Aalborg Universitet

AALBORG UNIVERSITY
DENMARK

ETLMR

A Highly Scalable Dimensional ETL Framework Based on MapReduce

Liu, Xiufeng; Thomsen, Christian; Pedersen, Torben Bach

Published in:
Lecture Notes in Computer Science

DOI (link to publication from Publisher):
10.1007/978-3-642-23544-3_8

Publication date:
2011

Document Version
Early version, also known as pre-print

Link to publication from Aalborg University

Citation for published version (APA):
Liu, X., Thomsen, C., & Pedersen, T. B. (2011). ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce. *Lecture Notes in Computer Science*, *6862*, 96-111. https://doi.org/10.1007/978-3-642-23544-3_8

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from vbn.aau.dk on: April 26, 2024

ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce

Xiufeng Liu, Christian Thomsen, and Torben Bach Pedersen

Dept. of Computer Science, Aalborg University
{xiliu,chr,tbp}@cs.aau.dk

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 paper 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 paper 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* [6], 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

A. Cuzzocrea and U. Dayal (Eds.): DaWaK 2011, LNCS 6862, pp. 96–111, 2011.
© Springer-Verlag Berlin Heidelberg 2011

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 paper, 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 paper to specifically address ETL for *dimensional* schemas on MapReduce. The paper 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 paper. 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 Fig. 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 the paper is structured as follows: Section 2 gives an overview of ETLMR. Sections 3 and 4 present dimension processing and fact processing, respectively. Section 5 introduces the implementation of ETLMR in the Disco MapReduce framework, and presents the experimental evaluation. Section 6 reviews related work. Finally, Section 7 concludes the paper and provides ideas for future work.

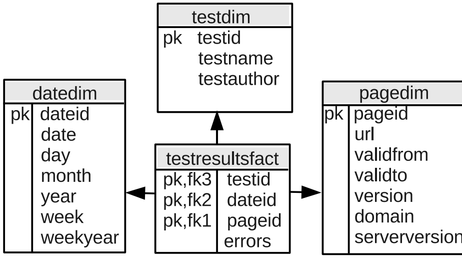

Fig. 1. Star schema of the running example

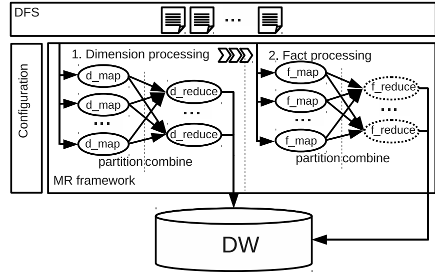

Fig. 2. Data flow on MapReduce

2 Overview

Fig. 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 Fig. 2) if no aggregation of the fact data is done before it is loaded.

Algorithm 1 shows the details of the whole process of using ETLMR. 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 DFS is installed).

Algorithm 1. The ETL process

1: Partition the input data sets;
2: Read the configuration parameters 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 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.

ETLMR defines all the run-time parameters in a configuration file, including declarations of dimension and fact tables, dimension processing methodologies, user-defined-functions (UDFs) for processing data, number of mappers and reducers, and others. A complete example is available at [9].

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

3 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.

3.1 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 Fig. 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.

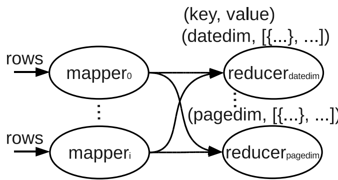

Fig. 3. ODOT

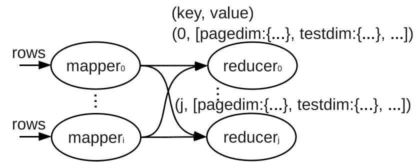

Fig. 4. ODAT

3.2 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 Fig. 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 3.4) 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.

3.3 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*.

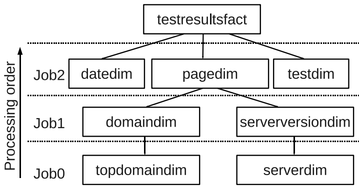

Fig. 5. Level-wise processing

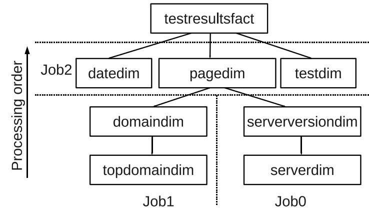

Fig. 6. Hierarchy-wise processing

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 Fig. 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 Fig. 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*.

taskid	domaindim		
	domid	dom	topdomid
1	1	www.dom1.tl1	1
1	2	www.dom2.tl2	2
2	1	www.dom2.tl2	1

taskid	topdomaindim	
	topdomid	topdom
1	1	tl1
1	2	tl2
2	1	tl2

taskid	pagedim					
	pageid	url	validfrom	validto	version	domid
1	1	www.dom1.tl1/p0.htm	2010-01-01	null	1	1
1	2	www.dom2.tl2/p0.htm	2010-01-01	null	1	2
2	1	www.dom2.tl2/p0.htm	2010-12-31	null	1	1

Fig. 7. Before post-fixing

domaindim		
domid	dom	topdomid
1	www.dom1.tl1	1
2	www.dom2.tl2	2

topdomaindim	
topdomid	topdom
1	tl1
2	tl2

pagedim					
pageid	url	validfrom	validto	version	domid
1	www.dom1.tl1/p0.htm	2010-01-01	null	1	1
2	www.dom2.tl2/p0.htm	2010-01-01	2010-12-31	1	2
3	www.dom2.tl2/p0.htm	2010-12-31	null	2	2

Fig. 8. After post-fixing

3.4 Post-fixing

As discussed in Section 3.2, 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. *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 Fig. 8.*

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

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
```

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.

3.5 Offline Dimensions

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 the 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*. As the ODOT (offline) is similar to the ODOT we discussed in Section 3.1, we now only describe the latter. 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). As the input data for most data-intensive dimension is partitioned based on business keys, the rows with identical business key values are processed within the same mapper such that when we employ a global ID generator to generate the dimension key values, the post-fixing is not needed. This improves the processing performance.

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

4 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 pseudocode for processing facts.

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.

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*)

5 Implementation and Evaluation

ETLMR uses and extends *pygrametl* [14], a Python code-based programming framework, which enables high programmer productivity in implementing an ETL program. We choose Disco [2] as our MapReduce platform since it has the best support for Python. In the rest of this section, we measure the performance achieved by the proposed methods. We evaluate the system scalability on various sizes of tasks and data sets and compare with other business intelligence tools using MapReduce.

5.1 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.

5.2 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 [5], 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 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).

5.3 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 Fig. 9 and Fig. 10, 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 Fig. 10 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.

5.4 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

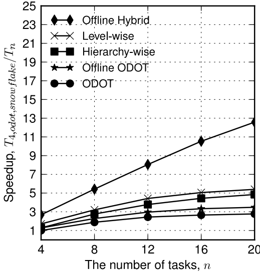

Fig. 9. Parallel ETL for snowflake schema, 20 GB

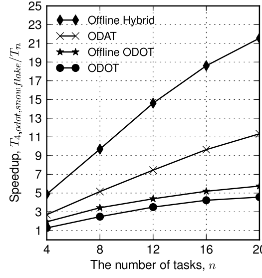

Fig. 10. Parallel ETL for star schema, 20 GB

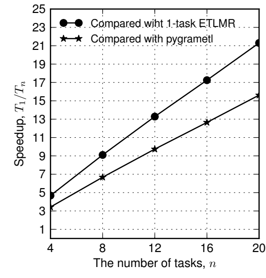

Fig. 11. Speedup with increasing tasks, 80 GB

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. Fig. 11 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 1). 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

Table 1. 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

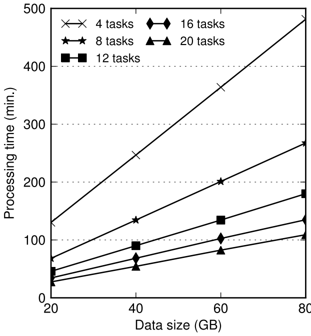

Fig. 12. Proc. time when scaling up bigdim data

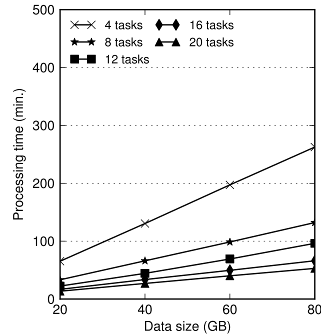

Fig. 13. Proc. time when scaling up smalldim data

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 3.4).

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.

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. Fig. 12 and Fig. 13 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.

5.5 Comparison with other Data Warehousing Tools

There are some MapReduce data warehousing tools available, including Hive [15,16], Pig [10] and Pentaho Data Integration (PDI) [3]. 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 available in the full paper [9].

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 2 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 2. 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

6 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 [10,15]. 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 5.5), Chaiken et al. present the SQL-like language SCOPE [4] on top of Microsoft's Cosmos MapReduce and distributed file system. Friedman et al. introduce SQL/MapReduce [7], 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 pipe-lined queries instead of small independent tasks as in MapReduce. The database research community has recently compared the two classes of systems. Pavlo et al. [11], and Stonebraker et al. [13] 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. [5] 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 [13].

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.

7 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 paper, 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 paper 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.

References

1. `wiki.apache.org/hadoop/PoweredBy` (June 06, 2011)
2. `http://www.discoproject.org/` (June 06, 2011)
3. `http://www.pentaho.com` (June 06, 2011)

4. Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: easy and efficient parallel processing of massive data sets. PVLDB 1(2), 1265–1276 (2008)
5. Dean, J., Ghemawat, S.: MapReduce: A Flexible Data Processing Tool. CACM 53(1), 72–77 (2010)
6. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of OSDI, pp. 137–150 (2004)
7. Friedman, E., Pawlowski, P., Cieslewicz, J.: SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions. PVLDB 2(2), 1402–1413 (2009)
8. Kovoor, G., Singer, J., Lujan, M.: Building a Java MapReduce Framework for Multi-core Architectures. In: Proc. of MULTIPROG, pp. 87–98 (2010)
9. Liu, X., Thomsen, C., Pedersen, T.B.: ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce. In: DBTR-29. Aalborg University (2011), www.cs.aau.dk/DBTR
10. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-so-foreign Language for Data Processing. In: Proc. of SIGMOD, pp. 1099–1110 (2008)
11. Pavlo, A., Paulson, E., Rasin, A., Abadi, D., DeWitt, D., Madden, S., Stonebraker, M.: A Comparison of Approaches to Large-scale Data Analysis. In: Proc. of SIGMOD, pp. 165–178 (2009)
12. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: Proc. of HPCA, pp. 13–24 (2007)
13. Stonebraker, M., Abadi, D., DeWitt, D., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and Parallel DBMSs: friends or foes? CACM 53(1), 64–71 (2010)
14. Thomsen, C., Pedersen, T.B.: pygrametl: A Powerful Programming Framework for Extract-Transform-Load Programmers. In: Proc. of DOLAP, pp. 49–56 (2009)
15. Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: A Warehousing Solution Over a Map-reduce Framework. PVLDB 2(2), 1626–1629 (2009)
16. Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive – A Petabyte Scale Data Warehouse Using Hadoop. In: Proc. of ICDE, pp. 996–1005 (2010)
17. Yoo, R., Romano, A., Kozyrakis, C.: Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-memory System. In: Proc. of IISWC, pp. 198–207 (2009)