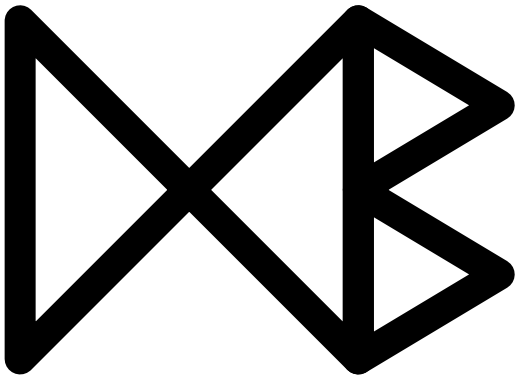

CloudETL: Scalable Dimensional ETL for Hive

Xiufeng Liu, Christian Thomsen and Torben Bach Pedersen

July, 2013

TR-31

A DB Technical Report

Title	CloudETL: Scalable Dimensional ETL for Hive
Author(s)	Xiufeng Liu, Christian Thomsen and Torben Bach Pedersen
Publication History	July 2013 (replaces a previous version). A DB Technical Report

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) programs process data from sources into data warehouses (DWs). Due to the rapid growth of data volumes, there is an increasing demand for systems that can scale on demand. Recently, much attention has been given to *MapReduce* which is a framework for highly parallel handling of massive data sets in cloud environments. The MapReduce-based *Hive* has been proposed as an RDBMS-like system for DWs and provides good and scalable analytical features. It is, however, still challenging to do proper dimensional ETL processing with (relational) Hive; for example, the concept of *slowly changing dimensions* (SCDs) is not supported (and due to lacking support for UPDATEs, SCDs are both complex and hard to handle manually). To remedy this, we here present the cloud-enabled ETL framework *CloudETL*. CloudETL uses *Hadoop* to parallelize the ETL execution and to process data into Hive. The user defines the ETL process by means of high-level constructs and transformations and does not have to worry about the technical details of MapReduce. CloudETL provides built-in support for different dimensional concepts, including star schemas and SCDs. In the paper, we present how CloudETL works. We present different performance optimizations including a purpose-specific data placement policy to *co-locate* data. Further, we present an extensive performance study and compare with other cloud-enabled systems. The results show that CloudETL scales very well and significantly outperforms the dimensional ETL capabilities of Hive both with respect to performance and programmer productivity. For example, Hive uses 3.9 times as long to load an SCD in an experiment and needs 112 statements while CloudETL only needs 4.

1 Introduction

In data warehousing, data from different source systems is processed into a central DW by an Extract–Transform–Load (ETL) process in a periodic manner. Traditionally, the DW is implemented in a relational database where the data is stored in fact tables and dimension tables which form a star schema [15]. Many enterprises collect and analyze hundreds of gigabytes data each day and there is an increasing need for a new data warehousing architecture that can achieve better scalability and efficiency. With the emergence of cloud computing technologies, such as MapReduce [5], many enterprises have shifted away from deploying their analytical systems on high-end proprietary machines and instead moved towards clusters of cheaper commodity machines [1]. The system Hive [25] uses the Hadoop [10] MapReduce implementation and can be used for scalable data warehousing. Hive stores data in the Hadoop Distributed File System (HDFS), and presents the data by logical *tables*. The data in the tables is queried by user-written (SQL-like) *HiveQL* scripts which are translated into MapReduce jobs to process the data. However, Hive only has limited dimensional ETL capabilities and it is not straightforward to use in an ETL process. It is more like a DBMS and less like an ETL tool. For example, Hive lacks support for high-level ETL-specific constructs including those for looking up a dimension member or, if not found, updating the dimension table. There is also no specialized support for the commonly used SCDs [15]. Writing HiveQL scripts for such processing is cumbersome and requires a lot of programming efforts [17]. In addition, Hive also lacks support for UPDATEs which makes handling of SCDs even more complicated when time-valued attributes are used to track the changes of dimension values.

In this paper, we present *CloudETL* which is a scalable dimensional ETL framework for Hive. CloudETL supports the aforementioned ETL constructs. CloudETL sits on top of Hive and aims at making it easier and faster to create scalable and efficient ETL processes that load data into Hive DWs. CloudETL allows ETL programmers to easily translate a high-level ETL design into actual MapReduce jobs on Hadoop by only using high-level constructs in a Java program and without handling MapReduce details. We provide a library of commonly used ETL constructs as building blocks. All the complexity associated with parallel programming is transparent to the user, and the programmer only needs to think about how to apply the constructs to a given DW schema leading to high programmer productivity.

The contributions of this paper are listed in the following: First, we present a novel and scalable dimensional ETL framework which provides direct support for high-level ETL constructs, including handling of star schemas and SCDs. Second, we present a method for processing SCDs enabling update capabilities in a cloud environment. Third, we present how to process big dimensions efficiently through purpose-specific *co-location* of files on the distributed file system, and we present *in-map updates* to optimize dimension processing. Fourth, we present lookup indices and multi-way lookups for processing fact data efficiently in parallel. Fifth, we provide an extensible set of high-level transformations to simplify the implementation of a parallel, dimensional ETL program for a MapReduce environment. Finally, we provide an extensive experimental evaluation of the proposed techniques.

The rest of the paper is structured as follows. In Section 2, we give an overview of CloudETL and its components and introduce a running example. In Section 3, we detail dimension processing including the parallelization for multiple dimension tables, co-location of data, and the updates for SCDs. In Section 4, we present the approach for processing facts. In Section 5, we describe the implementation of CloudETL. In Section 6, we study the performance CloudETL and compare with other systems. In Section 7, we present the related work. Finally, in Section 8, we summarize the paper and discuss the future research directions.

2 System Overview

CloudETL employs Hadoop as the ETL execution platform and Hive as the warehouse system (see Figure 1). CloudETL has a number of components, including the application programming interfaces (APIs) used by ETL programs, ETL transformers performing data transformations, and a job manager used to control the execution of the jobs to submit to Hadoop.

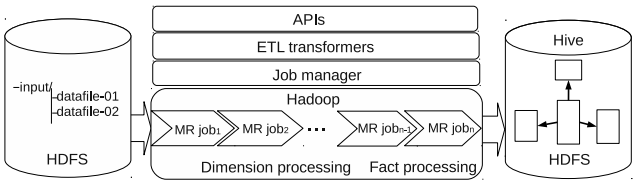

Figure 1: CloudETL Architecture

The ETL workflow in CloudETL consists of two sequential steps: dimension processing and fact processing. The source data is assumed to be present in HDFS (including but not limited to Hive files) when the MapReduce (MR) jobs are started (see the left of Figure 1). CloudETL allows processing of data into multiple tables within a job. The source data is processed into dimension values or facts by the user-specified transformations which are executed by mappers and reducers. A transformer integrates a number of transformations used for processing data, such as data type convertions, lookups (for getting dimension key values), and others. The CloudETL job manager submits the jobs to the Hadoop JobTracker in sequential order. The jobs for dimension processing are to be run before the jobs for fact processing (see the middle of Figure 1) as processing facts requires looking up referenced primary key values from the dimension tables.

Hive employs HDFS for physical data storage but presents the data in the HDFS files as logical tables. Therefore, when Hive is used, we can easily write data directly into files which then can be used by Hive. The right-most part of Figure 1 shows a star schema in Hive which consists of four (or more) files in HDFS.

Running Example In the following, we use a running example to show how CloudETL processes data into dimension tables and fact tables. This example is inspired by our work in a previous project [21]. This example considers a DW with data about tests of web pages. The star schema shown in Figure 2 consists of a fact table `testresultsfact` with the single measure `errors` telling how many errors were detected on a given version of a web page on a given date. There are three dimension tables, `testdim`,

`pagedim`, and `datedim`, which represent tests, web pages, and dates, respectively. Note that `pagedim` is a "type-2" SCD [15] which tracks changes by having multiple versions of its dimension values. To track the different versions, there is a `version` attribute which represents the *version number* and `validfrom` and `validto` attributes which hold dates representing when a given version was valid. Note that `validto` typically is NULL (i.e., unknown) for a new version and must be updated later on when another version appears. `pagedim` is also a *data-intensive* dimension table which contains many more rows than the other two dimension tables. We use this example, instead of a more common example such as TPC-H [26], because it has an SCD and a data-intensive dimension. The chosen example thus allows us to illustrate and test our system more comprehensively.

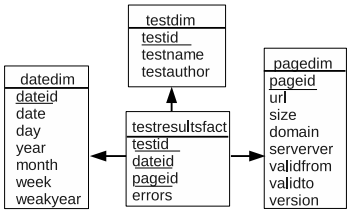

Figure 2: Star schema

3 Dimension Processing

3.1 Challenges and Designs

In conventional ETL processing, many transformation capabilities rely on the underlying DW DBMS to, e.g., automatically generate sequential key values, perform SCD updates, and insert rows into dimension tables and fact tables. However, support for all these is not available in Hive and Hadoop, and in particular, Hive and other MapReduce-based programs for data analysis do not support the UPDATE operation known from SQL. In addition, a number of limitations make the ETL on Hadoop more challenging. For example, the nodes running the MR jobs share no global state, which makes certain tasks (such as handling different dimension value versions correctly) more difficult for an ETL process, and Hadoop does not hold schema information for the data to process, which makes data cleansing difficult.

We address these limitations with the following design which is explained further in the following sections: First, a line read from the data source is made into a *record* which migrates from mappers to reducers. A record contains schema information about the line, i.e., the names of attributes and the data types. For example, a record with `pagedim` data has the schema ⟨`url string, size int, moddate date`⟩. Example attribute values are ⟨`www.dom.com/p0.htm, 10, 2012-02-01`⟩. Second, to correctly maintain an SCD, we union the *incremental* data (i.e., the new data to add to the DW such as new test results of new pages) with the existing dimension data that has already been processed into a dimension table, and replace the attribute values for both when necessary. Third, we assign each new row a unique, sequential key value.

3.2 Execution Flow

Figure 3 shows the execution flow for dimension processing. For simplicity, we only show the ith reducer which processes a part of the intermediate data output by all mappers. The dimension source data in HDFS is *split* (by Hadoop) and assigned to the map tasks. The records from a file split are processed by user-specified transformations in the mappers. Note that a mapper can process a source with data that will go to different

dimensions. In the *shuffling*, the data is sent to different reducers. The reducer output (corresponding to a dimension table) is written to the HDFS.

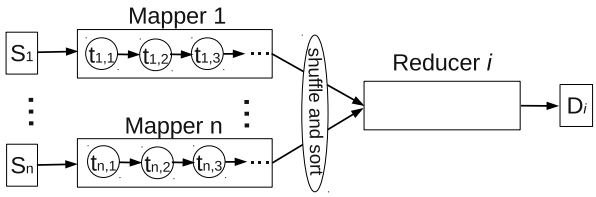

Figure 3: Execution flow of dimension processing

We now describe how to process changes by using the `pagedim` dimension as an example. All dimensions are considered to be SCDs, but if there are no changes in the source data, the dimensions will of course not be updated. The main problem of processing SCDs is how to update the special SCD type-2 attribute values (with validity dates and version numbers) in a MapReduce environment. For type-2 SCDs, the validity dates and version numbers are updated by following the original change order in the source data. For example, the end date of a given version is set to the start date of its successor. For type-1 SCDs, the attribute values of a dimension member are overwritten by new values. When doing incremental loading, we also need to update the versions that have already been loaded into Hive. The idea here is to collect different versions of dimension values from both the incremental data and the existing dimension data, and to perform the updates in reducers. Therefore, the execution flow is as follows: First, we do the transformations on the incremental data in mappers, then hash partitioning on the *business keys* (a dimension must have a key that distinguishes its members) of the map output. For example, we partition the map output for `pagedim` based on the values of the business key `url`. Therefore, the rows with identical key values are shuffled to the same reducer. To acquire the original change order, we sort the intermediate data by the modification date. For example, `moddate` of the `pagedim` source data tells when a given page was changed. If the source data does not include a date, we assume that the line numbers show the changing order (line numbers should be explicitly given in input data if the input is made up of several files). Alternatively, the user can choose that another attribute should be used to get the change order.

In any case, we include the sorting attribute(s) in the key of the map output since Hadoop only supports sorting on keys, but not on values. We include both the business key and the SCD date (or another sorting attribute) in the key of the map output (this is an application of the *value-to-key* MapReduce design pattern [16]). In addition, we make a task support processing of multiple dimension tables by tagging the key-value pairs with the name of the dimension table to which the data will be written. Therefore, the map output has the format $(\langle$ *the name of a dimension table, business key, SCD date/line no.* $\rangle, \langle$ *the rest of dimension values*$\rangle)$ where the key is composite of three attributes.

To make the above more concrete, we in Figure 4 show the input and output of map and reduce when processing `pagedim`. We assume that a single dimension value has already been loaded into Hive, and now we perform incremental loading with two new values. As shown in Figure 4, the map input consists of both *incremental data* and *existing dimension data*. We discuss an optimization in Section 3.5. A record from the incremental data has the attributes `url`, `size`, and `moddate` which indicate the web page address, the size (may change when the page is modified) and the modification date of a page, respectively. The existing dimension record contains the three additional SCD attributes `version`, `validfrom`, and `validto`. In the mapper, we transform the raw incremental data into dimension values by adding the three additional SCD attributes and the surrogate key `id`. Then, the mapper emits the records in the described output format. For the existing dimension record, no transformation is needed, but the record is re-structured in line with the map output format. This results in the shown *map output*. When the map output is shuffled to the reducers, the key-value pairs are grouped by the composite key values and sorted by the validity SCD

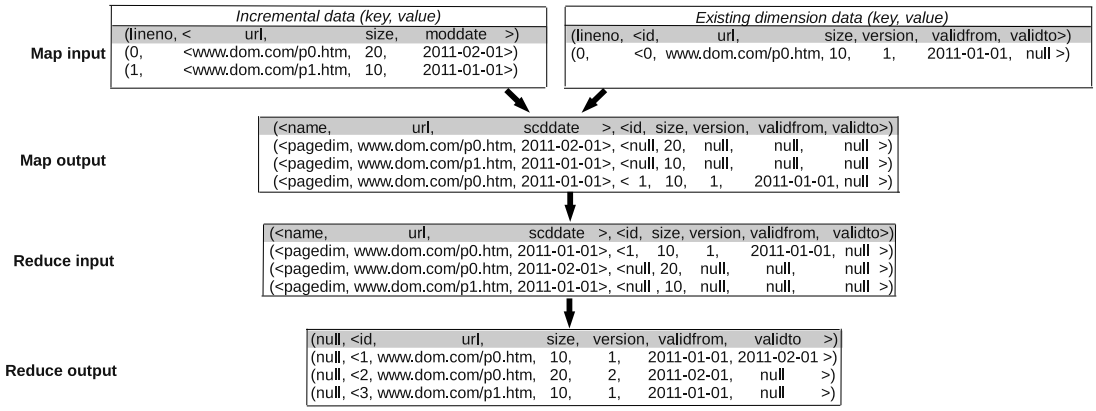

Figure 4: MapReduce input and output when processing the type-2 SCD `pagedim`.

date in ascending order (see the *reduce input* in Figure 4). In the reducer, unique numbers are assigned to the key attribute, i.e., `id`, of the two new dimension values and the values of the SCD attributes are updated, i.e., `validto` of a version is updated to the starting valid date of the following version (see `validfrom` and `validto` of the different versions for `url=www.dom.com/p0.htm`). The version number is also updated accordingly. The three records are finally written to HDFS; null in the `validto` attribute represents that a dimension record is valid till now (see the *reduce output* in Figure 4).

3.3 Algorithm Design

Listing 1 shows pseudocode for the mapper. In the code, Γ is a sequence of transformations to apply as defined by the user. A transformation can thus be followed by other transformations. The first transformation defines the schematic information of the data source such as the names of attributes, the data types, and the attributes for sorting of versions (such as a date).

Listing 1 Mapper

```
1: class Mapper
2:     method INITIALIZE( )
3:         Γ ← GETTRANSFORMATIONS()
4:     method MAP(offset o, Record r)
5:         for all t ∈ Γ do
6:             r ← t.PROCESSRECORD(r)          ▷ Returns ⊥ if r is filtered out.
7:             if r = ⊥ then return            ▷ Do nothing
8:             else
9:                 key ← CREATECOMPOSITEKEY(r, d)
10:                value ← CREATEVALUE(r, d)
11:                EMIT(key, value)
12: ■
```

Listing 2 shows the reducer code (in Section 3.5, we present a specialized map-only method for big dimensions). The input is automatically grouped by the composite key values and sorted (by the validity date or line no.) before it is fed to the REDUCE method. The REDUCE method reads all the dimension data with a particular composite key value. Further, CloudETL uses its own partitioning function such that all data with a given business key is processed by the same reducer.

For type-2 SCDs (lines 4–13), we keep the dimension values temporarily in a buffer (lines 5–10), assign a sequential number to the key of a new dimension record (line 9), and update the SCD attribute values (line 11), including the validity dates and the version number. The method MAKEDIMENSIONRECORD extracts the dimension's business key from the composite key given to the mapper and combines it with the remaining values. Finally, we write the reduce output with the name of the dimension table as the key,

and the dimension data as the value (line 12). The reduce output format is customized so that the value is written into the directory named by the output key value. For type-1 SCDs (lines 14–21), we overwrite the old values. That is, we only keep the latest version with a given business key value. An important point here is that if r_0 has a primary key value set (here denoted $r_0[id]$), the data in r_0 comes from the existing table and the primary key value should be reused (line 18). If the primary key value is not set in r_0, we are handling an entirely new member and should get a new primary key value.

Listing 2 Reducer

```
1: class Reducer
2:     method REDUCE(CompositeKey key, values[0...n])
3:         name ← GETNAMEOFDIMENSIONTABLE(key)
4:         if DIMTYPE(name) = type2SCD then
5:             L ← new LIST()
6:             for i ← 0, n do
7:                 r ← MAKEDIMENSIONRECORD(key, values[i])
8:                 if r[id] = ⊥ then                          ▷ id is the dimension key
9:                     r[id] ← GETDIMENSIONID(name)
10:                L.ADD(r)
11:            UPDATESCDATTRIBUTEVALUES(L)
12:            for all r ∈ L do
13:                EMIT(name, r)
14:        else                                               ▷ Type-1: key value from the 1st record and the rest from the last
15:            r_0 ← MAKEDIMENSIONRECORD(key, values[0])
16:            r_n ← MAKEDIMENSIONRECORD(key, values[n])
17:            if r_0[id] ≠ ⊥ then
18:                r_n[id] ← r_0[id]
19:            else
20:                r_n[id] ← GETDIMENSIONID(name)
21:            EMIT(name, r_n)
22: ■
```

3.4 Pre-update in Mappers

In Hadoop, it is relatively time-consuming to write map output to disk and transfer the intermediate data from mappers to reducers. For type-1 SCDs, we thus do *pre-updates* in mappers to improve the efficiency by shrinking the size of the intermediate data shuffled to the reducers. This is done by only transferring the resulting dimension member (which may have been updated several times) from the mapper to the reducer. On the reduce side, we then do *post-updates* to update the dimension to represent the dimension member correctly.

Listing 3 shows how pre-updates for type-1 SCDs are handled. For simplicity, we no longer show the ETL transformation operations in this algorithm. In the map initialization (line 3), we create a hash map M to cache the mappings of a business key value to the corresponding dimension values. Since the state of M is preserved during the multiple calls to the MAP method, we can use M until the entire map task finishes when it has processed its *split* (in Hadoop, the default split size is 64MB). In the mapper, the dimension attribute values are always updated to the latest version's if there are any changes (lines 6–10). Here, we should preserve the key value (i.e., the value of id) if the member is already represented, but update the other attribute values (lines 9 and 10). The construction and emission of the composite key-value pairs are deferred to the CLOSE method which is called when the mapper has finished processing a file split.

With the pre-updates, we can decrease the size of the intermediate data transferred over the network, which is particularly useful for data with frequent changes. Of course, we can also do the updates in a combiner. However, doing pre-updates in a combiner would typically be more expensive since we have to transfer the intermediate data from the mapper to the combiner, which involves object creation and

Listing 3 Pre-update in mappers for type-1 SCDs

```
1: class Mapper
2:     method INITIALIZE( )
3:         M ← new HASHMAP()
4:     method MAP(offset o, Record r)
5:         [Perform transformations...]
6:         k ← GETBUSINESSKEY(r)
7:         prev ← M[k]
8:         if prev ≠ ∅and (prev[scddate] < r[scddate]) then
9:             r[id] ← prev[id]                    ▷ Preserve id of the existing dimension
10:        M[k] ← r                                ▷ Update the old attribute values
11:    method CLOSE( )
12:        for all m ∈ M do
13:            key ← CREATECOMPOSITEKEY(m)
14:            value ← CREATEVALUE(m)
15:            EMIT(key, value)
16: ■
```

destruction, and object serialization and deserialization if the in-memory buffer is not big enough to hold the intermediate data.

3.5 Processing of Big Dimensions

Typically, the size of dimension data is relatively small compared to the fact data and can be efficiently processed by the method we discussed above. This is the case for `datedim` and `testdim` of the running example. However, some dimensions – such as `pagedim` – are very big and have much more data than typical dimensions. In this case, shuffling a large amount of data from mappers to reducers is not efficient. We now present a method that processes data for a big dimension in a *map-only* job. The method makes use of *data locality* in HDFS and is inspired by the general ideas of CoHadoop [9], but is in CloudETL automated and made purpose-specific for dimensional data processing. We illustrate it by the example in Figure 5. Consider an incremental load of the `pagedim` dimension and assume that the previous load resulted in three dimension data files, D_1, D_2, and D_3, each of which is the output of a task. The files reside in the three data nodes `node1`, `node2`, and `node3`, respectively (see the left of Figure 5). For the incremental load, we assume the incremental data is partitioned on the business key values using the same partitioning function as in the previous load. Suppose that the partitioning has resulted in two partitioned files, S_1 and S_3. The hash values of the business keys in D_1 and S_1 are congruent modulo the number of partitions and the same holds for D_3 and S_3. When S_1 and S_3 are created in HDFS, *data co-location* is applied to them, i.e., S_1 is placed together with D_1 and D_3 is placed together with S_3 (see the right of Figure 5). Then, a map-only job is run to process the co-located data on each node locally. In this example, note that no incremental data is co-located with D_2. The existing dimension data in D_2 is not read or updated during the incremental load.

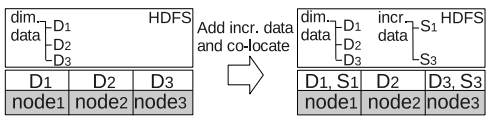

Figure 5: Co-location of files in HDFS

Figure 6 shows how the *blocks* of the co-located files are handled. For simplicity, we show the blocks when the replication factor is set to 1. As shown, all blocks of co-located files are placed on the same data node. For example, the blocks of D_1 (d_{11}, d_{12}, and d_{13}) and the blocks of S_1 (s_{11} and s_{12}) are on `node1`.

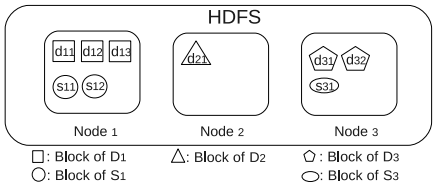

Figure 6: Data blocks of co-location (repl. factor = 1)

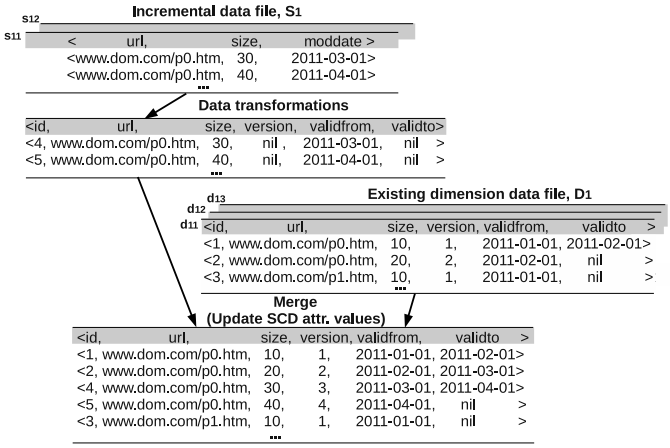

Figure 7: Type-2 SCD updates in mapper for D_1 and S_1

If the replication factor is different from 1, the block replicas of the co-located files are *also co-located* on other nodes.

Since Hadoop 0.21.0, users can use their own *block placement policy* by setting the configuration property "dfs.block.replicator.classname". This does not require re-compiling Hadoop or HDFS. The block placement policy used by CloudETL is shown in Listing 4. In CloudETL, we (unlike CoHadoop) co-locate the files based on their names, i.e., the files whose names match the same regular expression are co-located. The regular expressions are defined in the name node configuration file, `core-site.xml`. For example, if we define the regular expression `(.*\.page1)`, the data blocks of the existing dimension data files and the partitioned incremental files with the name extension `.page1` are placed together. When the name node starts up, a hash dictionary, M, is created to hold mappings from a data node to information about the blocks on the node, i.e., the total number of blocks of the files whose names match a regular expression (lines 2–12). When an HDFS client writes a block, it first asks the name node to choose target data nodes for the block and its replicas. The name node checks if the name of the file matches the regular expression. If the name matches, the name node chooses the targets based on the statistics in M. If M is empty, the client is writing the first block of co-located data and the name node chooses the targets by the default policy and updates M (lines 28–31). If M is non-empty, the name node chooses the targets based on the existing co-located data blocks in HDFS. The name node selects a data node for each replica to store. As in CoHadoop, the data nodes with the highest number of blocks and with enough space are selected (this is done by sorting M by its values, adding the nodes into a queue Q, and checking the nodes in an descending order, see line 17–22). When all the nodes in Q have been checked, but fewer data nodes than needed have been selected, the name node chooses the remaining nodes randomly. Each of the chosen nodes is tested to see if it meets the selection criteria and has sufficient space (lines 23–27). If the file name does not match the regular expression, the file should not be co-located with anything and the name node uses the default policy of HDFS to choose the targets (lines 33).

CloudETL also offers a program for partitioning data that is not already partitioned. It runs a MapReduce job to partition the data into a number of files in HDFS based on a user-specified business key. Data

Listing 4 Choosing targets in the block placement policy

```
1: class BlockPlacementPolicy
2:     method INITIALIZE( )
3:         M ← new HASHMAP()
4:         regex ← GETFILENAMEREGEXFROMCONFIG()
5:         n ← GETREPLICANUMFROMCONFIG()
6:         F ← GETFILESFROMNAMESYSTEM(regex)                    ▷ Get matching file names
7:         for all f ∈ F do
8:             B ← GETBLOCKSFROMBLOCKMANAGER(f)
9:             for all b ∈ B do
10:                D ←GETDATANODES(b)
11:                for all d ∈ D do
12:                    M[d] ← M[d] + 1
13:    method CHOOSETARGETS(String filename)
14:        D ←new COLLECTION()
15:        if regex.MATCHES(filename) then
16:            if SIZE(M) > 0 then
17:                Q ← GETDATANODES(M)                          ▷ Sorted desc. by number of blocks.
18:                while SIZE(Q) > 0 and SIZE(D) < n do
19:                    d ← Q.POP()
20:                    if ISGOODTARGET(d) then
21:                        D.ADD(d)
22:                        M[d] ← M[d] + 1
23:                while SIZE(D) < n do
24:                    d ← CHOOSERANDOM()
25:                    if ISGOODTARGET(d) then
26:                        D.ADD(d)
27:                        M[d] ← M[d] + 1
28:            else
29:                D ← CHOOSETARGETSBYDEFAULTPOLICY(filename)
30:                for all d ∈ D do
31:                    M[d] ← M[d] + 1
32:        else
33:            D ← CHOOSETARGETSBYDEFAULTPOLICY(filename)
34:        return D
35: ■
```

with the same business key value is written into the same file and sorted by SCD attribute values. For example, before loading `pagedim`, we can partition the incremental source data on `url` and sort it on `moddate` if it is not already partitioned.

As a partitioned file and its co-located existing dimension data file both are sorted, we can simply run a map-only job to merge the data from the two co-located files (the blocks of a resulting file will also be stored together). Figure 7 illustrates the processing of `pagedim` on `node1` which contains the co-located files D_1 and S_1. The lines from S_1 (in the blocks s_{11} and s_{12}) are first processed by user-specified transformations. We then merge them with the lines from the existing dimension data file D_1 (in the blocks d_{11}, d_{12} and d_{13}). The SCD updates are performed during the merging, and the final dimension values are written to HDFS. As explained above, it can, however, happen that the incremental data does not get totally co-located with the existing dimension data, e.g., if a data node lacks free space. In that case, a reduce-side update should be used.

4 Fact Processing

Fact processing is the second step in the ETL flow in CloudETL. It involves reading and transforming[1] source data and then retrieving surrogate key values from the referenced dimension tables. We call this

[1]The current prototype does transformations only in the mappers, but this could be extended to also allow *aggregating* transformations in the reducers.

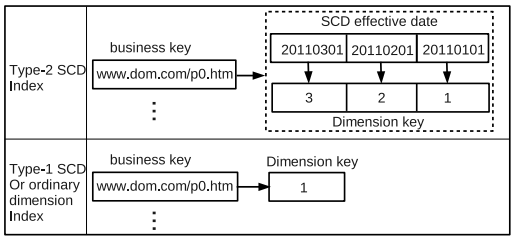

Figure 8: Lookup indices for the `pagedim` dimension

operation *lookup*. Hive, however, does not support fast lookups from its tables. In addition, the size of the fact data is typically very large, several orders of magnitude larger than the size of the dimension data. It is, thus, very important to support efficient fact processing. Therefore we use *multi-way lookups* to retrieve the dimension key values through so-called *lookup indices* and exploit a map-only job to process the fact data.

4.1 Lookup Indices

A lookup index (which is generated as a side activity during dimension processing) contains the minimal information needed for doing lookups, including the business key values, the dimension key values, and the SCD dates for type-2 SCDs. The data of lookup indices is read into main memory in the map initialization. The structures of lookup indices for type-2 and type-1 SCDs are shown in Figure 8. As a type-2 SCD index needs to keep track of the changes of the dimension, the index maps a business key value to all the versions that share that business key value. The result is thus a list of key-value pairs with the format $\langle$ *SCD effective date, dimension key*$\rangle$ in descending order such that the newest version is the first item to be retrieved by a lookup operation. Given the business key `www.dom.com/p0.htm` and the date `20110301`, we can thus find the surrogate key value of the correct dimension member version by first using the hash dictionary and then getting the first element from the resulting list. For type-1 SCDs, only the first step is needed (see the lower part in Figure 8).

In dimension processing, a lookup index file is generated for the incremental data for each dimension table. It is stored as a Hadoop sequence file and maps from business-key values to dimension-key values. The lookup index is distributed and kept in each node permanently (for handling of big dimensions, see Section 4.3).

4.2 Multi-way Lookups

We now describe how the lookup indices are used during the fact processing (see Listing 5).

We run a *map-only* job to process the fact data. We thus avoid the sorting and shuffling as well as the reduce step and get better efficiency. In the map initialization, the lookup indices relevant to the fact data are read into main memory (lines 2–7). The mapper first does all data transformations (line 9). This is similar to the dimension processing. Then, the mapper does multi-way lookups to get dimension key values from the lookup indices (lines 10–11). Finally, the mapper writes the map output with the name of the fact table as the key, and the record (a processed fact) as the value (line 12). The record is directly written to the directory in HDFS (named by the key of map output) using a customized record writer. Note that all mappers can work in parallel on different parts of the fact data since the lookup indices are distributed.

We now give more details about the lookup operator in the algorithm (lines 13–22). If it is a lookup in a type-2 SCD, we first get all the versions from the SCD index by the business key, bk (line 17). Recall that different versions are sorted by the SCD effective dates in descending order. We get the correct version by comparing the effective date of a version and the SCD date sd (lines 18–20). For the lookup index of a type-1 SCD table, the dimension key value is returned directly through a hash lookup operation (line 22).

Listing 5 Map for fact processing and lookup

```
1: class Mapper
2:     method INITIALIZE( )
3:         f ← GETCURRENTFACTTABLE()
4:         D ← GETTHEREFERENCEDDIMENSIONS(f)
5:         for all d ∈ D do
6:             if LKI[d] = ∅ then
7:                 LKI[d] ← Read the lookup index of d from local file system
8:     method MAP(offset o, Record r)
9:         r ← TRANSFORM(r)
10:        for all d ∈ D do
11:            r[d.key] ← LOOKUP(d, r)
12:        EMIT(f, r)
13:    method LOOKUP(Dimension d, Record r)
14:        bk ← r[d.businesskey]
15:        if d.type = type2SCD then
16:            sd ← r[d.scdDate]
17:            V ←LKI[d][bk]                    ▷ Get versions by the business key bk
18:            for all v ∈ V do
19:                if sd ⩾ v.date then
20:                    return v.id
21:        else
22:            return LKI[d][bk]
23: ■
```

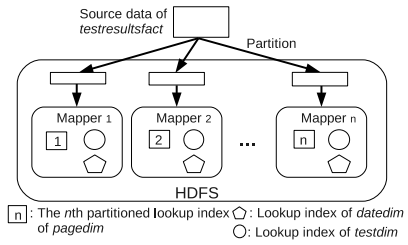

Figure 9: Fact processing with partitioned lookup index

4.3 Lookup on a Big Dimension Table

Typically, the lookup indices are small and can be fully cached in main memory. However, when a dimension table is very big, and its lookup index is too big to be fully cached in the main memory, we propose the following two approaches for retrieving dimension key values. The first is a *hybrid solution* where a Hive join and multi-way lookups are used. The source fact data is first joined in Hive with the big dimension table to retrieve the dimension key values and then the lookup indices are used for the small dimension tables in CloudETL. The other solution, called the *partitioned lookup-index solution*, uses multi-way lookups for both big and small dimension tables which requires using a partitioned lookup index. The partitioned lookup index is generated for the big dimension table. Recall that we assume that the source data for a big dimension table is partitioned on the business key values. A partition of the data is processed into dimension values saved in a data file in HDFS. At the same time, a lookup index is also generated for each partition. We call this a *partitioned lookup index*. The partitioned lookup indexes are stored in files that are co-located with the corresponding data files. When the fact source data is also partitioned with the same partitioning function as for the big dimension data (this is, e.g., possible in case the source data comes from another MapReduce job or from a database), we can exploit the co-location of partitioned lookup index files and data files and run a map-only job to do the multi-way lookups. We illustrate this in Figure 9. Suppose the job runs n mappers, each of which processes a partition of fact source data. Each mapper reads a partition of the lookup index for `pagedim`, and the (full) lookup indices for `datedim` and `testdim` (small dimension tables) into memory, and then does multi-way lookups.

5 Implementation

5.1 Input and Output

The output of CloudETL is organized as hierarchical directories and data files in HDFS. Hive employs HDFS to store data. A folder and the files in it can be recognized as a logical table without changing the file format. When a table is created, a folder with the same name is created in HDFS. CloudETL can insert data into a table by simply adding files to its folder.

We now describe the input. As discussed in Section 3, processing SCDs requires updating both existing and incremental data and the input consists of both the incremental data and the existing dimension data. The existing data files of a dimension table are in a folder in the HDFS. When a dimension-processing job starts, the existing data files are first moved into a temporary directory. Moving files between folders only needs to update the meta-data in the name node. The job then processes the data in the temporary folder and the incremental files. When the job has finished, the new output is written and the temporary folder and its files are removed. If the job does not finish successfully, CloudETL does a *rollback* simply by moving the files back to their original folder.

In fact processing, no update is done on existing fact data. The incremental fact data files are just added to the directory of the fact table.

5.2 An ETL Program Example

We now show how to use CloudETL to easily implement a parallel dimensional ETL program. The code in Figure 10 shows how to process the `pagedim` SCD. The implementation consists of four steps: 1) define the data source, 2) setup the transformations, 3) define the target table, and 4) add the sequence of transformations to the job manager and start. When the data source is defined, the schema information, the business key, and the date attribute to use in SCD processing are specified (lines 1–5). CloudETL provides some commonly used transformations (and the user can easily implement others) and lines 10–12 show transformations for excluding a field, adding the dimension key, and renaming a field, respectively. Lines 16–21 define the target, a type-2 SCD table parameterized by the name of a table, the names of attributes and their data types, and SCD attributes. CloudETL also offers other dimension and fact classes. Here, only one statement is needed to define a target while the complexity of processing SCDs is transparent to users. Finally, the transformer is added to the job manager which submits jobs to the Hadoop JobTracker (line 24).

```
0   /* 1) Define the data source of page dimension */
1   Reader  pagesReader = new CSVFileReader("/user/cloudetl/input/pages")
2                 .setField("url", DataType.STRING, FieldType.BKEY)
3                 .setField("size", DataType.INT)
4                 .setField("moddate", DataType.DATE, FieldType.SCD_DATE);
5
6   /* 2) Create the transform pipe, and add the  transformation
7         operators for cleansing data. */
8   TransformingReader pipe = new TransformingReader(pagesReader)
9               .add(new ExcludeFields("size"))
10                .add(new AddField("pageid", new Seq("pageid"), DataType.INT))
11                .add(new RenameField("moddate", "validfrom"));
12
13  /* 3) Define the target  dimension table */
14  Writer pagedim = new SlowlyChangingDimensionWriter("/user/cloudetl/output",
15                                                     "pagedim")
16                .setField("pageid", DataType.INT, FieldType.PKEY)
17                .setField("url", DataType.STRING, FieldType.LOOKUP)
18                .setField("version", DataType.INT, FieldType.SCD_VERSION)
29                .setField("validfrom", DataType.DATE, FieldType.SCD_VALIDFROM)
20                .setField("validto", DataType.DATE, FieldType.SCD_VALIDTO);
21
22  /* 4) Add transformer and start ETL */
23  JobPlanner.addTransformer(pipe, pagedim).start();
```

Figure 10: The ETL code for the SCD `pagedim`

This is much more difficult to do in Hive. As Hive does not support the UPDATEs needed for SCDs, we have to resort to a cumbersome and labour-intensive workaround including joins and heavy overwriting of tables to achieve the update effect. The HiveQL code is available in the appendix. CloudETL has much better programmer efficiency and only needs 4 statements (780 characters, a statement ended by ";"), while Hive uses 112 statements (4192 characters, including the statements for HiveQL and UDFs). The CloudETL code is thus an order of magnitude less complex.

The code in Figure 11 shows how to do the fact processing. Only 4 statements with 907 characters are needed. In Hive, we need 12 statements with 938 characters for the fact processing.

```
0  /* 1) Define the fact data source */
1  DataReader testResultsReader = new CSVFileReader("/user/cloudetl/input/testresults")
2                  .setField("localfile", DataType.STRING)
3                  .setField("url", DataType.STRING)
4                  .setField("lastmoddate", DataType.DATE)
5                  .setField("downloaddate", DataType.DATE)
6                  .setField("test", DataType.STRING)
7                  .setField("errors", DataType.INT);
8
9  /* 2) Do the necessary data transformation and look up dimension key values */
10 TransformingReader testresultsfactPipe = new TransformingReader(testResultsReader)
11   .add(new ExcludeFields("localfile"))
12   .add(new LookupTransformer("pageid", new SCDLookup(pagedim, "url", lastmoddate", -1)))
13   .add(new LookupTransformer("dateid", new Lookup(datedim, "downloaddate", -1)))
14   .add(new LookupTransformer("testid", new Lookup(testdim, "test", -1)));
15
16  /* 3) Define the target fact table */
17  DataWriter testresultsfact = new FactTableWriter("/user/cloudetl/fact", "testresultsfact")
18                  .setField("pageid", DataType.INT)
19                  .setField("dateid", DataType.INT)
20                  .setField("testid", DataType.INT)
21                  .setField("errors", DataType.INT);
22
23 /* 4) Add transformer and start ETL */
24 JobPlanner.addTransfer(testresultsfactPipe, testresultsfact).start();
```

Figure 11: The ETL code for fact processing

6 Experiments

In this section, we empirically evaluate the performance of CloudETL by studying 1) the performance of processing different DW schemas, including a star schema and schemas with an SCD and a big dimension table, and 2) the effect of the various optimization techniques applied to CloudETL, including the pre-updates in the mapper and co-location of data. We compare CloudETL with our previous work ETLMR [17] which is a parallel ETL programming framework using MapReduce. ETLMR is selected because CloudETL is implemented with the same goal as ETLMR and both make use of MapReduce to parallelize ETL execution. ETLMR is, however, designed for use with an RDBMS-based warehouse system. We also compare with Hive and the co-partitioning of Hadoop++ [8].

Cluster setup: Our experiments are performed using a local cluster of nine machines: eight machines are used as the DataNodes and TaskTrackers, each of them has two dual-core Intel Q9400 processors (2.66GHz) and 3GB RAM. One machine is used as the NameNode and JobTracker and it has two quad-core Intel Xeon E5606 (2.13GHz) processors and 20GB RAM. The disks of the worker nodes are 320GB SATA hard drives (7200rpm, 16MB cache, and 3.0Gb/s). Fedora 16 with the 64-bit Linux 3.1.4 kernel is used as the operating system. All machines are connected via an Ethernet switch with 1Gbit/s bandwidth.

We use Hadoop 0.21.0 and Hive 0.8.0. Based on the number of cores, we configure Hadoop to run up to four map tasks or four reduce tasks concurrently on each node. Thus, at any point in time, at most 32 map tasks or 32 reduce tasks run concurrently. The following configuration parameters are used: the sort buffer size is 512MB, JVMs are reused, speculative execution is turned off, the HDFS block size is 512MB, and the replication factor is 3. Hive uses the same Hadoop settings. For ETLMR, we use Disco 0.4 [7] as MapReduce platform (as required by ETLMR), set up the GlusterFS distributed file system (the DFS that comes with Disco) in the cluster, and use PostgreSQL 8.3 as the DW DBMS.

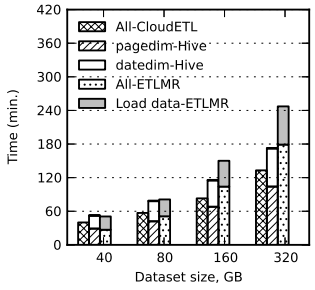

Figure 12: Star schema, no SCD

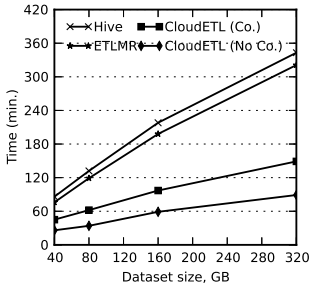

Figure 13: Init. load type-2 SCD

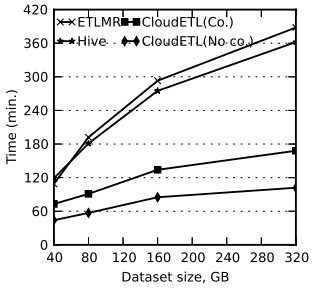

Figure 14: Incr. load type-2 SCD

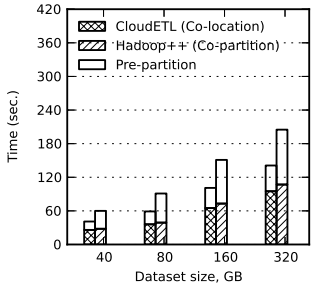

Figure 15: Proc. type-2 SCD

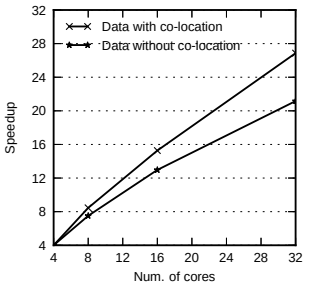

Figure 16: Speedup of dim. load

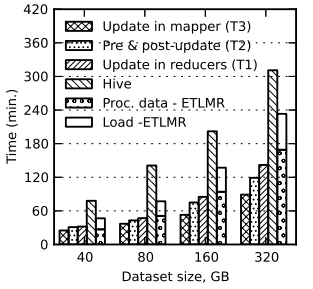

Figure 17: Proc. type-1 SCD

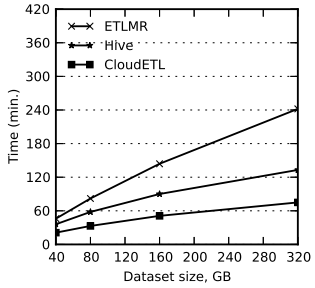

Figure 18: Proc. facts, small dims, SCD

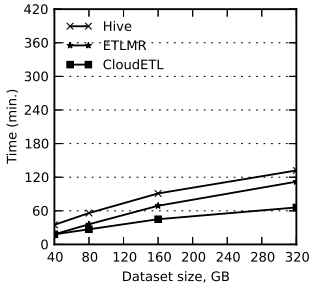

Figure 19: Proc. facts,small dims,no SCD

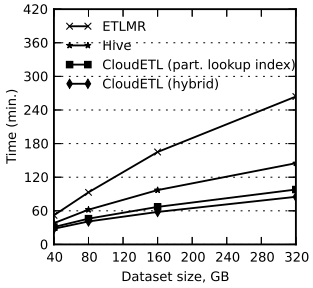

Figure 20: Proc. facts, big dims, SCD

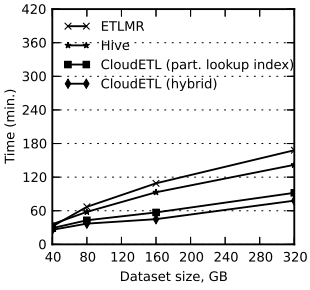

Figure 21: Proc. facts, big dims, no SCD

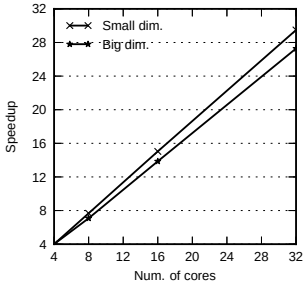

Figure 22: Speedup of fact processing

Data sets: We use generated data sets for the running example and consider the star schema in Figure 2. In the experiments, the `pagedim` and `datedim` dimensions get data from the same source data set which we scale from 40GB to 320GB. Every 10GB of source data result in 1.85GB `pagedim` dimension data (113,025,455 rows) and 1.01MB `datedim` dimension data (41,181 rows). The `testdim` dimension has its own source data set with a fixed size of 32KB (1,000 rows). The fact source data set is also scaled from 40GB to 320GB. Every 10GB of source data result in 1.45GB fact data (201,233,130 rows) in the DW. The reason that the size of the loaded data is smaller is that the source data contains more redundant data like complete URLs for each test.

6.1 Dimension Data Processing

Simple star schema: In the first experiments, we process data into the dimension tables of a star schema without any SCD updates. To make Hive support dimension processing, we implement a number generator similar to CloudETL's to generate the dimension key values and use a user-defined function (UDF) to get the numbers. We use all 32 tasks to process the data and scale the data from 40GB to 320GB. We measure the time from the start to the end of each run.

Figure 12 shows the results. CloudETL processes the three dimension tables within one job and does data transformations in mappers. The data for a dimension table is collected and written to HDFS in a reducer. Hive, however, has to process the statements for different dimension tables in different jobs. The total time used by Hive is *up to 28% higher* than the time used by CloudETL (the time for `testdim` is not shown in Figure 12 since it is negligible). During the tests, we observe that Hive can employ map-only jobs to process `pagedim` and `testdim`, but has to use both map and reduce to process `datedim` since `datedim` requires the DISTINCT operator to find duplicate records. ETLMR uses its so-called offline dimension scheme in which the data is first processed and stored locally on each node, then collected and loaded into the DW by the DBMS bulk loader (PostgreSQL's COPY is used in this experiment). As shown, ETLMR is efficient to process relatively small-sized data sets, e.g., 40GB, but the time grows fast when the data is scaled up and ETLMR uses about 81% more time than CloudETL for 320GB.

Processing a type-2 SCD: We now study the performance when processing the big dimension table `pagedim` which is a type-2 SCD (see Figure 13 and Figure 14). We test both initial load ("init.") and incremental load ("incr.") in this experiment. In an initial load, `pagedim` is cleared before a job starts. In an incremental load, 320GB source data is already loaded into `pagedim` before the job starts. For CloudETL, the initial and incremental loads are both tested using data with and without co-location. Figure 13 and Figure 14 show the results of the initial and incremental loads, respectively. The results show that data co-location improves the performance significantly and between 60% and 73% more time is used when there is no co-location. This is because the co-located data can be processed by a map-only job which saves time. CloudETL outperforms ETLMR. When the data is scaled to 320GB, ETLMR uses up to 3.8 times as long for the load. The processing time used by ETLMR grows faster which is mainly due to the database-side operation called post-fixing [17] used to set SCD attribute values correctly. CloudETL also outperforms Hive significantly. For example, when tested using 320GB data, Hive uses up to 3.9 times as long for the initial load while for the incremental load, it uses up to 3.5 times as long. This is because the workaround to achieve the update effect for the SCD handling requires several sequential jobs (4 jobs for the initial load and 5 jobs for the incremental load as shown in the appendix).

Compared with Hadoop++: We now compare with Hadoop++ for incremental load of `pagedim`. Hadoop++ co-partitions two data sets by adding a "Trojan" join and index through MapReduce jobs. The lines (from the two data sets) with identical join key values are put into the same data split and the same data node. We change our program to read the co-partitioned data splits, and to run a map-only job. Figure 15 shows the test results. The results include the times for prepartitioning the data sets and indicate that Hadoop++ uses about 2.2 times as long as CloudETL to partition the same data set. Processing the co-partitioned data also takes longer, 8%–14% more time. We found that the Hadoop++ co-partitioning is much more tedious and has jobs for the following tasks: converting textual data to binary, co-partitioning, and creating the index. In addition, for an incremental load, Hadoop++ has to rebuild the index from scratch which is increasingly expensive when the data amounts grow. The co-location of CloudETL, however, makes use of a customized block placement policy to co-locate the data. It is very light-weight and more suitable for incremental load.

Processing a type-1 SCD: We now process `pagedim` as a type-1 SCD and do the following three tests: T_1) do the type-1 updates in the reducers; T_2) first do "pre-updates" in the mappers, then do "post-updates" in the reducers; and T_3) first partition the source data, co-locate the partitioned files, and then do map-only

updates. The results are shown in Figure 17. The map-only updates (T_3) are the fastest followed by pre- and post-updates (T_2) which use between 16% and 42% more time. Updates in the reducers (T_1) use between 28% and 60% more time. The ETLMR offline dimension scheme supports type-1 SCDs by processing data on MapReduce and then loading the processed data into the DW (see also the discussion of Figure 12). It uses more time to process the scaled data, e.g., the time for 320GB is 16%, 42% and 90% more than that of T_1, T_2 and T_3, respectively. Hive requires 4 jobs to process the type-1 SCD and takes 3.5 times longer than CloudETL.

Speedup: We now study the speedup by varying the number of cores from 4 to 32. We do the speedup tests using 320GB `pagedim` data with and without co-location, respectively, i.e., the tests are done when only map is used, and when map and reduce both are used. Figure 16 shows the speedup lines of both tests. The results indicate that the speedup with data co-location is 27 times for 32 cores and is close to linear, and better than without co-location. In other words, loading co-located data can achieve better speedup since a map-only job is run for the data. The slight sub-linearity is mainly due to the communication cost as well as task setup overheads on Hadoop.

6.2 Fact Data Processing

We now study the performance of fact processing. Fact processing includes doing data transformations and looking up dimension key values. We load fact data into `testresultsfact` in this experiment and use both small and big dimension tables by varying the size of `pagedim`. With the small dimension table, the lookup indices are used and cached in main memory for multi-way lookups. The lookup index sizes are 32KB (`testdim`), 624KB (`datedim`), 94MB `pagedim` (as a traditional dimension, i.e., not an SCD) and 131MB `pagedim` (as a type-2 SCD). They are generated from 2GB dimension source data. For ETLMR, we use its offline dimension scheme when `pagedim` is used as a non-SCD. This is the ETLMR scheme with the best performance [17]. When `pagedim` is used as an SCD, the online dimension scheme is used for comparison. This scheme retrieves dimension key values from the underlying RDBMS. For Hive, we join the fact data with each of the dimension tables to retrieve dimension key values. Figures 18 and 19 present the results from using the small dimension tables with and without SCD support, respectively. The comparison of the results in two figures shows that CloudETL (without SCD support) has the highest performance while the processing with an SCD uses about 5%–16% more time than without an SCD. CloudETL outperforms both ETLMR and Hive when using small dimension tables since a map-only job is run and in-memory multi-way lookups are used. In contrast, Hive requires four sequential jobs (an additional job is used for projection after getting the dimension key values) and uses up to 72% more time. ETLMR with SCD support takes about 2.1 times longer than CloudETL (see Figure 18) due to the increasing cost of looking up dimension key values from the DW. ETLMR without SCD support also runs a map-only job, and its performance is slightly better when processing the relatively small-sized data (see Figure 19), e.g., 40GB, but the time grows faster when the data is scaled.

We now study the performance with big dimension tables. The dimension values in the big table `pagedim` is generated from 40GB source data. We use the following two approaches for the lookups. The first is the hybrid solution where a Hive join is used to retrieve the key values from the big dimension table and then multi-way lookups are used to retrieve the key values from the small dimension tables (the lookup index sizes are 4.5MB for `datedim` and 32KB for `testdim`). The other is the partitioned lookup-index solution. We assume that the fact source data has already been partitioned. Each mapper caches only one partition of the big lookup index in addition to the two small lookup indices. A partition of the fact data is processed by the mapper which caches the lookup index partition that is relevant to the fact partition. A map-only job is run to do multi-way lookups. Figure 20 and 21 show the results with and without SCD support, respectively. As shown, CloudETL again outperforms both Hive and ETLMR. When the hybrid solution is used, CloudETL is more efficient than when the partitioned big lookup index is used.

The partitioned lookup-index solution requires between 11% and 18% more time. Hive and ETLMR do not scale as well and, e.g, when there is an SCD, they require 35% and 85% more time, respectively, to process 40GB source data while they require 71% and 211% more time, respectively, for 320GB.

We now study the speedup when scaling up of the number of the nodes. Figure 22 shows the speedup when processing up to 320GB fact source data using small dimension tables and big dimension tables (using the hybrid solution). As shown, CloudETL achieves nearly linear speedup in both cases. For 32 cores, the speedup is 27.5 times for big dimension tables and 29 times for small dimension tables. A reason for the difference is that the hybrid solution requires an additional Hive job for the big dimension.

In summary, CloudETL is the fastest solution in all the experiments. For a simple star schema, Hive uses up to 28% more time while ETLMR uses 81% more time. When a type-2 SCD is processed, Hive uses up to 3.9 times as long as CloudETL. Hadoop++ has a performance which is closer to CloudETL's, but CloudETL remains faster. The experiments have also shown that the co-location used by CloudETL has a positive performance impact and that CloudETL's scalability is close to linear.

7 Related Work

To tackle large-scale data, parallelization is the key technology to improve the scalability. The MapReduce paradigm [5] has become the de facto technology for large-scale data-intensive processing due to its ease of programming, scalability, fault-tolerance, etc. Multi-threading is another parallelization technology which has been used for a long time. Our recent work [22] shows multi-threading is relatively easy for ETL developers to apply. It is, however, only effective on Symmetric Processor Systems (SMP) and does not scale out on many clustered machines. Thus, it can only achieve limited scalability. The recent parallelization systems Clustera [6] and Dryad [11] support general cluster-level computations to data management with parallel SQL queries. They are, however, still only available as academic prototypes and remain far less studied than MapReduce. Like MapReduce, they are not ETL-specific.

Stonebraker et al. compare MapReduce with two parallel DBMSs (a row-store and a column-store) and the results show that the parallel DBMSs are significantly faster than MapReduce [20, 19]. They analyze the architectures of the two system types, and argue that MapReduce is not good at query-intensive analysis as it does not have a declarative query language, schema, or index support. Olston et al. complain that MapReduce is too low-level, rigid, hard to maintain and reuse [18]. In recent years, HadoopDB [1], Aster, Greenplum, Cloudera, and Vertica all have developed hybrid products or prototypes by using two class systems which use both MapReduce and DBMSs. Other systems built only on top of MapReduce while providing high-level interfaces also appear, including Pig [18], Hive [25], and Clydesdale [14]. These systems provide MapReduce scalability but with DBMS-like usability. They are generic for large-scale data analysis and processing, but not specific to dimensional ETL where a DW must be loaded. For example, they do not support SCD updates or simple commands for looking up key values dimension members and inserting them if not found. In contrast, CloudETL provides built-in support for processing different dimensional DW schemas including SCDs such that the programmer productivity is much higher. Our previous work ETLMR [17] extends the ETL programming framework pygrametl [23] to be used with MapReduce. ETLMR, however, is built for processing data into an RDBMS-based DW and some features rely on the underlying DW RDBMS, such as generating and looking up dimension key values. CloudETL, on the other hand, provides built-in support of the functionality needed to process dimensions and facts. ETLMR uses the RDBMS-side operation "post-fixing" to repair inconsistent data caused by parallelization, while CloudETL solves this issue by exploiting global key value generation and SCD updates on Hadoop. The recent project Cascading [4] is able to assemble distributed processes and plan them to run in a Hadoop cluster. The workflow of Cascading is somewhat similar to the transformers of CloudETL. However, Cascading does not consider DW-specific data processing such as key generation, lookups, and processing star schemas and SCDs.

The co-location of data in CloudETL is similar to the co-location in CoHadoop [9]. CoHadoop is, however, a general extension to Hadoop that requires applications to explicitly co-locate files by using a common identifier (a "locator") when a file is created. Instead of changing Hive to do that, we in CloudETL exploit how files are named and define co-location by means of a regular expression. Further, this is fully implemented by using HDFS's standard block placement mechanism such that CloudETL requires no changes of the Hadoop/HDFS code. HadoopDB [1] and Hadoop++ [8] do also co-locate data. HadoopDB does so by using RDBMS instances instead of HDFS. Hadoop++ considers the entire data set when doing co-location which is impractical when incremental data is added.

Pig and Hive provide several join strategies in terms of the features of the joined data sets. HadoopDB [1] is a hybrid solution that uses both Hadoop and DBMS instances. It pushes joins to the DBMSs on each node. The join algorithms for MapReduce are compared and extensively studied in [12, 3]. The join implementations above process a join operation within one MapReduce job, which causes a non-trivial cost. To address this issue, [2, 13] propose multi-way joins which shuffle the joining of data to reducers in a one-to-many fashion and do the joins on the reduce side. This, however, becomes expensive if there are many tables to join. In contrast, CloudETL does "multi-way lookups" (similar to the multi-way joins) in map side when processing fact data and only a minimal amount of data is saved in the lookup index files and used for joins. This is much more efficient for our particular purpose.

8 Conclusion and Future Work

With the ever-growing amount of data, it becomes increasingly challenging for data warehousing technologies to process the data in a timely manner. This paper presented the scalable dimensional ETL framework CloudETL. Unlike traditional ETL systems, CloudETL exploits Hadoop as the ETL execution platform and Hive as the warehouse system. CloudETL provides built-in support of high-level ETL-specific constructs for common, dimensional DW schemas including star schemas and SCDs. The constructs facilitate easy implementation of parallel ETL programs and improve programmer productivity very significantly. In particular, it is much easier to handle SCDs since CloudETL can perform the necessary updates (i.e., overwrites for type-1 SCDs and updates of validity dates and version numbers for type-2 SCDs). When CloudETL is not used, this requires extensive programming and overwriting of tables.

In the paper, we presented an approach for efficient processing of updates of SCDs in a distributed environment. We proposed a method for processing type-1 SCDs which does pre-updates in mappers and post-updates in reducers. We also presented a block placement policy for co-locating the files in HDFS to place data to load such that a map-only job can do the load. In fact processing, we proposed to use distributed lookup indices for multi-way lookups to achieve efficient retrieval of dimension key values. We conducted extensive experiments to evaluate CloudETL and compared with ETLMR and Hive. The results showed that CloudETL achieves better performance than ETLMR when processing different dimension schemas and outperforms the dimensional ETL capabilities of Hive: It offers significantly better performance than Hive and it requires an order of magnitude less code to implement parallel ETL programs for the schemas. In an experiment, Hive used 3.9 times as long as CloudETL and required 112 statements while CloudETl required only 4. We also compared CloudETL with Hadoop++ and found that CloudETL was faster.

There is a number of future research directions for this work. First, we plan to make CloudETL support more ETL transformations. Second, it would be interesting to consider more efficient backends, such as Spark [24], for fast ETL processing. Last, it would be good to create a graphical user interface (GUI) where users can "draw" an ETL flow by using visual transformation operators and still get a highly parallel ETL program.

Acknowledgments We would like to thank Dr. Jorge-Arnulfo Quiané-Ruiz for sharing the Hadoop++ source code with us.

References

[1] A. Abouzeid et al. “HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads”. *PVLDB* 2(1):22–933, 2009.

[2] F. N. Afrati and J. D. Ullman. “Optimizing Joins in a Map-reduce Environment”. In *Proc. of EDBT*, pp.99-110, 2010

[3] S. Blanas et al. “A Comparison of Join Algorithms for Log Processing in MapReduce”. In *Proc. of SIGMOD*, pp.975–986, 2010.

[4] “Cascading”. `http://www.cascading.org` as of 2013-05-16.

[5] J. Dean and S. Ghemawat. “Mapreduce: Simplified Data Processing on Large Clusters”. *CACM* 1(51):107–113, 2008.

[6] D. DeWitt et al. “Clustera: An Integrated Computation and Data Management System”. *PVLDB* 1(1):28–41, 2008.

[7] “Disco”. `http://discoproject.org` as of 2013-07-15.

[8] J. Dittrich et al. “Hadoop++: Making a Yellow Elephant Run Like a Cheetah”. *PVLDB* 3(1):518–529, 2010.

[9] M.Y. Eltabakh et al. “CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop”. *PVLDB* 4(9):575–585, 2011.

[10] “Hadoop”. `http://hadoop.apache.org/` as of 2013-07-15.

[11] M. Isard et al. “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks”. In *Proc. of EuroSys*, pp. 59–72, 2007.

[12] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. “The Performance of MapReduce: An In-depth Study”. *PVLDB* 3(1):472–483, 2010.

[13] D. Jiang, A. K. H. Tung, and G. Chen. “Map-join-reduce: Towards Scalable and Efficient Data Analysis on Large Clusters”. *TKDE* 23(9)1299–1311, 2011.

[14] T. Kaldeway, E. J. Shekita, and S. Tata. “Clydesdale: structured data processing on MapReduce”. In *Proc. of EDBT*, pp. 15–25, 2012.

[15] R. Kimball and M. Ross. “The Data Warehouse Toolkit”. John Wiley and Son, 1996.

[16] J. Lin and C. Dyer. “Data-Intensive Text Processing with MapReduce”. Morgan & Claypool Publishers, 2010.

[17] X. Liu, C. Thomsen, and T. B. Pedersen, “ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce”. In *Proc. of Dawak*, 2011.

[18] C. Olston et al. “Pig Latin: A Not-so-foreign Language for Data Processing”. In *Proc. of SIGMOD*, pp. 1099–1110, 2008.

[19] A. Pavlo et al. “A Comparison of Approaches to Large-scale Data Analysis”. In *Proc. of SIGMOD*, pp. 165–178, 2009.

[20] M. Stonebraker et al. “MapReduce and Parallel DBMSs: friends or foes?”. *CACM*, 53(1):64–71, 2010.

[21] C. Thomsen and T. B. Pedersen. “Building a Web Warehouse for Accessibility Data”. In *Proc. of DOLAP*, pp. 43–50, 2006.

[22] C. Thomsen and T. B. Pedersen. “Easy and Effective Parallel Programmable ETL”. In *Proc. of DOLAP*, pp. 37–44, 2011.

[23] C. Thomsen and T. B. Pedersen. “pygrametl: A Powerful Programming Framework for Extract-Transform-Load Programmers”. In *Proc. of DOLAP*, pp. 49–56, 2009.

[24] Spark Project, `http://spark-project.org/` as of 2013-07-15.

[25] A. Thusoo et al. “Hive: A Warehousing Solution Over a Map-reduce Framework”. *PVLDB* 2(2):1626–1629, 2009.

[26] “TPC-H”. `http://tpc.org/tpch/` as of 2013-07-15.

Appendix

A Process Type-2 SCDs Using HiveQL

A.1 Initial Load

```
-- 1. Create the type-2 SCD table:

CREATE TABLE IF NOT EXISTS pagescddim(pageid INT, url STRING, serverversion STRING, size INT,
validfrom STRING, validto STRING, version INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n' STORED AS TEXTFILE;

-- 2. Create the data source table and load the source data:

CREATE TABLE pages(localfile STRING, url STRING, serverversion STRING, size INT, downloaddate
STRING, moddate STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '/q/disk_0/xiliu/dataset/pages40GB.csv' OVERWRITE INTO TABLE pages;

-- 3. Create UDFs, add the UDFs jars to the classpath of Hive, and create the temporary
--    functions for the UDFs:

// Create the UDFs for reading dimension key values from the global sequential number generator.
package dk.aau.cs.hive.udf;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;

public final class SeqUDF extends UDF {
        enum ServerCmd {BYE, READ_NEXT_SEQ}
        String seqName = null;
        byte[] nameInBytes;
        IntWritable curSeq = new IntWritable();
        IntWritable endSeq = new IntWritable();

        SocketChannel channel;
        ByteBuffer buffer = ByteBuffer.allocate(512);
        Configuration conf;
        static final String hostname = "localhost";
        static final int port = 9250;
        final IntWritable delta = new IntWritable(10000);

    public IntWritable evaluate(final Text name) {
          if (name.toString().equalsIgnoreCase("close")){
               this.cleanup();
               return new IntWritable(-1);
          }
          try {
               if (seqName == null) {
                    this.seqName = name.toString();
                    this.nameInBytes = SeqUDF.getBytesUtf8(this.seqName);
                    this.setup();
               }
               return new IntWritable(this.nextSeq());
          } catch (Exception e) {
               e.printStackTrace();
          }
          return new IntWritable(-1);
```

```
    }

    private void setup() {
        try {
            this.channel = SocketChannel.open(new InetSocketAddress(hostname, port));
            this.curSeq.set(this.readNextFromServer());
            this.endSeq.set(curSeq.get() + delta.get());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private int readNextFromServer() throws IOException {
        buffer.clear();
        buffer.putInt(ServerCmd.READ_NEXT_SEQ.ordinal())
              .putInt(nameInBytes.length).put(nameInBytes).flip();
        channel.write(buffer);

        buffer.clear();
        channel.read(buffer);
        return buffer.getInt(0);
    }

    private int nextSeq() throws IOException {
        if (curSeq.get() >= endSeq.get()) {
            this.curSeq.set(readNextFromServer());
            this.endSeq.set(curSeq.get() + delta.get());
        }
        int ret = curSeq.get();
        curSeq.set(ret+1);
        return ret;
    }

    private void cleanup() {
        try {
            buffer.clear();
            buffer.putInt(ServerCmd.BYE.ordinal()).flip();
            channel.write(buffer);
            channel.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static byte[] getBytesUtf8(String string) throws UnsupportedEncodingException {
        if (string == null) {
            return null;
        }
        return string.getBytes("UTF-8");
    }
}

// Create the UDF for generating version IDs for SCDs
package dk.aau.cs.hive.udf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;

public final class VersionIDUDF extends UDF {
    IntWritable curID = new IntWritable();
    Text bkey = new Text();

    public IntWritable evaluate(final Text bkey) {
        if (bkey.toString().equalsIgnoreCase(this.bkey.toString())) {
            int curVersionID = curID.get();
            IntWritable ret = new IntWritable(curVersionID);
            curID.set(curVersionID + 1);
```

```
            return ret;
        } else {
            IntWritable ret = new IntWritable(1);
            curID.set(2);
            this.bkey.set(bkey);
            return ret;
        }
    }
}

-- Add the UDF jar package to the classpath of Hive
ADD jar file:///hive/hiveudfs.jar;

-- Create the temporary functions in Hive for UDFs
CREATE TEMPORARY FUNCTION nextseq AS 'dk.aau.cs.hive.udf.SeqUDF';
CREATE TEMPORARY FUNCTION versionid AS 'dk.aau.cs.hive.udf.VersionIDUDF';

-- 4. Load data from the source into the type-2 pagescddim table.
--    This requires several intermediate steps.

CREATE TABLE pagestmp1 (pageid INT, url STRING, serverversion STRING, size INT,
validFROM STRING, validto STRING, version INT) ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t' STORED AS TEXTFILE ;

CREATE TABLE pagestmp2 (pageid INT, url STRING, serverversion STRING, size INT,
validfrom STRING, validto STRING, version INT) ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t' STORED AS TEXTFILE ;

-- Sort by SCD date for each group
INSERT OVERWRITE TABLE pagestmp1 AS SELECT nextseq('pagedim_id') AS pageid, url,
serverversion, size, moddate AS validfrom, NULL AS validto, NULL AS version
FROM pages ORDER BY url, moddate ASC;

-- Add the version number
INSERT OVERWRITE TABLE pagestmp2 AS SELECT pageid, url, serverversion, size,
validfrom, validto, versionid(url) AS version FROM pagestmp1;

DROP TABLE pagestmp1;

-- Update the validto attribute values, 1 Job
INSERT OVERWRITE TABLE pagescddim SELECT a.pageid, a.url, a.serverversion, a.size,
a.validfrom, b.validfrom AS validto, a.version FROM pagestmp2 a  LEFT OUTER JOIN
pagestmp2 b ON (a.version=b.version-1 AND a.url=b.url);

DROP TABLE pagestmp2;
```

A.2 Incremental Load

```
CREATE TABLE pagestmp3 (pageid INT, url STRING, serverversion STRING, size INT,
validfrom STRING, validto STRING, version INT) ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t' STORED AS TEXTFILE ;

INSERT OVERWRITE TABLE pagestmp3 SELECT * FROM (SELECT pageid, url, serverversion, size,
validfrom, validto, version FROM pagescddim UNION ALL SELECT pageid, url, serverversion,
size, validfrom, validto, version FROM pagestmp2) a ORDER BY a.url, a.validfrom ASC;

CREATE TABLE pagestmp4 AS SELECT pageid, url, serverversion, size, validfrom, validto,
versionid(url) AS version FROM pagestmp3;

DROP TABLE pagestmp3;

INSERT OVERWRITE TABLE pagescddim SELECT a.pageid, a.url, a.serverversion, a.size,
a.validfrom, b.validfrom as validto, a.version FROM pagestmp4 a  LEFT OUTER JOIN
pagestmp4 b  ON (a.version=b.version-1 AND a.url=b.url);

DROP TABLE pagestmp4;
```

B Process Facts Using HiveQL

```
-- 1. Create fact table:
CREATE TABLE IF NOT EXISTS testresultsfact(pageid INT, testid int, dateid int, error int)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES
TERMINATED BY '\n' STORED AS TEXTFILE;

-- 2. Create the data source table and load the source data:
CREATE TABLE testresults(localfile STRING, url STRING, serverversion STRING, test STRING,
size INT, downloaddate STRING, moddate STRING, error INT) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE ;

LOAD DATA LOCAL INPATH '/q/disk_0/xiliu/dataset/testresults.csv'
OVERWRITE INTO TABLE testresults;

-- 3. Filter the uncessary attribute values and look up dimension key values:
CREATE TABLE testresults_tmp1 AS SELECT b.pageid, downloaddate, serverversion,
test, error FROM testresults a LEFT OUTER JOIN pagedim b
ON (a.url=b.url AND moddate>=validfrom AND moddate<validto);

CREATE TABLE testresults_tmp2 AS SELECT a.pageid, b.testid, a.serverversion, a.downloaddate,
a.error FROM testresults_tmp1 a LEFT OUTER JOIN testdim b ON (a.test = b.testname);

DROP TABLE testresults_tmp1;

CREATE TABLE testresults_tmp3 AS SELECT a.pageid, b.testid, a.dateid, a.error
FROM testresults_tmp2 a LEFT OUTER JOIN datedim b ON (a.downloaddate = b.date);

DROP TABLE  testresults_tmp2;

CREATE TABLE testresults_tmp4 AS SELECT * FROM (SELECT * FROM testresults_tmp3
UNION ALL SELECT pageid, testid, dateid, error FROM testresultsfact);

DROP TABLE  testresults_tmp3;

INSERT OVERWRITE TABLE testresultsfact AS SELECT * from testresults_tmp4;

DROP TABLE testresults_tmp4;
```