011, pp. 96-111

For additional information, see the DB TECH REPORTS homepage: ⟨`dbtr.cs.aau.dk`⟩.

The DB TECH REPORTS icon is made from two letters in an early version of the Rune alphabet, which was used by the Vikings, among others. Runes have angular shapes and lack horizontal lines because the primary storage medium was wood, although they may also be found on jewelry, tools, and weapons. Runes were perceived as having magic, hidden powers. The first letter in the logo is "Dagaz," the rune for day or daylight and the phonetic equivalent of "d." Its meanings include happiness, activity, and satisfaction. The second letter is "Berkano," which is associated with the birch tree. Its divinatory meanings include health, new beginnings, growth, plenty, and clearance. It is associated with Idun, goddess of Spring, and with fertility. It is the phonetic equivalent of "b."

Abstract

Extract-Transform-Load (ETL) flows periodically populate data warehouses (DWs) with data from different source systems. An increasing challenge for ETL flows is processing huge volumes of data quickly. MapReduce is establishing itself as the de-facto standard for large-scale data-intensive processing. However, MapReduce lacks support for high-level ETL specific constructs, resulting in low ETL programmer productivity. This report presents a scalable dimensional ETL framework, *ETLMR*, based on MapReduce. ETLMR has built-in native support for operations on DW-specific constructs such as star schemas, snowflake schemas and slowly changing dimensions (SCDs). This enables ETL developers to construct scalable MapReduce-based ETL flows with very few code lines. To achieve good performance and load balancing, a number of dimension and fact processing schemes are presented, including techniques for efficiently processing different types of dimensions. The report describes the integration of ETLMR with a MapReduce framework and evaluates its performance on large realistic data sets. The experimental results show that ETLMR achieves very good scalability and compares favourably with other MapReduce data warehousing tools.

1 Introduction

In data warehousing, ETL flows are responsible for collecting data from different data sources, transformation, and cleansing to comply with user-defined business rules and requirements. Traditional ETL technologies face new challenges as the growth of information explodes nowadays, e.g., it becomes common for an enterprise to collect hundreds of gigabytes of data for processing and analysis each day. The vast amount of data makes ETL extremely time-consuming, but the time window assigned for processing data typically remains short. Moreover, to adapt rapidly changing business environments, users have an increasing demand of getting data as soon as possible. The use of parallelization is the key to achieve better performance and scalability for those challenges. In recent years, a novel "cloud computing" technology, *MapReduce* [9], has been widely used for parallel computing in data-intensive areas. A MapReduce program is written as *map* and *reduce* functions, which process key/value pairs and are executed in many parallel instances.

We see that MapReduce can be a good foundation for the ETL parallelization. In ETL, the data processing exhibits the *composable* property such that the processing of dimensions and facts can be split into smaller computation units and the partial results from these computation units can be merged to constitute the final results in a DW. This complies well with the MapReduce paradigm in term of *map* and *reduce*.

ETL flows are inherently complex, which is due to the plethora of ETL-specific activities such as transformation, cleansing, filtering, aggregating and loading. Programming of highly parallel and distributed systems is also challenging. To implement an ETL program to function in a distributed environment is thus very costly, time-consuming, and error-prone. MapReduce, on the other hand, provides programming flexibility, cost-effective scalability and capacity on commodity machines and a MapReduce framework can provide inter-process communication, fault-tolerance, load balancing and task scheduling to a parallel ETL program out of the box. Further, MapReduce is a very popular framework and is establishing itself as the de-facto standard for large-scale data-intensive processing. It is thus interesting to see how MapReduce can be applied to the field of ETL programming.

MapReduce is, however, a generic programming model. It lacks support for high-level DW/ETL specific constructs such as the dimensional constructs of star schemas, snowflake schemas, and SCDs. This results in low ETL programmer productivity. To implement a parallel ETL program on MapReduce is thus still not easy because of the inherent complexity of ETL-specific activities such as the processing for different schemas and SCDs.

In this report, we present a parallel dimensional ETL framework based on MapReduce, named *ETLMR*, which directly supports high-level ETL-specific dimensional constructs such as star schemas, snowflake schemas, and SCDs. We believe this to be the first report to specifically address ETL for *dimensional*

schemas on MapReduce. The report makes several contributions: We leverage the functionality of MapReduce to the ETL parallelization and provide a scalable, fault-tolerable, and very lightweight ETL framework which hides the complexity of MapReduce. We present a number of novel methods which are used to process the dimensions of a star schema, snowflaked dimensions, SCDs and data-intensive dimensions. In addition, we introduce the offline dimension scheme which scales better than the online dimension scheme when handling massive workloads. The evaluations show that ETLMR achieves very good scalability and compares favourably with other MapReduce data warehousing tools.

The running example: To show the use of ETLMR, we use a running example throughout this report. This example is inspired by a project which applies different tests to web pages. Each test is applied to each page and the test outputs the number of errors detected. The test results are written into a number of tab-separated files, which serve as the data sources. The data is processed to be stored in a DW with the star schema shown in Figure 1. This schema comprises a fact table and three dimension tables. Note that *pagedim* is a slowly changing dimension. Later, we will consider a partly snowflaked (i.e., normalized) schema.

The remainder of this report is structured as follows: Section 2 gives a brief review of the MapReduce programming model. Section 3 gives an overview of ETLMR. Sections 4 and 5 present dimension processing and fact processing, respectively. Section 7 introduces the implementation of ETLMR in the Disco MapReduce framework, and presents the experimental evaluation. Section 8 reviews related work. Finally, Section 9 concludes the report and provides ideas for future work.

2 MapReduce Programming Model

The programming model of MapReduce [9] expresses parallel computations into two primitives: *map* and *reduce*, taking a list of input key/value pairs, and producing a list of output key/value pairs.

```
Map: (k1, v1) ->  list(k2, v2)
Reduce: (k2, list(v2)) -> list(v3)
```

Map, defined by users, takes an input $(k1, v1)$ pair and produces a list of intermediate key/value $(k2, v2)$ pairs. MapReduce then groups all intermediate values with the same intermediate key $k2$, and passes them on. *Reduce*, also defined by users, then takes the pair of key $k2$ and the list of values for $k2$, and merges, e.g., aggregates, together these values to form a possibly smaller list of values $list(v3)$.

Besides the `map` and `reduce` interfaces, there are 5 other standard programming interfaces offered by most MapReduce frameworks, including interfaces for input reading, data partitioning, combining map output, sorting, and output writing. Users can tailor or extend these interfaces according to their requirements. A MapReduce framework achieves parallel computations by executing the implemented interfaces on clustered computers, each processing a chunk of the data sets.

3 ETLMR Overview

In this section, we give an overview of ETLMR on a MapReduce framework, and describe the details of the data processing phases.

To show the use of ETLMR, we use a running example throughout this report. This example is inspired by the work we did in the European Internet Accessibility Observatory (EIAO) project [16], which automates the testing of the accessibility of web pages. A test is applied to each page and the test outputs the number of errors detected. The test results are written into a number of tab-separated files, which serve as the data sources. The data is processed to be stored in a DW with the star schema shown in Figure 1. This schema comprises a fact table and three dimension tables. Note that *pagedim* is a slowly changing dimension. Later, we will consider a partly snowflaked (i.e., normalized) schema.

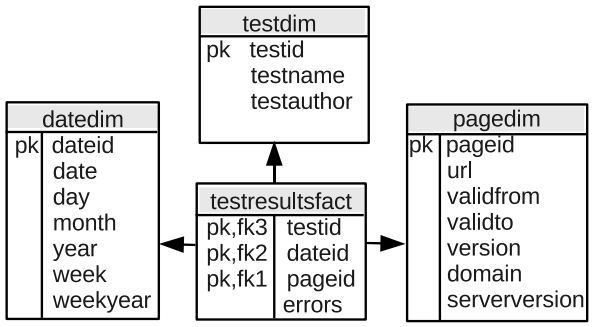

Figure 1: Star schema of the running example

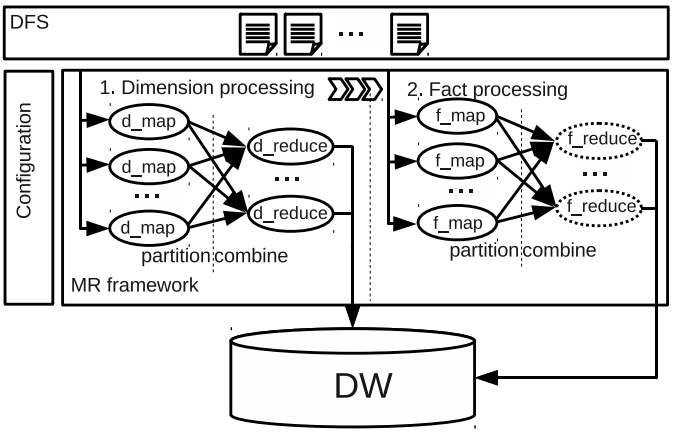

Figure 2: ETL Data flow on MapReduce framework

Figure 2 illustrates the data flow using ETLMR on MapReduce. In ETLMR, the dimension processing is done at first in a MapReduce job, then the fact processing is done in another MapReduce job. A MapReduce job spawns a number of parallel map/reduce tasks[1] for processing dimension or fact data. Each task consists of several steps, including reading data from a distributed file system (DFS), executing the map function, partitioning, combining the map output, executing the reduce function and writing results. In dimension processing, the input data for a dimension table can be processed by different processing methods, e.g., the data can be processed by a single task or by all tasks. In fact processing, the data for a fact table is partitioned into a number of equal-sized data files which then are processed by parallel tasks. This includes looking up dimension keys and bulk loading the processed fact data into the DW. The processing of fact data in the reducers can be omitted (shown by dotted ellipses in Figure 2) if no aggregation of the fact data is done before it is loaded.

Before a MapReduce job starts, the data sets from heterogeneous storage systems are partitioned into multiple approximately equal sized pieces, which are distributed to map/reduce tasks, and subsequently processed into dimensions or facts. ETLMR employs the following two partitioning methods. 1) *Round-robin partitioning*: This method distributes rows among the tasks such that row number n is assigned to task number $(n \bmod nr\_map)$ where nr_map is the number of tasks. It ensures that the input data sets are evenly divided among the tasks. This method is suitable when a dimension's data should be processed by

[1] map/reduce task denotes map tasks and reduce tasks running separately.

all tasks. 2) *Hash by field partitioning*: This method designates one or more attributes as the partitioning attributes. The tuples with the same hash values on the partitioning attributes are assigned to the same task. If there are nr_map tasks, a tuple with hash value $hash$ is assigned to task number ($hash$ mod nr_map). This method is suitable when all tuples with identical values in the hash attributes should be processed by a single task.

ETLMR provides map readers implementing the above two partitioning methods. In addition, most MapReduce frameworks provide different kind of readers for selection and allow users to customize their own readers.

In the dimension processing, the input data sets for dimensions are processed among all map/reduce tasks, using configurable processing methods. The most fundamental of these is to configure one dimension per task, e.g., given 3 dimensions and 3 tasks, each task processes the data sets of a single dimension. Unlike this method where only a limited number of tasks can be used, another approach is to let all tasks process a given dimension such that each task processes parts of the dimension's data sets. If there are dependencies between dimensions, such as snowflaked (or normalized) dimensions, special processing methods (*level-wise* or *hierarchy-wise* processing), are configured for dealing with the processing order and parallelization of the dimensions. In a further optimization, dimensions can be configured to be stored distributedly over the nodes, and loaded into the DW on demand.

In the fact processing, each map/reduce task processes an equally-sized data set, including reading data, looking up key values from dimension tables, transformation and loading. If a fact is an aggregated fact, reducers are configured for computing measures on all tuples using aggregation functions such as `sum`, `average` and `count`. To optimize performance, reducers can be omitted if no aggregation is required (dotted ellipses shown in Figure 2). Further, dimensions can be read fully or partially into main memory to speed up the lookups of dimension keys, and bulk-loads are employed to transfer the processed data from main memory to the DW at run-time.

Algorithm 1 details the whole process. The operations in lines 2-4 and 6-7 are the MapReduce steps which are responsible for initialization, invoking jobs for processing dimensions and facts, and returning processing information. Line 1 and 5 are the non-MapReduce steps which are used for preparing input data sets and synchronizing dimensions among nodes (if no distributed file system (DFS) is installed).

Algorithm 1 ETL process on MapReduce framework

1: Partition the input data sets;
2: Read the configuration parameters (Table 1) and initialize;
3: Read the input data and relay the data to the map function in the map readers;
4: Process dimension data and load it into online/offline dimension stores;
5: Synchronize the dimensions across the clustered computers, if applicable;
6: Prepare fact processing (connect to and cache dimensions);
7: Read the input data for fact processing and perform transformations in mappers;
8: Bulk-load fact data into the DW.

In ETLMR, all run-time parameters are stored in a single configuration file, including the settings of data sources, partitioning methods such as the keys for partitioning, dimensions, facts, data-intensive (big) dimensions, and the number of mappers and reducers. Table 1 summarizes the key configuration parameters. These parameters provide users with flexibility to configure the tasks to be more efficient. For example, if a user knows that a dimension is a data-intensive dimension, (s)he can add it to the list $bigdims$ so that an appropriate processing method can be chosen to achieve better performance and load balancing.

Algorithm 1 details the ETLMR process on a MapReduce framework. The operations in lines 2-4 and 6-7 are MapReduce steps, which are responsible for initialization, invoking jobs for processing dimensions and facts, and returning processing information. Line 1 and 5 are non-MapReduce steps, which are used for

Table 1: The key configuration parameters

Parameters	Description
Dim_i	Dimension table definition, $i = 1, \ldots, n$
$Fact_i$	Fact table definition, $i = 1, \ldots, m$
Set_{bigdim}	data-intensive dimensions whose business keys are used for partitioning the data sets if applicable
$Dim_i(a_0, a_1, ..., a_n)$	Define the relevant attributes $a_0, a_1, ..., a_n$ of Dim_i in data source
$DimScheme$	Dimension scheme, online/offline (online is the default)
nr_reduce	Number of reducers
nr_map	Number of mappers

preparing input data sets and synchronizing dimensions among nodes (if no distributed file system (DFS) is installed).

In ETLMR, all run-time parameters are stored in a single configuration file, including the settings of data sources, partitioning methods such as the keys for partitioning, dimensions, facts, data-intensive (big) dimensions, and the number of mappers and reducers. Table 1 summarizes the key configuration parameters. These parameters provide users with flexibility to configure the tasks to be more efficient. For example, if a user knows that a dimension is a data-intensive dimension, (s)he can add it to the list $bigdims$ so that an appropriate processing method can be chosen to achieve better performance and load balancing.

4 Dimension Processing

In ETLMR, each dimension table has a corresponding definition in the configuration file. For example, we define the object for the dimension table *testdim* of the running example by *testdim = CachedDimension(name='testdim', key='testid', defaultidvalue =-1, attributes=['testname', 'testauthor'], lookupatts=['testname',])*. It is declared as a cached dimension which means that its data can be temporarily kept in memory. ETLMR also offers other dimension classes for declaring different dimension tables, including *SlowlyChangingDimension* and *SnowflakedDimension*, each of which are configured by means of a number of parameters for specifying the name of the dimension table, the dimension key, the attributes of dimension table, the lookup attributes (which identify a row uniquely), and others. Each class offers a number of functions for dimension operations such as *lookup*, *insert*, *ensure*, etc.

ETLMR employs MapReduce's primitives *map*, *partition*, *combine*, and *reduce* to process data. This is, however, hidden from the user who only specifies transformations applied to the data and declarations of dimension tables and fact tables. A map/reduce task reads data by iterating over lines from a partitioned data set. A line is first processed by *map*, then by *partition* which determines the target reducer, and then by *combine* which groups values having the same key. The data is then written to an intermediate file (there is one file for each reducer). In the reduce step, a reduce reader reads a list of key/values pairs from an intermediate file and invokes *reduce* to process the list. In the following, we present different approaches to process dimension data.

4.1 One Dimension One Task

In this approach, map tasks process data for all dimensions by applying user-defined transformations and by finding the relevant parts of the source data for each dimension. The data for a given dimension is then processed by a single reduce task. We name this method *one dimension one task* (*ODOT* for short).

The data unit moving around within ETLMR is a dictionary mapping attribute names to values. Here, we call it a *row*, e.g., *row={'url':'www.dom0.tl0/p0.htm','size': '12553','serverversion':'SomeServer/1.0',*

'downloaddate':'2011-01-31','lastmoddate': '2011-01-01', 'test':'Test001', 'errors':'7'}. ETLMR reads lines from the input files and passes them on as rows. A mapper does projection on rows to prune unnecessary data for each dimension and makes key/value pairs to be processed by reducers. If we define dim_i for a dimension table and its relevant attributes, $(a_0, a_1..., a_n)$, in the data source schema, the mapper will generate the map output, $(key, value) = (dim_i.name, \prod_{a_0,a_1,...,a_n}(row))$ where $name$ represents the name of dimension table. The MapReduce partitioner partitions map output based on the key, i.e., $dim_i.name$, such that the data of dim_i will go to a single reducer (see Figure 3). To optimize, the values with identical keys (i.e., dimension table name) are combined in the combiner before they are sent to the reducers such that the network communication cost can be reduced. In a reducer, a row is first processed by UDFs to do data transformations, then the processed row is inserted into the dimension store, i.e., the dimension table in the DW or in an offline dimension store (described later). When ETLMR does this data insertion, it has the following *reduce* functionality: If the row does not exist in the dimension table, the row is inserted. If the row exists and its values are unchanged, nothing is done. If there are changes, the row in the table is updated accordingly. The ETLMR dimension classes provide this functionality in a single function, $dim_i.ensure(row)$. For an SCD, this function adds a new version if needed, and updates the values of the SCD attributes, e.g., the validto and version.

We have now introduced the most fundamental method for dimension processing where only a limited number of reducers can be utilized. Therefore, its drawback is that it is not optimized for the case where some dimensions contain large amounts of data, namely data-intensive dimensions.

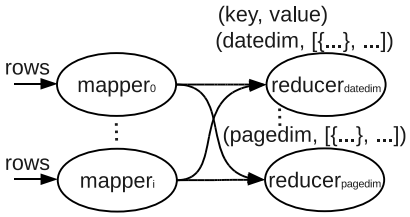

Figure 3: ODOT

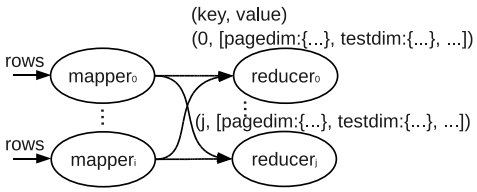

Figure 4: ODAT

4.2 One Dimension All Tasks

We now describe another approach in which all reduce tasks process data for all dimensions. We name it *one dimension all tasks* (*ODAT* for short). In some cases, the data volume of a dimension is very large, e.g., the *pagedim* dimension in the running example. If we employ ODOT, the task of processing data for this dimension table will determine the overall performance (assume all tasks run on similar machines). We therefore refine the ODOT in two places, the map output partition and the reduce functions. With ODAT, ETLMR partitions the map output by round-robin partitioning such that the reducers receive equally many rows (see Figure 4). In the reduce function, two issues are considered in order to process the dimension data properly by the parallel tasks:

The first issue is how to keep the uniqueness of dimension key values as the data for a dimension table is processed by all tasks. We propose two approaches. The first one is to use a global ID generator and use *post-fixing* (detailed in Section 4.4) to merge rows having the same values in the dimension *lookup* attributes (but different key values) into one row. The other approach is to use private ID generators and post-fixing. Each task has its own ID generator, and after the data is loaded into the dimension table, post-fixing is employed to fix the resulting duplicated key values. This requires the uniqueness constraint on the dimension key to be disabled before the data processing.

The second issue is how to handle concurrency problem when data manipulation language (DML) SQL such as UPDATE, DELETE, etc. is issued by several tasks. Consider, for example, the type-2 SCD table

pagedim for which INSERTs and UPDATEs are frequent (the SCD attributes *validfrom* and *validto* are updated). There are at least two ways to tackle this problem. The first one is row-based commit in which a COMMIT is issued after every row has been inserted so that the inserted row will not be locked. However, row-based commit is more expensive than transaction commit, thus, it is not very useful for a data-intensive dimension table. Another and better solution is to delay the UPDATE to the post-fixing which fixes all the problematic data when all the tasks have finished.

In the following section, we propose an alternative approach for processing snowflaked dimensions without requiring the post-fixing.

4.3 Snowflaked Dimension Processing

In a snowflake schema, dimensions are normalized meaning that there are foreign key references and hierarchies between dimension tables. If we consider the dependencies when processing dimensions, the post-fixing step can be avoided. We therefore propose two methods particularly for snowflaked dimensions: *level-wise processing* and *hierarchy-wise processing*.

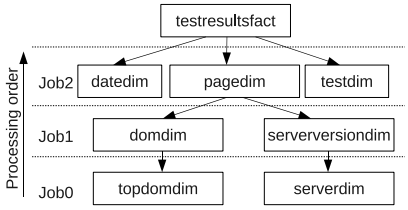

Figure 5: Level-wise processing

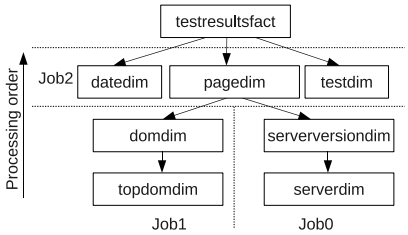

Figure 6: Hierarchy-wise processing

Level-wise processing This refers to processing snowflaked dimensions in an order from the leaves towards the root (the dimension table referred by the fact table is the root and a dimension table without a foreign key referencing other dimension tables is a leaf). The dimension tables with dependencies (i.e., with foreign key references) are processed in sequential jobs, e.g., *Job1* depends on *Job0*, and *Job2* depends on *Job1* in Figure 5. Each job processes independent dimension tables (without direct and indirect foreign key references) by parallel tasks, i.e., one dimension table is processed by one task. Therefore, in the level-wise processing of the running example, *Job0* first processes *topdomaindim* and *serverdim* in parallel, then *Job1* processes *domaindim* and *serverversiondim*, and finally *Job2* processes *pagedim*, *datedim* and *testdim*. It corresponds to the configuration *loadorder = [('topdomaindim', 'serverdim'), ('domaindim', 'serverversiondim'), ('pagedim', 'datedim', 'testdim')]*. With this order, a higher level dimension table (the referencing dimension table) is not processed until the lower level ones (the referenced dimension tables) have been processed and thus, the referential integrity can be ensured.

Hierarchy-wise processing This refers to processing a snowflaked dimension in a branch-wise fashion (see Figure 6). The root dimension, *pagedim*, derives two branches, each of which is defined as a separate snowflaked dimension, i.e., *domainsf = SnowflakedDimension([(domaindim, topdomaindim)])*, and *serverversionsf = SnowflakedDimension([(serverversiondim, serverdim)])*. They are processed by two parallel jobs, *Job0* and *Job1*, each of which processes in a sequential manner, i.e., *topdomaindim* followed by *domaindim* in *Job0* and *serverdim* followed by *serverversiondim* in *Job1*. The root dimension, *pagedim*, is not processed until the dimensions on its connected branches have been processed. It, together with *datedim* and *testdim*, is processed by the *Job2*.

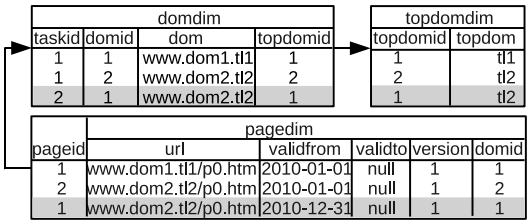

Figure 7: Before post-fixing

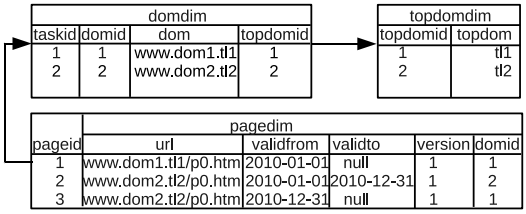

Figure 8: After post-fixing

4.4 Post-fixing

As discussed in Section 4.2, post-fixing is a remedy to fix problematic data in ODAT when all the tasks of the dimension processing have finished. Four situations require data post-fixing: 1) using a global ID generator which gives rise to duplicated values in the lookup attributes; 2) using private ID generators which produce duplicated key values; 3) processing snowflaked dimensions (and *not* using level-wise or hierarchy.wise processing) which leads to duplicated values in lookup and key attributes; and 4) processing slowly changing dimensions which results in SCD attributes taking improper values.

Example 1 (Post-fixing) *Consider two map/reduce tasks, task 1 and task 2, that process the* `page` *dimension which we here assume to be snowflaked. Each task uses a private ID generator. The root dimension,* `pagedim`*, is a type-2 SCD. Rows with the lookup attribute value url='www.dom2.tl2/p0.htm' are processed by both the tasks.*

Figure 7 depicts the resulting data in the dimension tables where the white rows were processed by task 1 and the grey rows were processed by task 2. Each row is labelled with the `taskid` *of the task that processed it. The problems include duplicate IDs in each dimension table and improper values in the SCD attributes,* `validfrom`*,* `validto`*, and* `version`*. The post-fixing program first fixes the* `topdomaindim` *such that rows with the same value for the lookup attribute (i.e.,* `url`*) are merged into one row with a single ID. Thus, the two rows with* `topdom` *= tl2 are merged into one row. The references to* `topdomaindim` *from* `domaindim` *are also updated to reference the correct (fixed) rows. In the same way,* `pagedim` *is updated to merge the two rows representing www.dom2.tl2. Finally,* `pagedim` *is updated. Here, the post-fixing also has to fix the values for the SCD attributes. The result is shown in Figure 8.*

Algorithm 2 post_fix(*dim*)

```
refdims ← The referenced dimensions of dim
for ref in refdims do
    itr ← post_fix(ref)
    for ((taskid, keyvalue), newkeyvalue) in itr do
        Update dim set dim.key = newkeyvalue where dim.taskid=taskid and dim.key=keyvalue
ret ← An empty list
Assign newkeyvalues to dim's keys and add ((taskid, keyvalue), newkeyvalue) to ret
if dim is not the root then
    Delete the duplicate rows, which have identical values in dim's lookup attributes
if dim is a type-2 SCD then
    Fix the values on SCD attributes, e.g., dates and version
return ret
```

The post-fixing invokes a recursive function (see Algorithm 2) to fix the problematic data in the order from the leaf dimension tables to the root dimension table. It comprises four steps: 1) assign new IDs to the rows with duplicate IDs; 2) update the foreign keys on the referencing dimension tables; 3) delete

duplicated rows which have identical values in the business key attributes and foreign key attributes; and 4) fix the values in the SCD attributes if applicable. In most cases, it is not needed to fix something in each of the steps for a dimension with problematic data. For example, if a global ID generator is employed, all rows will have different IDs (such that step 1 is not needed) but they may have duplicate values in the lookup attributes (such that step 3 is needed). ETLMR's implementation uses an embedded SQLite database for data management during the post-fixing. Thus, the task IDs are not stored in the target DW, but only internally in ETLMR.

4.5 Offline Dimension

In ODOT and ODAT, the map/reduce tasks interact with the DW's ("*online*") dimensions directly through database connections at run-time and the performance is affected by the outside DW DBMS and the database communication cost. To optimize, the *offline dimension* scheme is proposed, in which the tasks do not interact with the DW directly, but with distributed offline dimensions residing physically in all nodes. It has several characteristics and advantages. First, a dimension is partitioned into multiple smaller-sized sub-dimension, and small-sized dimensions can benefit dimension *lookup*s, especially for a data-intensive dimension such as *pagedim*. Second, high performance storage systems can be employed to persist dimension data. Dimensions are configured to be fully or partially cached in main memory to speedup the *lookup*s when processing facts. In addition, offline dimensions do not require direct communication with the DW and the overhead (from the network and the DBMS) is greatly reduced. ETLMR has offline dimension implementations for one dimension one task (*ODOT (offline)* for short) and *hybrid*, which are described in the following.

4.5.1 ODOT (offline)

Figure 9 depicts the run-time architecture when we use two map/reduce tasks to process data. The data for each dimension table is saved locally in its own store in the node that processes it or in the DFS (shown in the center of the Figure 9). The data for a dimension table is processed by one and only one reduce task, which resembles to the online ODOT, and does the following: 1) Select the values of the fields that are relevant to the dimension table in mappers; 2) Partition the map output based on the names of dimension table; 3) Process the data for dimension tables by using user-defined transformation functions in the reducers (a reducer only processes the data for one dimension table); 4) When all map/reduce tasks have finished, the data for all dimension tables are synchronized across the nodes if no DFS is installed (here, only the data files of the offline dimension stores are synchronized).

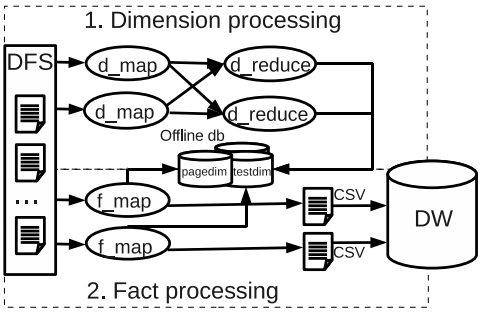

Figure 9: ODOT (offline)

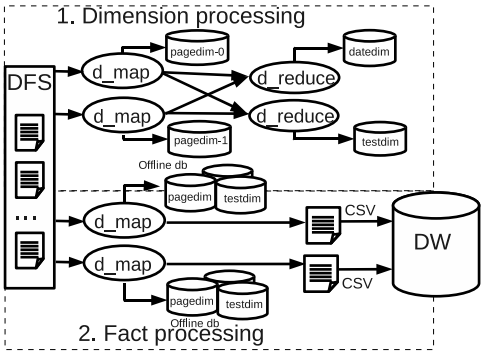

Figure 10: Hybrid

4.5.2 Hybrid

Hybrid combines the characteristics of ODOT and ODAT. In this approach, the dimensions are divided into two groups, the most data-intensive dimension and the other dimensions. The input data for the most data-intensive dimension table is partitioned based on the business keys, e.g., on the `url` of `pagedim`, and processed by all the map tasks (this is similar to ODAT), while for the other dimension tables, their data is processed in reducers, a reducer exclusively processing the data for one dimension table (this is similar to ODOT).

This corresponds to the following steps: 1) Choose the most data-intensive dimension and partition the input data sets, for example, on the business key values; 2) Process the chosen data-intensive dimension and select the required data for each of the other dimensions in the mappers; 3) Round-robin partition the map output; 3) Process the dimensions in the reducers (each is processed by one reducer); 4) When all the tasks have finished, synchronize all the processed dimensions across the nodes if no DFS is installed, but keep the partitioned data-intensive dimension in all nodes.

Example 2 (Hybrid) *Consider using two parallel tasks to process the dimension tables of the running example (see the upper part in Figure 10). The data for the most data-intensive dimension table,* `pagedim`*, is partitioned into a number of chunks based on the business key* `url`*. The chunks are processed by two mappers, each processing a chunk. This results to two offline stores for* `pagedim`*,* `pagedim-0` *and* `pagedim-1`*. However, the data for the other dimension tables is processed similarly to the offline ODOT, which results in the other two offline dimension stores,* `datedim` *and* `testdim`*, respectively.*

In the offline dimension scheme, the dimension data in the offline store is expected to reside in the nodes permanently and will not be loaded into the DW until this is explicitly requested.

5 Fact Processing

Fact processing is the second phase in ETLMR, which consists of looking up of dimension keys, doing aggregation on measures (if applicable), and loading the processed facts into the DW. Similarly to the dimension processing, the definitions and settings of fact tables are also declared in the configuration file. ETLMR provides the *BulkFactTable* class which supports bulk loading of facts to DW. For example, the fact table of the running example is defined as *testresultsfact=BulkFactTable(name='testresultsfact', keyrefs=['pageid', 'testid', 'dateid'], measures=['errors'], bulkloader=UDF_pgcopy, bulksize=5000000)*. The parameters are the fact table name, a list of the keys referencing dimension tables, a list of measures, the bulk loader function, and the size of the bulks to load. The bulk loader is a UDF which can be configured to satisfy different types of DBMSs.

Algorithm 3 shows the pseudo code for processing facts.

Algorithm 3 $process\_fact(row)$

Require: A *row* from the input data and the *config*

```
1: facttbls ← the fact tables defined in config
2: for facttbl in facttbls do
3:     dims ← the dimensions referenced by facttbl
4:     for dim in dims do
5:         row[dim.key] ← dim.lookup(row)
6:     rowhandlers ← facttbl.rowhandlers
7:     for handler in rowhandlers do
8:         handler(row)
9:     facttbl.insert(row)
```

The function can be used as the map function or as the reduce function. If no aggregations (such as *sum*, *average*, or *count*) are required, the function is configured to be the map function and the reduce step is omitted for better performance. If aggregations are required, the function is configured to be the reduce function since the aggregations must be computed from all the data. This approach is flexible and good for performance. Line 1 retrieves the fact table definitions in the configuration file and they are then processed sequentially in line 2–8. The processing consists of two major operations: 1) look up the keys from the referenced dimension tables (line 3–5), and 2) process the fact data by the *rowhandlers*, which are user-defined transformation functions used for data type conversions, calculating measures, etc. (line 6–8). Line 9 invokes the insert function to insert the fact data into the DW. The processed fact data is not inserted into the fact table directly, but instead added into a configurably-sized buffer where it is kept temporarily. When a buffer becomes full, its data is loaded into the DW by using the bulk load. Each map/reduce task has a separate buffer and bulk loader such that tasks can do bulk loading in parallel.

6 Implementation and Optimization

ETLMR is designed to achieve plug-in like functionality to facilitate the integration with Python-supporting MapReduce frameworks. In this section, we introduce the ETL programming framework, *pygrametl* which is used to implement ETLMR. Further, we give an overview of MapReduce frameworks with a special focus on Disco [2] which is our chosen MapReduce platform. The integration and optimization techniques employed are also described.

6.1 pygrametl

`pygrametl` was implemented in our previous work [17]. It has a number of characteristics and functionalities. First, it is a code-based ETL framework with very high efficiency in development due to its simplicity and use of Python. Second, it supports processing of dimensions in both star schema and snowflake schema and it, further, provides direct support of slowly changing dimensions. Regardless of the dimension type, the ETL implementation is very concise and convenient. It uses an object to represent a dimension. Only a single method call such as `dimobj.insert(row)` is needed to insert new data. In this method call, all details, such as key assignment and SQL generation, are transparent to users. `pygrametl` supports the most commonly used operations on a dimension, such as `lookup`, `insert`, `ensure`. Third, `pygrametl` supports caching, batch insertion, and bulk-load. In the implementation of ETLMR, most functions offered by `pygrametl` can be re-used, but some of them need to be extended or modied to support the MapReduce operations.

6.2 MapReduce Frameworks

There are many open source and commercial MapReduce frameworks available. The most popular one is Apache Hadoop [4], which is implemented in Java and includes a distributed file system (HDFS). Hadoop is embraced by a variety of academic and industrial users, including Amazon, Yahoo!, Facebook, and many others [1]. Google's MapReduce implementation is extensively used inside the company, but is not available for public. Apart from them, many companies and research units develop their own MapReduce frameworks for their particular needs, such as [14, 21, 11].

For ETLMR, we select the open source framework, Disco [2], as the MapReduce platform. Disco is developed by Nokia using the Erlang and Python programming languages. Disco is chosen for the following reasons. First, Disco's use of Python, facilitates rapid scripting for distributed data processing programs, e.g., a complicated program or algorithm can be expressed in tens of lines of code. This feature is consistent with our aim of providing users a simple and easy means of implementing a parallel ETL. Second, Disco

achieves a high degree of flexibility by providing many customizable MapReduce programming interfaces. These make the integration of ETLMR very convenient by having a ”plug-in” like effect. Third, unlike the alternatives, it provides direct support for the distributed programs written in Python. Some MapReduce frameworks, implemented in other programming languages, also claim to support Python programs, e.g., Hadoop, but they require bridging middle-ware and are, thus, not implementation-friendly.

6.3 Integration with Disco

Disco's architecture shares clear similarities to the Google and Hadoop MapReduce architectures, in which intermediate results are stored as local files and accessed by appropriate reduce tasks. However, the current Disco (version 0.2.4) does not include a built-in distributed file system (DFS), but it supports any POSIX-compatible DFS such as GlusterFS. If no DFS is installed, the input files are required to be distributed initially to each node so that they can be read locally [2]. The integration with Disco is through functional assignment statements, or function parameters, where the parallel ETL functions work as function objects passed directly to the corresponding interfaces from Disco.

We present the architecture of ETLMR on Disco in Figure 11, where all ETLMR processes are running in parallel on many processors of clustered computers (or nodes). This architecture is capable of shortening the time of data extract-transform-load by scaling up to any size of the nodes in the cluster. Disco has a master/worker architecture with one master and many workers. The master is responsible for scheduling the jobs' components(tasks) to run on the workers, assigning partitioned data sets to the workers, tracking the status, and monitoring the health of the workers. Disco uses HTTP as the communication protocol to distribute jobs. When the master receives ETL jobs, it puts them to a queue and distributes them to the available workers. In each node, there is a worker supervisor started by the master which is responsible for spawning and monitoring all tasks on that particular node. When a worker receives a task, it runs this task exclusively in a processor of this node, processes the input data, and saves the processed data to the DW.

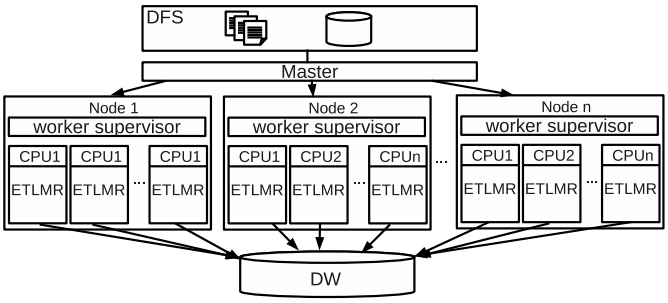

Figure 11: Parallel ETL overview

The master-workers architecture has a highly fault-tolerant mechanism. It achieves reliability by distributing the input files to the nodes in the cluster. Each worker reports to the master periodically with the completed tasks and their status. If a worker falls silent for longer than an interval, the master will record this worker as dead and send out the node's assigned tasks to the other workers.

6.4 Optimizations

The map readers that come with Disco (version 0.2.4) only support read operations on data files. The data sets from different storage systems must, therefore, be prepared into data files. Each file is provided with a unique address for map readers, such as a URL, file path, or distributed file path. In many cases, however, a

data source may be a database, or an indexed file structure, so that we can make use of its indices for efficient filtering, i.e., rather than returning all the data, only the columns needed for processing, or a sub-range of the data, are selected. If the data sets are pre-split, such as several data files from heterogeneous systems, the split parts are directly processed and combined in MapReduce. Different map readers are implemented in ETLMR for reading data from different storage systems, such as the DBMS reader supporting user-defined SQL statements and text file reader. If the data sets are not split, such as a big data file, Dean and Ghemwat [8] suggest utilizing many MapReduce processes to run complete passes over the data sets, and process the subsets of the data. Accordingly, we implement such a map reader (see Program 1). It supports reading data from a single data source, but does not require the data sets to be split before being processed. In addition, we implement the offline dimension scheme by using the Python `shelve` package [3], in which main-memory database systems, such as `bsdbdb`, can be configured to persist dimension data. At run-time, the dimension data is fully or partially (by *least recently used* (LRU) mechanism) kept in main memory such that the *lookup* operation can be done efficiently.

Program 1 Map reader function

```
def map_reader(content, bkey, thispartition=this_partition()):
  while True:
    line = content.next()
    if not line:
        break
    if (hash(line[bkey])%Task.num_partitions)==thispartition:
        yield line
```

7 Experimental Evaluation

In this section, we measure the performance improvements achieved by the proposed methods. Further, we evaluate the system scalability on various sizes of tasks and data sets and compare with other business intelligence tools using MapReduce.

7.1 Experimental Setup

All experiments are conducted on a cluster of 6 nodes connected through a gigabit switch and each having an Intel(R) Xeon(R) CPU X3220 2.4GHz with 4 cores, 4 GB RAM, and a SATA hard disk (350 GB, 3 GB/s, 16 MB Cache and 7200 RPM). All nodes are running the Linux 2.6.32 kernel with Disco 0.2.4, Python 2.6, and ETLMR installed. The GlusterFS DFS is set up for the cluster. PostgreSQL 8.3 is used for the DW DBMS and is installed on one of the nodes. One node serves as the master and the others as the workers. Each worker runs 4 parallel map/reduce tasks, i.e., in total 20 parallel tasks run. The time for bulk loading is not measured as the way data is bulk loaded into a database is an implementation choice which is independent of and outside the control of the ETL framework. To include the time for bulk loading would thus clutter the results. We note that bulk loading can be parallelized using off-the-shelf functionality.

7.2 Test Data

We continue to use the running example. We use a data generator to generate the test data for each experiment. In line with Jean and Ghemawat's assumption that MapReduce usually operates on numerous small files rather than a single, large, merged file [8], the test data sets are partitioned and saved into a set of files. These files provide the input for the dimension and fact processing phases. We generate two data

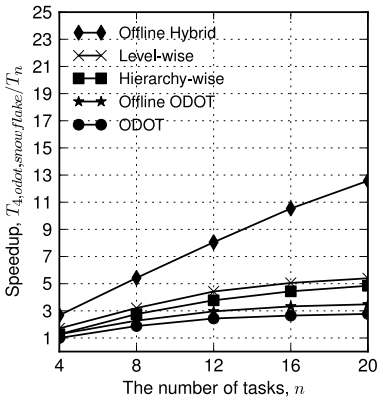

Figure 12: Parallel ETL for snowflake schema, 20 GB

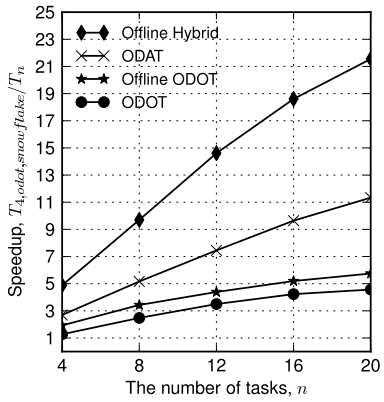

Figure 13: Parallel ETL for star schema, 20 GB

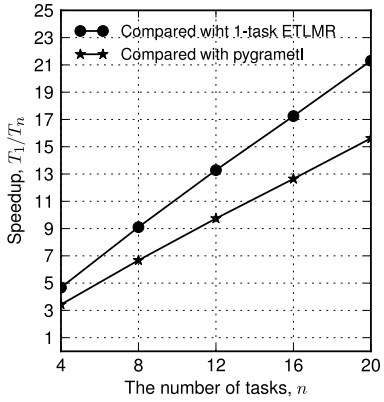

Figure 14: Speedup with increasing tasks, 80 GB

sets, *bigdim* and *smalldim* which differ in the size of the *page* dimension. In particular, 80 GB *bigdim* data results in 10.6 GB fact data (193,961,068 rows) and 6.2 GB *page* dimension data (13,918,502 rows) in the DW while 80 GB *smalldim* data results in 12.2 GB (222,253,124 rows) fact data and 54 MB *page* dimension data (193,460 rows) in the DW. Both data sets produce 32 KB *test* (1,000 rows) and 16 KB *date* dimension data (1,254 rows).

7.3 Scalability of Proposed Processing Methods

In this experiment, we compare the scalability and performance of the different ETLMR processing methods. We use a fixed-size *bigdim* data set (20 GB), scale the number of parallel tasks from 4 to 20, and measure the total elapsed time from start to finish. The results for a snowflake schema and a star schema are shown in Figure 12 and Figure 13, respectively. The graphs show the *speedup*, computed by $T_{4,odot,snowflake}/T_n$ where $T_{4,odot,snowflake}$ is the processing time for *ODOT* using 4 tasks in a snowflake schema and T_n is the processing time when using n tasks for the given processing method.

We see that the overall time used for the star schema is less than for the snowflake schema. This is because the snowflake schema has dimension dependencies and hierarchies which require more (level-wise) processing. We also see that the offline hybrid scales the best and achieves almost linear speedup. The ODAT in Figure 13 behaves similarly. This is because the dimensions and facts in offline hybrid and ODAT are processed by all tasks which results in good balancing and scalability. In comparison, ODOT, offline ODOT, level-wise, and hierarchy-wise do not scale as well as ODAT and hybrid since only a limited number of tasks are utilized to process dimensions (a dimension is only processed in a single task). The offline dimension scheme variants outperform the corresponding online ones, e.g., offline ODOT vs. ODOT. This is caused by 1) using a high performance storage system to save dimensions on all nodes and provide in-memory lookup; 2) The data-intensive dimension, *pagedim*, is partitioned into smaller chunks which also benefits the lookup; 3) Unlike the online dimension scheme, the offline dimension scheme does not communicate directly with the DW and this reduces the communication cost considerably. Finally, the results show the relative efficiency for the optimized methods which are much faster than the baseline ODOT.

7.4 System Scalability

In this experiment, we evaluate the scalability of ETLMR by varying the number of tasks and the size of the data sets. We select the hybrid processing method, use the offline dimension scheme, and conduct the testing on a star schema, as this method not only can process data among all the tasks (unlike ODOT in which only a limited number of tasks are used), but also showed the best scalability in the previous experiment. In the dimension processing phase, the mappers are responsible for processing the data-intensive dimension *pagedim* while the reducers are responsible for the other two dimensions, *datedim* and *testdim*, each using only a single reducer. In the fact processing phase, no reducer is used as no aggregation operations are required.

We first do two tests to get comparison baselines by using one task (named *1-task ETLMR*) and (plain, non-MapReduce) *pygrametl*, respectively. Here, pygrametl also employs 2-phase processing, i.e., the dimension processing is done before the fact processing. The tests are done on the same machine with a single CPU (all cores but one are disabled). The tests process 80 GB *bigdim* data. We compute the speedups by using T_1/T_n where T_1 represents the elapsed time for 1-task ETLMR or for pygrametl, and T_n the time for ETLMR using n tasks. Figure 14 shows that ETLMR achieves a nearly linear speedup in the number of tasks when compared to 1-task ETLMR (the line on the top). When compared to pygrametl, ETLMR has a nearly linear speedup (the lower line) as well, but the speedup is a little lower. This is because the baseline, 1-task ETLMR, has a greater value due to the overhead from the MapReduce framework.

To learn more about the details of the speedup, we break down the execution time of the slowest task by reference to the MapReduce steps when using the two data sets (see Table 2). As the time for dimension processing is very small for *smalldim* data, e.g., 1.5 min for 4 tasks and less than 1 min for the others, only its fact processing time is shown. When the *bigdim* data is used, we can see that partitioning input data, map, partitioning map output (dims), and combination (dims) dominate the execution. More specifically, partitioning input data and map (see the *Part.Input* and *Map func.* columns) achieve a nearly linear speedup in the two phases. In the dimension processing, the map output is partitioned and combined for the two dimensions, *datedim* and *testdim*. Also here, we see a nearly linear speedup (see the *Part.* and *Comb.* columns). As the combined data of each is only processed by a single reducer, the time spent on reducing is proportional to the size of data. However, the time becomes very small since the data has been merged in combiners (see *Red. func.* column). The cost of post-fixing after dimension processing is not listed in the table since it is not required in this case where a global key generator is employed to create dimension IDs and the input data is partitioned by the business key of the SCD *pagedim* (see section 4.4).

Table 2: Execution time distribution, 80 GB (min.)

Testing data	Phase	Task Num	Part. Input	Map func.	Part.	Comb.	Red. func.	Others	Total
bigdim data (results in 10.6GB facts)	dims	4	47.43	178.97	8.56	24.57	1.32	0.1	260.95
		8	25.58	90.98	4.84	12.97	1.18	0.1	135.65
		12	17.21	60.86	3.24	8.57	1.41	0.1	91.39
		16	12.65	47.38	2.50	6.54	1.56	0.1	70.73
		20	10.19	36.41	1.99	5.21	1.32	0.1	55.22
	facts	4	47.20	183.24	0.0	0.0	0.0	0.1	230.44
		8	24.32	92.48	0.0	0.0	0.0	0.1	116.80
		12	16.13	65.50	0.0	0.0	0.0	0.1	81.63
		16	12.12	51.40	0.0	0.0	0.0	0.1	63.52
		20	9.74	40.92	0.0	0.0	0.0	0.1	50.66
smalldim data (results in 12.2GB facts)	facts	4	49.85	211.20	0.0	0.0	0.0	0.1	261.15
		8	25.23	106.20	0.0	0.0	0.0	0.1	131.53
		12	17.05	71.21	0.0	0.0	0.0	0.1	88.36
		16	12.70	53.23	0.0	0.0	0.0	0.1	66.03
		20	10.04	42.44	0.0	0.0	0.0	0.1	52.58

In the fact processing, the reduce function needs no execution time as there is no reducer. The time for all the other parts, including map and reduce initialization, map output partitioning, writing and reading intermediate files, and network traffic, is relatively small, but it does not necessarily decrease linearly when more tasks are added (*Others* column). To summarize (see *Total* column), ETLMR achieves a nearly linear speedup when the parallelism is scaled up, i.e., the execution time of 8 tasks is nearly half that of 4 tasks, and the execution time of 16 tasks is nearly half that of 8 tasks.

Table 2 shows some overhead in the data reading (see *Part. input* column) which uses nearly 20% of the total time. We now compare the two approaches for reading data, namely when it is split into several smaller files ("pre-split") and when it is available from a single big file ("no-split"). We use *bigdim* data and only process dimensions.

We express the performance improvement by the speedup, $T_{4,no\text{-}split}/T_n$, where $T_{4,no\text{-}split}$ is the time taken to read data and process dimensions on the no-split data using 4 tasks (on 1 node), and T_n is the time when using n tasks. As illustrated in Figure 15, for no-split, the time taken to read data remains constant with increasing number of tasks as all read the same data sets. When the data is pre-split, the time taken to read data and process dimensions scales much better than for no-split since a smaller sized data set is processed by each task. In addition, pre-split is inherently faster – by a factor of 3 – than no-split. The slight sub-linear scaling is seen because the system management overhead (e.g., the time spent on communication, adjusting, and maintaining the overall system) increases with the growing number of tasks. However, we can conclude that pre-split is by far the best option.

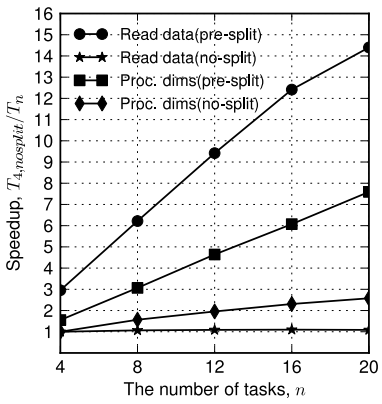

Figure 15: Speedup of pre-split and no split, 20 GB

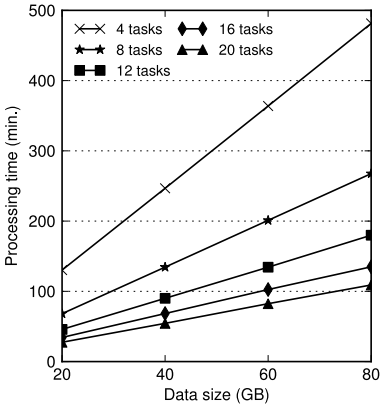

Figure 16: Proc. time when scaling up bigdim data

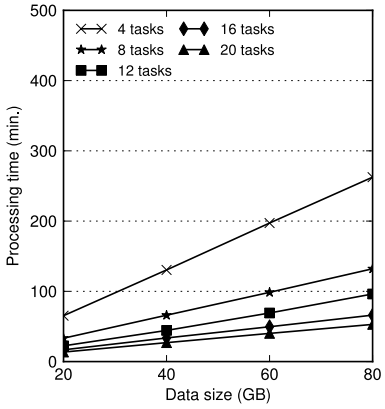

Figure 17: Proc. time when scaling up smalldim data

We now proceed to another experiment where we for a given number of tasks size up the data sets from 20 to 80 GB and measure the elapsed processing time. Figure 16 and Figure 17 show the results for the *bigdim* and *smalldim* data sets, respectively. It can be seen that ETLMR scales linearly in the size of the data sets.

7.5 Comparison with Other Data Warehousing Tools

There are some MapReduce data warehousing tools available, including Hive [18, 19], Pig [12] and Pentaho Data Integration (PDI) [5]. Hive and Pig both offer data storage on the Hadoop distributed file system (HDFS) and scripting languages which have some limited ETL abilities. They are both more like a DBMS instead of a full-blown ETL tool. Due to the limited ETL features, they cannot process an SCD which requires UPDATEs, something Hive and Pig do not support. It is possible to process star and snowflake schemas, but it is complex and verbose. To load data into a *simplified* version of our running example (with

no SCDs) require 23 statements in Pig and 40 statements in Hive. In ETLMR – which in contrast to Pig and Hive is dimensional – only 14 statements are required. ETLMR can also support SCDs with the *same* number of statements, while this would be virtually impossible to do in Pig and Hive. The details of the comparison are seen in Appendix A.

PDI is an ETL tool and provides Hadoop support in its 4.1 GA version. However, there are still many limitations with this version. For example, it only allows to set a limited number of parameters in the job executor, customized combiner and mapper-only jobs are not supported, and the transformation components are not fully supported in Hadoop. We only succeeded in making an ETL flow for the simplest star schema, but still with some compromises. For example, a workaround is employed to load the processed dimension data into the DW as PDI's *table output* component repeatedly opens and closes database connections in Hadoop such that performance suffers.

In the following, we compare how PDI and ETLMR perform when they process the star schema (with *page* as a normal dimension, not an SCD) of the running example. To make the comparison neutral, the time for loading the data into the DW or the HDFS is not measured, and the dimension lookup cache is enabled in PDI to achieve a similar effect of ETLMR using offline dimensions. Hadoop is configured to run 4 parallel task trackers in maximum on each node, and scaled by adding nodes horizontally. The task tracker JVM option is set to be -Xmx256M while the other settings are left to the default.

Table 3 shows the time spent on processing 80 GB *smalldim* data when scaling up the number of tasks. As shown, ETLMR is significantly faster than PDI for Hadoop in processing the data. Several reasons are found for the differences. First, compared with ETLMR, the PDI job has one more step (the reducer) in the fact processing as its job executor does not support a mapper-only job. Second, by default the data in Hadoop is split which results in many tasks, i.e., 1192 tasks for the fact data. Thus, longer initialization time is observed. Further, some transformation components are observed to run with low efficiency in Hadoop, e.g., the components to remove duplicate rows and to apply JavaScript.

Table 3: Time for processing star schema (no SCD), 80 GB *smalldim* data set, (min.)

Tasks	4	8	12	16	20
ETLMR	246.7	124.4	83.1	63.8	46.6
PDI	975.2	469.7	317.8	232.5	199.7

8 Related Work

We now compare ETLMR to other parallel data processing systems using MapReduce, and parallel DBMSs. In addition, we study the current status of parallel ETL tools. MapReduce is a framework well suited for large-scale data processing on clustered computers. However, it has been criticized for being too low-level, rigid, hard to maintain and reuse [12, 18]. In recent years, an increasing number of parallel data processing systems and languages built on the top of MapReduce have appeared. For example, besides Hive and Pig (discussed in Section 7.5), Chaiken et al. present the SQL-like language SCOPE [6] on top of Microsoft's Cosmos MapReduce and distributed file system. Friedman et al. introduce SQL/MapReduce [10], a user-defined function (UDF) framework for parallel computation of procedural functions on massively-parallel RDBMSs. These systems or languages vary in the implementations and functionalities provided, but overall they give good improvements to MapReduce, such as high-level languages, user interfaces, schemas, and catalogs. They process data by using query languages, or UDFs embedded in the query languages, and execute them on MapReduce. However, they do not offer direct constructs for processing star schemas, snowflaked dimensions, and slowly changing dimensions. In contrast, ETLMR runs separate ETL processes on a MapReduce framework to achieve parallelization and ETLMR directly supports ETL constructs for these schemas.

Another well-known distributed computing system is the parallel DBMS which first appeared two decades ago. Today, there are many parallel DBMSs, e.g., Teradata, DB2, Objectivity/DB, Vertica, etc. The principal difference between parallel DBMSs and MapReduce is that parallel DBMSs run long pipelined queries instead of small independent tasks as in MapReduce. The database research community has recently compared the two classes of systems. Pavlo et al. [13], and Stonebraker et al. [15] conduct benchmarks and compare the open source MapReduce implementation Hadoop with two parallel DBMSs (a row-based and a column-based) in large-scale data analysis. The results demonstrate that parallel DBMSs are significantly faster than Hadoop, but they diverge in the effort needed to tune the two classes of systems. Dean et al. [8] argue that there are mistaken assumptions about MapReduce in the comparison papers and claim that MapReduce is highly effective and efficient for large-scale fault-tolerance data analysis. They agree that MapReduce excels at complex data analysis, while parallel DBMSs excel at efficient queries on large data sets [15].

In recent years, ETL technologies have started to support parallel processing. Informatica PowerCenter provides a thread-based architecture to execute parallel ETL sessions. Informatica has also released PowerCenter Cloud Edition (PCE) in 2009 which, however, only runs on a specific platform and DBMS. Oracle Warehouse Builder (OWB) supports pipeline processing and multiple processes running in parallel. Microsoft SQL Server Integration Services (SSIS) achieves parallelization by running multiple threads, multiple tasks, or multiple instances of a SSIS package. IBM InfoSphere DataStage offers a process-based parallel architecture. In the thread-based approach, the threads are derived from a single program, and run on a single (expensive) SMP server, while in the process-based approach, ETL processes are replicated to run on clustered MPP or NUMA servers. ETLMR differs from the above by being open source and based on MapReduce with the inherent advantages of multi-platform support, scalability on commodity clustered computers, light-weight operation, fault tolerance, etc. ETLMR is also unique in being able to scale automatically to more nodes (with no changes to the ETL flow itself, only to a configuration parameter) while at the same time providing automatic data synchronization across nodes even for complex structures like snowflaked dimensions and SCDs. We note that the licenses of the commercial ETL packages prevent us from presenting comparative experimental results.

9 Conclusion and Future Work

As business intelligence deals with continuously increasing amounts of data, there is an increasing need for ever-faster ETL processing. In this report, we have presented ETLMR which builds on MapReduce to parallelize ETL processes on commodity computers. ETLMR contains a number of novel contributions. It supports high-level ETL-specific dimensional constructs for processing both star schemas and snowflake schemas, SCDs, and data-intensive dimensions. Due to its use of MapReduce, it can automatically scale to more nodes (without modifications to the ETL flow) while it at the same time provides automatic data synchronization across nodes (even for complex dimension structures like snowflakes and SCDs). Apart from scalability, MapReduce also gives ETLMR a high fault-tolerance. Further, ETLMR is open source, light-weight, and easy to use with a single configuration file setting all run-time parameters. The results of extensive experiments show that ETLMR has good scalability and compares favourably with other MapReduce data warehousing tools.

ETLMR comprises two data processing phases, dimension and fact processing. For dimension processing, the report proposed a number of dimension management schemes and processing methods in order to achieve good performance and load balancing. The online dimension scheme directly interacts with the target DW and employs several dimension specific methods to process data, including *ODOT*, *ODAT*, and *level-wise* and *hierarchy-wise* processing for snowflaked dimensions. The offline dimension scheme employs high-performance storage systems to store dimensions distributedly on each node. The methods, *ODOT* and *hybrid* allow better scalability and performance. In the fact processing phase, bulk-load is used to improve the loading performance.

Currently, we have integrated ETLMR with the MapReduce framework, Disco. In the future, we intend to port ETLMR to Hadoop and explore a wider variety of data storage options. In addition, we intend to implement dynamic partitioning which automatically adjusts the parallel execution in response to additions/removals of nodes from the cluster, and automatic load balancing which dynamically distributes jobs across available nodes based on CPU usage, memory, capacity and job size through automatic node detection and algorithm resource allocation.

10 Acknowledgments

This work was in part supported by Daisy Innovation and the European Regional Development Fund and the eGovMon project co-funded by the Research Council of Norway under the VERDIKT program (project no. Verdikt 183392/S10)

References

[1] "Applications and organizations using Hadoop". Available from `http://wiki.apache.org/hadoop/PoweredBy` as of 2011-08-06.

[2] "Disco project". Available from `http://discoproject.org/` as of 2011-08-06.

[3] "Shelve - Python object persistence". Available from `http://docs.python.org/library/shelve.html` as of 2011-08-06.

[4] "The Apache Hadoop Project". Available from `http://hadoop.apache.org` as of 2011-08-06.

[5] `www.pentaho.com` as of 2011-08-06.

[6] 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". In *PVLDB*, 1(2): 1265–1276, 2008.

[7] J. Dittrich, J. A. Quiane-Ruiz, A. Jindal, Y. Kargin, V. Setty and J. Schad, "Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)". In *PVLDB*, 3(1), 2010.

[8] J. Dean and S. Ghemawat, "MapReduce: A Flexible Data Processing Tool". In *CACM*, 53(1): 72–77, 2010.

[9] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters". In *Proc. of OSDI*, pp. 137–150, 2004.

[10] E. Friedman, P. Pawlowski, and J. Cieslewicz, "SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions". In *PVLDB*, 2(2): 1402–1413, 2009.

[11] G. Kovoor, J. Singer and M. Lujan, "Building a Java MapReduce Framework for Multi-core Architectures". In *Proc. of MULTIPROG*, 2010.

[12] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins, "Pig Latin: A Not-so-foreign Language for Data Processing". In *Proc. of SIGMOD*, pp. 1099–1110, 2008.

[13] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden and M. Stonebraker, "A Comparison of Approaches to Large-scale Data Analysis". In *Proc. of SIGMOD*, pp. 165–178, 2009.

[14] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems". In *Proc. of HPCA*, pp. 13–24, 2007.

[15] M. Stonebraker, D. Abadi, D. DeWitt, S. Madden, E. Paulson, A. Pavlo and A. Rasin, "MapReduce and Parallel DBMSs: friends or foes?". In *CACM*, 53(1): 64–71, 2010.

[16] C. Thomsen and T. Pedersen, "Building a Web Warehouse for Accessibility Data". In *Proc. of DOLAP*, 2009.

[17] C. Thomsen and T. Pedersen, "pygrametl: A Powerful Programming Framework for Extract-Transform-Load Programmers". In *Proc. of DOLAP*, 2009.

[18] A. Thusoo, J. 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". In *PVLDB*, 2(2): 1626–1629, 2009.

[19] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu and R. Murthy, "Hive-A Petabyte Scale Data Warehouse Using Hadoop". In *Proc. of ICDE*, pp. 996–1005, 2010.

[20] P. Vassiliadis and A. Simitsis, "Near Real Time ETL". In *J. of New Trends in Data Warehousing and Data Analysis*, pp. 1–31, 2008.

[21] R. Yoo, A. Romano and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-memory System". In *Proc. of IISWC*, pp. 198–207, 2009.

Appendix

A Comparison with other MapReduce tools for BI

ETL tools are used for extracting, transforming and loading data into a data warehouse. While this in some cases can be done by only using DBMS software, the typical scenario is that a specialized, stand-alone ETL tool is used. The reason is that ETL tools allow users to do things that are difficult to do with DBMS software such as reading different file formats, copying and writing files in different formats, sending email notifications, connecting to web services, etc. The large amount of different ETL tools available on the market in itself proves the industry-need for stand-alone ETL tools. In the following we compare ETLMR to Hive [18, 19] and Pig [12] which are generic MapReduce-based data warehouse systems for storing data and analysis. This is thus somewhat similar to comparing ETL tools and DBMSs, but for completeness we include the comparisons here. Unlike Hive and Pig, ETLMR does not have its own data storage (note that the offline dimension store is only for speedup purpose), but is an ETL tool suitable for processing large scale data in parallel. Hive and Pig share large similarity, such as using Hadoop MapReduce, using Hadoop distributed file system (HDFS) as their data storage, integrating a command line user interface, implementing a query language, being able to do some ETL data analysis, and others. In the following, we compare their ETL features with ETLMR.

Table 4 summarizes the comparison. First, each system has a user interface. Hive provides an SQL-like language HiveQL and a shell, Pig provides a scripting language Pig Latin and a shell, and ETLMR provides a configuration file to declare dimensions, facts, UDFs, and other run-time parameters. Unlike Hive and Pig which require users to write data processing scripts explicitly, ETLMR is intrinsically an ETL tool which implements ETL process within the framework. The advantage is that users do not have to learn the details of each ETL step, and are able to craft a parallel ETL program even without much ETL knowledge. Second, each system supports UDFs. In Hive and Pig, an external function or user customized code for a specific task can be implemented as a UDF, and integrated into their own language,

Table 4: The comparison of ETL features

Feature	ETLMR	HIVE	PIG
User Interface	Configuration file	Shell/HiveQL/Web JDBC/ODBC	Shell/Pig Latin
ETL knowledge required	Low	High	High
User Defined Functions (UDF)	Yes	Yes	Yes
Filter/Aggregation/Join	Yes	Yes	Yes
Star Schema	Yes (explicit)	By handcode (implicit)	By handcode (implicit)
Snowflake Schema	Yes (explicit)	By handcode (implicit)	By handcode (implicit)
Slowly Changing Dimension (SCD)	Yes (explicit)	No	No
ETL details to users	Transparent	Fine-level	Fine-level

e.g., functions for serialization/deserialization data. In ETLMR, UDFs are a number of *rowhandler*s (see Section 4 and 5) integrated into *map* and *reduce* functions. These UDFs are defined for data filtering, transformation, extraction, name mapping. ETLMR also provides other ETL primitive constructs, such as hash join or merge join between data sources, and the aggregation of facts by plugable function, i.e., used as the reduce function (see Section 5). In contrast, Hive and Pig achieve the functionlity of ETL constructs through a sequence of user-written statements, which are later translated into execution plans, and executed on Hadoop. Third, as ETLMR is a specialized tool developed for fast implementation of parallel ETLs, it explicitly supports the ETLs for processing different schemas, including star schema, snowflake schema and SCDs, and very large dimensions. Therefore, the implementation of a parallel ETL program is very concise and intuitive for these schemas, i.e., only fact tables, their referenced dimensions and *rowhandlers* if necessary must be declared in the configuration file. Although Hive and Pig both are able to process star and snowflake schemas technically, implementing an ETL, even the most simple star schema, is not a trivial task as users have to dissect the ETL, write the processing statements for each ETL step, implement UDFs, and do numerous testing to make them correct. Moreover, as the HiveQL and Pig Latin lack UPDATE and DELETE operations, they are not able to process SCDs, which require UPDATE operation on a dimension's valid date or/and version. Fourth, ETLMR is an alternative to traditional ETL tools but offer better scalability. In contrast, Hive and Pig are obviously not optimal for the situation, where an external DW is used.

In order to make the comparison more intuitive, we create ETL programs for processing the snowflaked schema for the running example (see Figure 5) in each of the tools. The scripts are shown in Appendix A.1, A.2 and A.3, respectively. In each script, the UDFs are not shown, but indicated by self-explaining names (starting with *UDF_*). As described, the implementation of ETLMR only includes a number of declarations in the configuration file, such as dimensions, fact tables, data sources and other parameters, and a single line to start the program. All the ETL details are transparent to users. In contrast, the scripts of Hive and Pig include the finest-level details of the ETL. In ETLMR, as only declarations are required, its script is more concise and readable, e.g., only containing 14 statements for processing the snowflaked schema (a statement may span several lines by "\" in Python). In contrast, the implementations consist of 23 and 40 statements by using HiveQL and Pig Latin, respectively (each statement ends with ";"). In addition, although we have a clear picture of the ETL processing of this schema, we still spent several hours to write scripts for Hive and Pig (the time of implementing UDFs is not included), and test each step. In contrast, it is of high efficiency to script in ETLMR.

A.1 ETLMR

```
# The configuration file, config.py
# Declare all the dimensions:
datedim = Dimension(name='date',key='dateid',attributes=['date','day','month',\
          'year','week','weekyear'],lookupatts=['date'])
testdim = Dimension(name='test',key='testid',defaultidvalue=-1,\
          attributes=['testname'],lookupatts=['testname'])
pagedim = SlowlyChangingDimension(name='page',key='pageid',lookupatts=['url'], attributes=['url',\
         'size', 'validfrom','validto','version','domain', 'serverversion'], versionatt='version',\
          srcdateatt='lastmoddate',fromatt='validfrom',toatt='validto',srcdateatt='lastmoddate')
topdomaindim = Dimension(name='topdomain',key='topdomainid',\
          attributes=['topdomain'],lookupatts=['topdomain'])
domaindim = Dimension( name='domain',key='domainid', attributes=['domain', 'topdomainid'],\
          lookupatts=['domain'])
serverdim = Dimension(name='server',key='serverid',attributes=['server'],lookupatts=['server'])
serverversiondim = Dimension(name='serverversion',key='serverversionid',attributes=['serverversion',\
          'serverid'],lookupatts=['serverversion'],refdims=[serverdim])

# Define the snowflaked referencing-ship:
pagesf = [(pagedim, [serverversiondim, domaindim]),(serverversiondim, serverdim),\
         (domaindim, topdomaindim)]

# Declare the facts:
testresultsfact = BulkFactTable(name='testresults',keyrefs=['pageid', 'testid', 'dateid'],\
        measures=['errors'], bulkloader=UDF_pgcopy,bulksize=5000000)

# Define the settings of dimensions, including  data source schema, UDFs,
# dimension load order, and the referenced dimensions of fact:
dims ={pagedim: {'srcfields':('url','serverversion','domain','size','lastmoddate'),\
      'rowhandlers':(UDF_extractdomain, UDF_extractserver)}, datedim: {'srcfields':('downloaddate',),\
      'rowhandlers':(UDF_explodedate,) }, testdim:{'srcfields':('test',),'rowhandlers':(, )},}

# Define the processing order of snowflaked dimesions:
loadorder = [('topdomaindim', 'serverdim'),('domaindim', 'serverversiondim'),\
      ('pagedim', 'datedim', 'testdim')]

# Define the settings of facts:
facts = {testresultsfact:{'refdims':(pagedim, datedim, testdim),'rowhandlers':(, )},}

# Define the input data:
inputdata = ['dfs://localhost/TestResults0.csv', 'dfs://localhost/TestResults1.csv']

# The main ETLMR program: paralleletl.py
# Start the ETLMR program:
ETLMR.load('localhost',inputdata,required_modules=[('config','config.py'),],nr_maps=4,nr_reduces=4)
```

A.2 HIVE

```
-- Copy the data sources from local file system to HDFS:
hadoop fs -copyFromLocal /tmp/DownloadLog.csv /user/test;
hadoop fs -copyFromLocal /tmp/TestResults.csv /user/test;

-- Create staging tables for the data sources:
CREATE EXTERNAL TABLE downloadlog(localfile STRING, url STRING, serverversion STRING,
size INT, downloaddate STRING,lastmoddate STRING) ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/downloadlog';

CREATE EXTERNAL TABLE testresults(localfile STRING, test STRING, errors INT) ROW FORMAT
DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/testresults';

-- Load the data into the staging tables:
LOAD DATA INPATH /user/test/input/DownloadLog.csv INTO TABLE downloadlog;
LOAD DATA INPATH /user/test/input/TestResults.csv INTO TABLE testresults;
```

```
-- Create all the dimension tables and fact tables:
CREATE EXTERNAL TABLE datedim(dateid INT, downloaddate STRING, day STRING,
month STRING, year STRING, week STRING, weekyear STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/datedim';

CREATE EXTERNAL TABLE testdim(testid INT, testname STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/testdim';

CREATE EXTERNAL TABLE topdomaindim(topdomainid INT, topdomain STRING) ROW FORMAT
DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/topdomaindim';

CREATE EXTERNAL TABLE domaindim(domainid INT, domain STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/domaindim';

CREATE EXTERNAL TABLE serverdim(serverid INT, server STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/serverdim';

CREATE EXTERNAL TABLE serverversiondim(serverversionid INT, serverversion STRING,
serverid INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE
LOCATION '/user/test/serverversiondim';

CREATE EXTERNAL TABLE pagedim(pageid INT, url STRING, size INT, validfrom STRING,
validto STRING, version INT, domainid INT,serverversionid INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/pagedim';

CREATE EXTERNAL TABLE testresultsfact(pageid INT, testid INT, dateid INT, error INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE
LOCATION '/user/test/testresultsfact';

-- Load data into the non-snowflaked dimension tables, testdim and datedim:
INSERT OVERWRITE TABLE  datedim SELECT UDF_getglobalid() AS dateid, downloaddate,
UDF_extractday(downloaddate), UDF_extractmonth(downloaddate), UDF_extractyear(downloaddate),
UDF_extractweek(downloaddate), UDF_extractweekyear(downloaddate) from downloadlog;

INSERT OVERWRITE TABLE testdim SELECT UDF_getglobalid() AS testid, A.testname FROM
(SELECT DISTINCT test as testname FROM testresults) A;

-- Load data into the snowflaked dimension tables from leaves to the root:
INSERT OVERWRITE TABLE topdomaindim SELECT UDF_getglobalid() AS topdomainid, A.topdomain FROM
(SELECT DISTINCT UDF_extracttopdomain(url) FROM downloadlog) A;

INSERT OVERWRITE TABLE domaindim SELECT UDF_getglobalid() AS domainid, A.domain, B.topdomainid FROM
(SELECT DISTINCT UDF_extractdomain(url) as domain, UDF_extracttopdomain(url) as topdomain
FROM downloadlog) A JOIN topdomaindim B ON (A.topdomain=B.topdomain);

INSERT OVERWRITE TABLE serverdim SELECT UDF_getglobalid() AS serverid, A.server FROM
(SELECT DISTINCT UDF_extractserver(serverversion) AS server FROM downloadlog) A;

INSERT OVERWRITE TABLE serverversiondim SELECT UDF_getglobalid() AS serverversionid, A. serverversion,
B.serverid FROM (SELECT DISTINCT  serverversion, UDF_extractserver(serverversion) as server
FROM downloadlog) A JOIN serverdim B ON (A.server=B.server);

INSERT OVERWRITE TABLE pagedim SELECT UDF_getglobalid() AS pageid, A.url, A.size, A.lastmoddate,
B.domainid, C.serverversionid FROM (SELECT url, size, lastmoddate, UDF_extractdomain(uri) AS domain,
serverversion FROM downloadlog) A JOIN domaindim B ON (A.domain=B.domain) JOIN serverversiondim C
JOIN (A.serverversion=C.serverversion);

CREATE EXTERNAL TABLE pagedim_tmp(pageid INT, url STRING, size INT, lastmoddate STRING, domainid INT,
serverversionid INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE LOCATION '/user/test/pagedim_tmp';

-- Load data into the fact table, testresultstact:
INSERT OVERWRITE TABLE testresultsfact SELECT C.pageid, E.testid, D.dateid, B.errors
FROM downloadlog A JOIN testresults B ON (A.localfile=B.localfile)
JOIN pagedim C ON (A.url=C.url) JOIN datedim D ON (A.downloaddate=D.downloaddate)
JOIN testdim E ON (B.test=E.testname);
```

A.3 PIG

```
-- Copy the data from local file system to HDFS:
hadoop fs -copyFromLocal /tmp/DownloadLog.csv /user/test;
hadoop fs -copyFromLocal /tmp/TestResults.csv /user/test;

-- Load the data into in PIG:
downloadlog = LOAD 'DownloadLog.csv' USING PigStorage('\t')
as (localfile, url, serverversion,size,downloaddate,lastmoddate);
testresults = LOAD 'TestResults.csv' USING PigStorage('\t') as (localfile, test, errors);

-- Load the dimension table, testdim:
testers = FOREACH testresults GENERATE test as testname;
distincttestname = DISTINCT testers;
testdim = FOREACH distincttestname GENERATE UDF_getglobalid() as testid, testname;
STORE testdim INTO '/tmp/testdim' USING PigStorage();

-- Load the dimension table, datedim:
downloadates = FOREACH downloadlog GENERATE downloaddate;
distinctdownloadates = DISTINCT downloadates;

datedim = FOREACH distinctdownloadates GENERATE UDF_getglobalid() AS dateid, downloaddate,
UDF_extractday(downloaddate) AS day, UDF_extractmonth(downloaddate) AS month,
UDF_extractyear(downloaddate) AS year, UDF_extractweek(downloaddate) AS week,
UDF_extractweekyear(downloaddate) as weekyear;

STORE datedim INTO '/tmp/datedim' USING PigStorage();

-- Load the sknowflaked dimension tables:
urls = FOREACH downloadlog GENERATE url;

serverversions = FOREACH downloadlog GENERATE serverversion;

domains =  FOREACH downloadlog GENERATE UDF_extractdomain(url) as domain;

distinctdomains = DISTINCT domains;

topdomains = FOREACH distinctdomains GENERATE UDF_extracttopdomain(domain) as topdomain;

distincttopdomains = DISTINCT topdomains;

topdomaindim = FOREACH distincttopdomains GENERATE UDF_getglobalid() ad topdomainid, topdomain;
STORE topdomaindim INTO '/tmp/topdomaindim' USING PigStorage();

ndomains = FOREACH distinctdomains GENERATE domain as domain,
UDF_extracttopdomain(domain) AS topdomain;

ndomainjoin = JOIN ndomains BY topdomain, topdomaindim BY topdomain;

domaindim = FOREACH ndomainjoin GENERATE UDF_getglobalid() as domainid, domain, topdomainid;
STORE domaindim INTO '/tmp/domaindim' USING PigStorage();

distinctserverversions = DISTINCT serverversions;

nserverversions = FOREACH distinctserverversions GENERATE serverversion AS serverversion,
UDF_extractserver(serverversion) AS server;

servers = FOREACH nserverversions GENERATE server as server;
distinctservers = DISTINCT servers;

serverdim = FOREACH distinctservers GENERATE UDF_getglobalid() as serverid, server;
STORE serverdim INTO '/tmp/serverdim' USING PigStorage();

nserverversionjoin = JOIN nserverversions BY server, serverdim BY server;

serverversiondim = FOREACH nserverversionjoin GENERATE UDF_getglobalid() as serverversionid,
serverversion, serverid;
```

```
STORE serverversiondim INTO ’/tmp/serverversiondim’ USING PigStorage();

joindomservversion = JOIN (JOIN downloadlog BY UDF_extractdomain(url),
domaindim by domain) BY serverversion, serverversiondim BY serverversion;

pagedim = FOREACH joindomservversion GENERATE UDF_getglobalid() as pageid,
url, size, lastmoddate, serverversionid, domainid;

STORE pagedim INTO ’/tmp/pagedim’ USING PigStorage();

-- Load the fact tables:
testresults = JOIN downloadlog BY localfile, testresults BY localfile;

joinpagedim = JOIN testresults BY url, pagedim BY url;

joindatedim = JOIN joinpagedim BY downloaddate, datedim BY downloaddate;

jointestdim = JOIN joindatedim BY test, testdim BY testname;

testresultsfact = FOREACH jointestdim GENERATE dateid, pageid, testid, errors;

STORE testresultsfact INTO ’/tmp/testresultsfact’ USING PigStorage();
```