Delta View Generation for Incremental Loading of Large Dimensions in a Data Warehouse

Igor Mekterović*, Ljiljana Brkić *
* University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia
igor.mekterovic@fer.hr
ljiljana.brkic@fer.hr

Abstract - Incremental load is the preferred approach in efficient ETL processes. Fact tables are the ones who benefit the most from this approach, since they are large in terms of row count. For the sake of simplicity, dimension tables are often ignored and populated in a full reload manner. However, big dimensions (e.g. Client) can also have a significant impact on the ETL process and should also be considered for incremental load. Although they have much smaller cardinality than a typical fact table, it usually takes much more resources to calculate one dimension table row than to calculate one fact table row. Large dimension tables are based on multiple source tables, and it is not trivial to determine the changed records that should be considered for the incremental load because changes in any and all of underlying source tables must be considered. In this paper, we present an algorithm for the dimension’s delta view generation. Delta view for a dimension encompasses all its source tables and produces a set of keys (e.g. ClientIds) that should be incrementally processed. We have employed this approach in a real world project and have noticed a significant reduction in the loading time of big dimensions.

I. INTRODUCTION

While OLTP systems are focused on supporting operational business processes requiring short response time, the intention of data warehouses (DW) is strategic planning and decision-making based on data integrated from heterogeneous systems. Multidimensional database technology is the key factor in interactive analysis of large amounts of data for decision-making. A multidimensional data model is commonly used in data warehouses that integrate enterprise’s data from several sources into large repositories. The data acquisition process represents one of the most technically demanding and one of the most expensive parts of building a data warehouse. The data needed for the data warehouse are being extracted from their sources during an extract, transfer and load (ETL) process. According to [1], ETL design and development work consumes 60 to 80 percent of an entire BI project, therefore it is essential to optimize this layer of BI environment. Typical DW architecture includes a data staging area where the extracted data are stored temporarily, and subsequently, upon cleaning and transformation phase, transferred into the data warehouse. Commonly, there is no real-time connection between a DW and its data sources, since the massive read in decision support systems would conflict with the continuous update of operational systems and result in poor response times.

After the first or initial loading, DW must be periodically refreshed to keep up with source data updates. DW loading is usually done periodically, by a background process. The update patterns (daily, weekly, etc.) for traditional data warehouses and data integration process result in outdated data to a greater or lesser extent [3]. The naive approach to refreshing a data warehouse is referred to as full reloading. Possible refreshment scenario is to repeat the initial loading process using modified source data, compare the results with the current DW in order to determine changes that need to be done and finally perform the changes. This strategy is known as full DW reload. With the increasing size and complexity of DW, full reloading becomes inadequate, and in some cases inapplicable. More appropriate approach is a gradual change of DW in accordance with the changes that have occurred in the data sources since the last synchronization. Only the data that has changed since the previous reload needs to be transformed and processed. This approach is known as incremental reloading. The volume of modified data is always small compared to the entire dataset, thus, incremental reloads can be assumed to work quicker and more efficiently than full reloads. In most cases, the content of a DW is updated once a day. However, there are businesses such as health systems, e-business, stock brokering and others where such level of freshness of data is not sufficient, since they operate in a business time schedule of 24x7 and need relevant information as fast as possible in order to react in a near real-time manner. Real-Time Data Warehousing refers to an approach where DWs are updated as frequently as possible, greatly reducing data accuracy and availability gap.

The basic assumption of incremental and real-time loading is that the changes of source data can be captured and later on propagated to the data warehouse. There is a number of known techniques for changed data capture. The most commonly used ones are discussed in chapter III.

II. RELATED WORK

Incremental warehouse reloading is evidently related to incremental maintenance of materialized views [2] [3] as both areas cope with same issue – how to update physically integrated data under a given time constraint [4]. The core idea of materialized view incremental update stated in [4] assumes computing new content using materialized parent view (the outdated view needed to be refreshed) and its incremental update, often referred to as
delta view. Finding tuples constituting the delta view is addressed in a number of papers [5] [6] dealing with so called CDC techniques. This concept of capturing information about changed tuples is valuable later in the extract phase of incremental ETL process [7] [8]. Work described in [6] is to some extent similar to ours, but we go further with optimization of extract phase. The authors presented a tool for automatic creation of all necessary structures for altered data capture and subsequent extraction. They track changes by storing changed tuples identifiers (primary keys) in a special CDC tables and later on use stored identifiers for fast retrieval of tuples relevant for the next ETL cycle. Complementary to their work we addressed the issue, and proposed the solution, for finding identifiers of tuples responsible for refreshment of a particular dimension table. Since the contents of denormalized dimensions are refreshed based on the contents of numerous source tables, finding such identifiers can be a challenging task.

III. CHANGE DATA CAPTURE TECHNIQUES IN DBMS

Change Data Capture (CDC) is a generic term for techniques that monitor data sources in order to detect and capture necessary data changes [9] [10]. Different CDC techniques are used in practice. All CDC approaches assume that a complete changed record will be available to the requesting target.

The following is an overview of three most popular CDC techniques applicable in DBMS environment.

A. Timestamp based Capture

Timestamp based data capture is performed by adding a timestamp column (specifying the time and date when observed row was last modified) to each table of interest. These timestamps provide the selection criteria for the capture of changed records. The need to change the structure of each table of interest, by adding a single timestamp column, is the first disadvantage of this approach.

Second, it is not possible to capture intermediate states of data. If a record changed state more than once since the last ETL run, only the current (precisely the last) state of the respective record would be at disposal. Accordingly, when using this method, the inserted records cannot be distinguished from updated ones.

An additional problem with this technique is a lack of possibility to detect deleted records. Possible solution to detecting deletions is to label these records as inactive until they have been processed [10]. This concept leads to other problems in operational systems. For example, all applications working with operational data have to treat inactive records as deleted ones.

This method causes frequent writes of timestamp attributes on the operational systems and frequent reads whenever ETL process runs. Indexing timestamps can speed up reading but on the other hand can slow down INSERT and UPDATE operations [11].

B. Trigger-Based Capture

This approach involves the usage of triggers to track changes in the source database. Whenever a modification takes place in the source database corresponding log record is written into a dedicated log table. These dedicated log tables serve to determine rows changed since the last ETL processing.

The log tables usually contain a timestamp column that provides the exact time and date when a given row was last modified. The statements that are executed when trigger is fired, maintain the timestamp column accurate i.e. update the timestamp column with the current time.

This technique assumes that the DBMS supports active behavior i.e. triggers. The set and granularity of the events and conditions that cause a trigger to fire are database specific.

This approach enables capturing all types of events (INSERT, UPDATE, DELETE, MERGE). For UPDATEed rows both before and after image can be available.

One shortcoming of this approach is that it might significantly affect performance on the source systems. For example, if all INSERTs, UPDATEs and DELETEs are captured, this method creates twice the workload for the database [10]. This technique imposes a considerable operational overhead at the source system. This impact should be carefully considered prior to implementation on a production source system.

C. Transaction Log Capture

This technique employs the logging and recovery capabilities of a DBMS [9]. Since transaction logs are utilized by DBMS to store transaction information for logging and recovery, these logs contain all information required to capture changed data. A specialized vendor-specific application must be written to monitor the log files and capture the data of interest.

One of the shortcomings of this approach is that it assumes that the transaction logs remain available until the changes of interest are captured. In order to prevent transaction loss, the database administrator must constantly monitor the transaction log area. The problem can occur when the recycling of the transaction log takes place, that is, when a part of the transaction log, containing transactions already committed, is reused by the DBMS [12].

The log capture method is probably the most efficient approach to incremental capture due to the fact that log writing is already well optimized and available on most DBMS platforms today [10].

IV. DELTA VIEWS

Once we know what records have changed in the source system, we can use that information to incrementally populate fact and dimension tables in the data warehouse. Incremental loading of fact tables is rather trivial and will not be further discussed. Fact tables are also the ones that benefit most from the incremental loading style since they typically contain a huge number of records. On the other
hand, dimension tables are usually relatively small and can be populated in a full reload manner. However, there are cases when dimension tables are also fairly large, both in terms of number of records and number of attributes. Such dimensions are sometimes referred to as “monster dimensions”, “whose size exceed 100 million rows” [13]. Typically, those are customer or client dimensions in large enterprises. In our case, our client dimension is of 10^5 order with more than a hundred attributes. Such dimensions may take a long time to load, since they are both large in record count and have a considerable number of attributes that need to be calculated along the way (e.g. student’s Grade Point Average, rank, etc.). We believe that, for such large dimensions, it is beneficial to construct the “delta view” – a view that will contain keys of records that need to be addressed in the ETL procedure. Delta view contains the keys of deleted, updated and inserted records of the relational source table, and that information is then used in accordance with the dimension type to update the dimension. For instance, for type 1 SCD (Slowly Changing Dimension), one would simply delete all dimension records that are members of delta view, and then reload those updated delta view records as well as the new ones. Type 2 SCD would handle updates and inserts differently, etc.

Unlike fact tables, delta view records are deduced not only from the base source table (for the dimensions), but also from many other tables that are being used in the dimension’s ETL process. In our, real-world case, our dimension table is based on 23 relational tables. A change in any of those 23 tables can impact the content of the dimension. In the following chapter, we elaborate on the prerequisites and algorithm for the delta view construction.

A. Delta view construction

Algorithm for the delta view construction will be explained with the help of the running example. Figure 1 shows a simplified version of the relational database subset used to populate dStudent dimension table:

For the sake of simplicity, all attributes except primary and foreign keys have been removed. Student enrolls in 0..N years using one of the supported financial models (scholarship, self-funded, etc.). Also, for each academic year we keep track of current city (and state) of residence and student’s health insurance. CityName is specified in different word grammatical cases (in Croatian language there are seven different grammatical cases) because different cases are needed for different contexts (in various reports). Similar constructs appear in multilingual databases where a name for an entity is specified in various languages.

From this model, various dimension attributes are calculated. Some are trivial (e.g. dStudent.currCityName), while others have to be calculated with stored procedures (e.g. a flag indicating whether student repeated/failed any academic year, whether the first/last year enrollment was funded by the Ministry of Science, etc.). Granulation of dStudent dimensional table is the same as Student – one record per one student. So, the goal is to determine all primary keys (idStudent) that have changed in between two loads. We assume that all tables have accompanying CDC tables, having the same name but with the underscore prefix, i.e. _Student, _City, etc. Any change in tables referenced by student, branching out, will affect the denormalized dStudent table. For instance, if the name of the city or state changes, this will affect dStudent dimension content both directly (via CityBorn relationship) and indirectly (via StudAcdmYear table, and current City/State attributes). At first, it might appear as if it is enough to start from the Student table and follow the referenced tables, however that is only part of the solution: referencing tables might also have an impact on the base table, e.g. YearEnroll is such a table. The same is true for all other tables that are iterated with the algorithm, e.g. CityName references and affects City table (in this context). These directions or ways that data from a table affects another table and consequently the base table used to populate the dimension table, cannot be determined just by inspecting the foreign key relationships. They have to be manually assigned by the domain expert. A tool can be used to assist in this task: using the foreign key relationships a graph can be constructed with default directions assigned according to the foreign key cardinalities. After that, a domain expert only has to correct directions of certain, if any, “arrows”. Figure 2 shows such a graph for the example used. Arrow directions can be interpreted as “data flows” in the context of...
dimension table being considered. That is, data from CityName “flows”, via City, to the dimension table dStudent.

In terms of metadata, to support such setup, one only has to add a single flag to the metadata table describing the foreign keys: a flag that indicates which way the data is “flowing”. In the conclusion, the standard metadata information (tables, primary key, foreign keys) enriched with a single flag in foreign key metadata table is enough information to construct the delta view for the dimension table.

Note that though we refer to relationships between tables as “foreign keys”, in general, they don’t have to be. The graph in fact represents ways in which tables have been joined throughout the ETL process to produce the dimension table. These can be arbitrary joins.

Furthermore, the relational database might not even employ foreign keys for some reason. This only makes metadata definition process a bit less trivial, since design expert has to manually connect the tables, rather than leveraging the foreign key relationship information read from the RDBMS’s system catalog.

The delta view construction idea is to, starting from the base table (e.g. Student), recursively traverse the graph and gather primary keys of the related tables. The related tables are divided in two sets:

- **REFD** – a set of tables referenced by the current table (e.g. 
  REFD (Student) = \{City\}, REFD (City) = \{State\})

- **REFING** – a set of tables referencing the current table (e.g. 
  REFING (Student) = \{YearEnroll, StudAcadmYear\}, 
  REFING (City) = \{CityName, StudAcadmYear, 
  Student\})

Also, we assume that every table \( t \) has a corresponding \( t' \) CDC table that contains at least primary key attributes (denoted with \( PK(t) \)) and operation attribute (update, insert or delete), e.g. for \( t = \text{Student}, \ t' = \_\text{Student} \) and \( PK(t) = \{\text{idStudent}\} \). For the sake of simplicity, in the definition of delta view we omit the operation attribute, though it should appear on every projection operation where \( PK(t) \) is used.

Finally, the delta view \( \text{delta}(t) \) for the dimension table \( t \) is given with:

\[
delta(t) = \bigcup_{r \in \text{REFD}(t)} \text{refdcdc}(t, r) \\
\bigcup_{r \in \text{REFING}(t)} \text{refingdcdc}(t, r)
\]

where:

\[
\text{cdc}(t) = \pi_{PK(t)} t' \\
\text{refdcdc}(t, r) = \pi_{PK(t)} t \bowtie \delta(t) \\
\text{refingdcdc}(t, r) = \begin{cases} 
\pi_{PK(t)}(r) & \text{if } PK(t) \subseteq PK(r) \\
\sigma_{op'=\delta t'}(r) & \text{otherwise}
\end{cases}
\]

Obviously, calling \text{refdcdc} and \text{refingdcdc} results in recursive calls that traverse the entire graph. We have divided the related tables in two sets because they produce different SQL constructs to attain the primary keys: \text{Referenced} tables must be joined to the base table to get the base table’s primary keys as stated in (3). For \text{referencing} tables, as stated in (4), there are two possible scenarios:

1. **Referencing table has a composite primary key whose subset is a foreign key referencing the current table** (e.g. StudAcadmYear has \{idStudent, acadmYear\} primary key, where \{idStudent\} is used to reference the Student table).

2. **Referencing table has a surrogate key and references the current table via the foreign key attribute that is not a part of the primary key**. In this case, the CDC set for the referencing table has to be calculated and then

In this case, it is superfluous to join the referencing table to the current table since current table’s CDC primary keys can be attained using referencing table only, e.g. \( \pi_{id\text{Student}} (\delta(\text{StudAcadmYear})) \)

We denote this case in the final delta view example with the comment /* -ABBREV- */; as this is the abbreviated SQL construct, without the current table join.

- **Referencing table has a surrogate key and references the current table via the foreign key attribute that is not a part of the primary key**. In this case, the CDC set for the referencing table has to be calculated and then
joined with the same table to extract the foreign key. For instance, for the YearEnroll table:

\[
\pi_{\text{idStudent}}(\text{delta}(\text{YearEnroll})) \rightarrow \alpha
\]

\[
(\text{YearEnroll} \cup \sigma_{\text{wp}}('\text{delta}(\text{YearEnroll})))
\]

We denote (alias) the table with “exp” as in “expanded”, since we are using the self-join to “expand the primary key” to get the foreign key. However, since we are using the current data to get the foreign key, we must also include (UNION) the deleted records. We denote this case in the final delta view example with the comment the delete exception.

Note that, when traversing the chart, certain nodes are reached more than once (e.g. City). edc function for that node returns the same keys (SQL) so there is no need to execute (traverse the subgraph) more than once. More precisely, our primary concern is not with the algorithm execution time or complexity in that respect, since these charts are small – it is hard to imagine more than few hundred nodes (tables) which presents no problem.

We are concerned with creating more compact and readable SQL which is, hopefully, more beneficial to the relational engine in terms of query optimization. To that end, when implementing the above algorithm, we use the ANSI SQL-99 WITH CLAUSE to define reusable SQL statements or “local views”. Every node that is reached more than once will create a WITH CLAUSE that will be used more than once in the main SELECT statement. With clauses may be referencing each other, so it is important to order them properly. Nodes further from the base table should be ordered first. For the node’s distance from the base table we take the maximal distance, since a node can be reached via different routes, e.g. City has distances 1 and 2 (via StudAcmdYear), therefore its distance is 2.

Finally, Figure 3 shows the final delta view for the Student dimension table.

We have applied this incremental loading technique to the real world project dimension table consisting of 62 attributes and two accompanying minidimensions (31 and 19 attributes), that is, 112 attributes in total. Dimension itself is based on 23 relational tables and trigger based change data capture is used to detect changes in the data. Table 1 shows the average dimension load times accumulated over the years (N is the number of load times, where one load corresponds to one day). When we switched from full reload to the proposed delta load, the average duration was about 89 minutes. Currently, using delta load the average duration is only 4 minutes. We’ve continued to perform full reloads occasionally (on weekends), for testing purposes, and the full reload duration with the current data volume averages roughly 127 minutes. Obviously, by employing incremental loading approach we have significantly reduced the time spent to populate the dimension.

### Table 1 Average load times in a real world case

<table>
<thead>
<tr>
<th>Load type</th>
<th>N (loads)</th>
<th>Average (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>707</td>
<td>88.67</td>
</tr>
<tr>
<td>Delta</td>
<td>1276</td>
<td>4.03</td>
</tr>
<tr>
<td>Full (test)</td>
<td>170</td>
<td>126.97</td>
</tr>
</tbody>
</table>

Initially, we have constructed the delta view by hand and spent several days writing and testing the query. Using the proposed algorithm, we were able to describe and generate the same delta view in less than an hour. The view spans 227 lines of SQL using the same formatting as shown in Figure 3.

V. CONCLUSION

In this paper we have presented an algorithm for delta view construction that enables incremental loading of dimension tables. This method is suitable for large, complicated (both in terms of attribute and row count) dimensions, because they can have significant impact on the ETL process. We have shown that delta view, yielding the keys of records that need to be incrementally
(re)processed, can be constructed from the graph of data flow dependencies that can be fairly easy assembled. Obviously, incremental loading of large dimensions can significantly shorten the ETL process, but writing (and maintaining!) large delta views manually is complex and error prone. On the other hand, data flow graph is simple to define, especially with the help of a tool that, for instance, uses RDBMS’s metadata. We therefore find this algorithmic approach to delta view generation to be the apt solution for the given problem. Finally, we have verified it by applying it to the real world case and achieved significant reduction in loading times.

REFERENCES