Near Real-Time Data Warehousing Using State-of-the-Art ETL Tools

Thomas Jörg and Stefan Dessloch

University of Kaiserslautern, 67653 Kaiserslautern, Germany,
joerg@informatik.uni-kl.de, dessloch@informatik.uni-kl.de

Abstract. Data warehouses are traditionally refreshed in a periodic manner, most often on a daily basis. Thus, there is some delay between a business transaction and its appearance in the data warehouse. The most recent data is trapped in the operational sources where it is unavailable for analysis. For timely decision making, today's business users asks for ever fresher data.
Near real-time data warehousing addresses this challenge by shortening the data warehouse refreshment intervals and hence, delivering source data to the data warehouse with lower latency. One consequence is that data warehouse refreshment can no longer be performed in off-peak hours only. In particular, the source data may be changed concurrently to data warehouse refreshment. In this paper we show that anomalies may arise under these circumstances leading to an inconsistent state of the data warehouse and we propose approaches to avoid refreshment anomalies.

Key words: Near real-time data warehousing, Change Data Capture (CDC), Extract-Transform-Load (ETL), incremental loading of data warehouses

1 Near Real-Time Data Warehousing

Data warehousing is a prominent approach to materialized data integration. Data of interest, scattered across multiple heterogeneous sources is integrated into a central database system referred to as the data warehouse. Data integration proceeds in three steps: Data of interest is first extracted from the sources, subsequently transformed and cleansed, and finally loaded into the data warehouse. Dedicated systems referred to as Extract-Transform-Load (ETL) tools have been built to support these data integration steps.

The data warehouse facilitates complex data analyses without placing a burden on the operational source systems that run the day-to-day business. In order to catch up with data changes in the operational sources, the data warehouse is refreshed in a periodic manner, usually on a daily basis. Data warehouse refreshment is typically scheduled for off-peak hours where both, the operational sources and the data warehouse experience low load conditions, e.g. at nighttime. In summary, the traditional data warehouse stores historical data as of yesterday while current data is available in the operational systems only.

Today's business users, however, demand for up-to-date data analyses to support timely decision making. A workable solution to this challenge is shortening the data warehouse loading cycles. This approach is referred to as *near real-time* data warehousing or *microbatch* ETL [4]. In contrast to "true" real-time solutions this approach builds on the mature and proven ETL system and does not require the re-implementation of the transformation logic. The major challenge of near real-time data warehousing is that data warehouse refreshment can no longer be postponed to off-peak hours. In particular, changes to the operational sources and data warehouse refreshment may happen concurrently, i.e. the ETL system cannot assume the source data to remain stable throughout the extraction phase. We show that *anomalies* may occur under these circumstances causing the data warehouse to end up in an incorrect state. Thus, special care must be taken when attempting to use traditional ETL jobs for near-real time data warehousing. In this paper, we propose several approaches to prevent data warehouse refreshment anomalies and discuss their respective advantages and drawbacks.

The remainder of this paper is structured as follows: In Section 2 we discuss related work on data warehouse refreshment anomalies. In Section 3 we briefly explain the concept of incremental loading and present examples for refreshment anomalies. In Section 4 we discuss properties of operational sources and present a classification. In Section 5 we then propose several approaches to prevent refreshment anomalies for specific classes of sources and conclude in Section 6.

2 Related Work

Zhuge et al. first recognized the possibility of warehouse refreshment anomalies in their seminal work on view maintenance in a warehousing environment [7]. To tackle this problem the authors proposed the Eager Compensating Algorithm (ECA) and later the Strobe family of algorithms [8]. The ECA algorithm targets at general Select-Project-Join (SPJ) views with bag semantics over a single remote data source. The Strobe family of algorithms is designed for a multi-source environment but more restrictive in terms of the view definitions supported. Strobe is applicable to SPJ views with set semantics including the key attributes of all base relations only. The basic idea behind both, the ECA algorithm and the Strobe family of algorithms is to keep track of source changes that occur during data warehouse refreshment and perform compensation to avoid the occurrence of anomalies.

The major difference between the ECA algorithm and the Strobe family lies in the way compensation is performed. ECA relies on compensation queries that are sent back to the sources to offset the effect of changes that occurred concurrently to data warehouse refreshment. In contrast, Strobe performs compensation locally, exploiting the fact that the warehouse view includes all key attributes of the source relations.

Both algorithms are tailored for a specific class of data sources: It is assumed that the sources actively notify the data warehouse about changes, as soon as

they occur. Furthermore, for ECA the sources need to be able (and willing) to evaluate SPJ queries issued by the data warehouse for compensation purposes. In this paper, we extend the discussion on data warehouse refreshment anomalies to other classes of data sources with different properties.

The ECA algorithm and the Strobe family of algorithms are rather complex. It is necessary to track unanswered queries sent to the sources, detect source changes that occurred concurrently to query evaluation, construct compensating queries, or perform local compensation of previous query results.[1] In particular, the algorithms are designed for a message-oriented data exchange with the source systems. State-of-the-art ETL tools, however, allow for the implementation and execution of rather simple data flows only. The underlying model is most often a directed, acyclic graph where the edges indicate the flow of data and the nodes represent various transformation operators provided by the ETL tool. Furthermore, ETL tools are not built for message-oriented data exchange but rather for processing data in large batches. Therefore, we do not see any possibility to implement either ECA nor Strobe using a state-of-the-art ETL tool. Future real-time ETL tools may well offer such advanced features, if there will be a convergence between ETL and EAI technologies. However, for the time being other approaches need to be considered to achieve near real-time capabilities. In this paper we discuss approaches to near real-time data warehouse refreshment that can be realized with state-of-the-art ETL tools.

3 Data Warehouse Refreshment Anomalies

In this section we provide examples to illustrate potential data warehouse refreshment anomalies. Throughout the paper, we use the relational model with set semantics for data and the canonical relational algebra for the description of an ETL job's transformation logic. We believe that this model captures the essentials of ETL processing[2] and is appropriate for the discussion of data warehouse refreshment anomalies.

Suppose there are two operational sources storing information about our customers and our sales representatives in the relations C and S, respectively, as shown in Figure 1. Table C stores the names of our customers and the city they live in while table S stores the names of our sales representatives and the city they are responsible for. Name values are assumed to be unique in both tables. Suppose we want to track the relationships between sales representatives and customers at the data warehouse using the table V. For this purpose, we employ an ETL job $\mathcal{E}$ that performs a natural join of C and S, i.e. $\mathcal{E} : V = C \bowtie_{C.city=S.city} S$. The first population of a data warehouse is referred to as *initial load*. During an initial load, data from the sources is fully extracted,

[1] A pseudo code outline is presented in [7] and [8].

[2] Taking the IBM InfoSphere DataStage ETL tool as an example, the relational algebra roughly covers two-thirds of the transformation operators (so called stages) available.

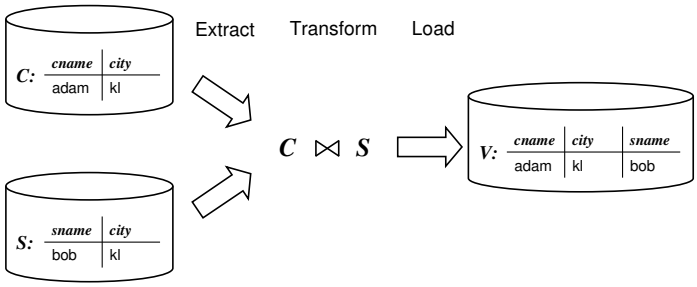

Fig. 1. Sample ETL job for initial loading

transformed, and delivered to the data warehouse. Thus, the warehouse table V initially contains a single tuple [adam, kl, bob].

As source data changes over time, the data warehouse gets stale, and hence, needs to be refreshed. Data warehouse refreshment is typically performed on a periodical basis. The naive approach is to simply rerun the initial load job, collect the resulting data, and compare it to the data warehouse content to detect changes.[3] This approach is referred to as *full reloading* and is obviously inefficient. Most often just a fraction of source data has changed and it is desirable to propagate just the changes to the data warehouse. This approach is known as *incremental loading*. ETL jobs for intial loading cannot be reused for incremental loading. In fact, incremental loading requires the design of additional ETL jobs dedicated to that purpose.

In [2, 3] we proposed an approach to derive ETL jobs for incremental loading from given ETL jobs for initial loading. We first identified distinguishing characteristics of the ETL environment, most notably properties of Change Data Capture mechanism at the sources and properties of the loading facility at the data warehouse. We then adapted change propagation approaches for the maintenance of materialized views to the ETL environment. However, data warehouse refreshment anomalies occur irrespective of the actual change propagation approach. For the reader's convenience, we ignore some aspects discussed in [2, 3] here and keep the sample ETL jobs presented below as simple as possible.

Suppose there are two relations $\triangle C$ and $\triangledown C$ that contain the insertions and deletions to C that occurred since the last loading cycle, respectively. Similarly, suppose there are two relations $\triangle S$ and $\triangledown S$ that contain the insertions and deletions to S, respectively. We refer to data about changes to base relations as *change data*. Incremental loading can be performed using two ETL jobs: The first job $\mathcal{E}_{\triangle}$ is used to propagate insertions and can be defined as $\mathcal{E}_{\triangle} : \triangle V = (C_{new} \ltimes \triangle S) \cup (\triangle C \ltimes S_{new})$ where C_{new} and S_{new} denote the current state of C and S, respectively, i.e. the changes took effect in these relations. The idea is

[3] Note that it is impractical to drop and reload the target tables because the data warehouse typically keeps a history of data changes. This aspect of data warehousing is, however, not relevant to the discussion of refreshment anomalies and therefore ignored in this paper.

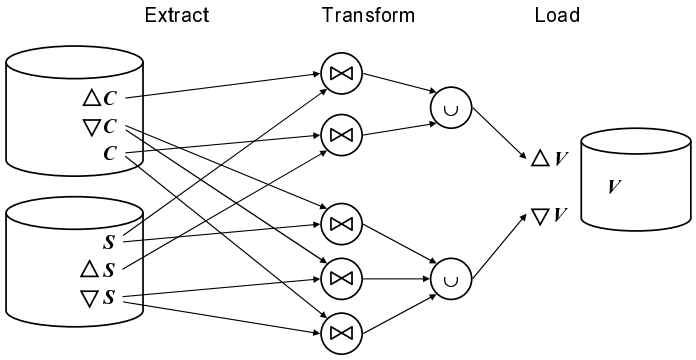

Fig. 2. Sample ETL jobs for incremental loading

to look for each inserted tuple $\triangle C$ and $\triangle S$ if matching tuples are found in the respective base relations S_{new} and C_{new}. Note that it is not required to join $\triangle C$ with $\triangle S$ since the changes already took effect in the base relations.

In a similar way, an ETL job to propagate deletions can be designed. In the expression above we could simply replace $\triangle C$ by $\triangledown C$, $\triangle S$ by $\triangledown S$, C_{new} by C_{old}, and S_{new} by S_{old}, where C_{old} and S_{old} denote the initial state of C and S, i.e. the changes did not take effect in these relations yet. However, operational sources usually cannot provide relations in their initial state, hence the ETL job must do without. The ETL job $\mathcal{E}_\triangledown$ to propagate deletions can be defined as $\mathcal{E}_\triangledown : \triangledown V = (C_{new} \ltimes \triangledown S) \cup (\triangledown C \ltimes S_{new}) \cup (\triangledown C \ltimes \triangledown S)$. Note that we sometimes "overestimate" the deletions $\triangledown V$ in this way, but this does not pose a problem here, since superfluous deletions of such tuples that are not in V do not take effect. The ETL jobs for incremental loading are depicted in Figure 2.

Example 1. Data warehouse refreshment without anomalies.
Suppose the base relations C and S initially contain the tuples $C_{old} = \{[\mathsf{adam}, \mathsf{kl}]\}$ and $S_{old} = \{[\mathsf{bob}, \mathsf{kl}]\}$. Thus, the initial state of relation V at the data warehouse is $V_{old} = \{[\mathsf{adam}, \mathsf{kl}, \mathsf{bob}]\}$. Now suppose the tuple $\triangle C = \{[\mathsf{carl}, \mathsf{kl}]\}$ is inserted into C and the tuple $\triangledown C = \{[\mathsf{adam}, \mathsf{kl}]\}$ is deleted from C. Thus, the current state of C is $C_{new} = \{[\mathsf{carl}, \mathsf{kl}]\}$. The state of S remained unchanged, i.e. $S_{new} = S_{old} = \{[\mathsf{bob}, \mathsf{kl}]\}$. To refresh the data warehouse, the ETL jobs for incremental loading $\mathcal{E}_\triangle$ and $\mathcal{E}_\triangledown$ are evaluated. $\mathcal{E}_\triangle : \triangle V = (C_{new} \ltimes \triangle S) \cup (\triangle C \ltimes S_{new})$ results in $\triangle V = \{[\mathsf{carl}, \mathsf{kl}, \mathsf{bob}]\}$ and $\mathcal{E}_\triangledown : \triangledown V = (C_{new} \ltimes \triangledown S) \cup (\triangledown C \ltimes S_{new}) \cup (\triangledown C \ltimes \triangledown S)$ evaluates to $\triangledown V = \{[\mathsf{adam}, \mathsf{kl}, \mathsf{bob}]\}$. V is refreshed by adding $\triangle V$ and subtracting $\triangledown V$ from its current state V_{old}. The new state of V is thus $V_{new} = \{[\mathsf{carl}, \mathsf{kl}, \mathsf{bob}]\}$. This is the correct result, i.e. no anomalies occurred.

Example 2. Data warehouse refreshment with a deletion anomaly.
Again, suppose the initial states of the base relations are $C_{old} = \{[\mathsf{adam}, \mathsf{kl}]\}$ and $S_{old} = \{[\mathsf{bob}, \mathsf{kl}]\}$. Now suppose that the tuples $[\mathsf{adam}, \mathsf{kl}]$ and $[\mathsf{bob}, \mathsf{kl}]$ are deleted from C and S, respectively. That is, C and S are empty in their current states $C_{new} = \{\}$ and $S_{new} = \{\}$. For reasons we will discuss in detail in the

subsequent sections, there may be some delay between the point in time changes affect the base relations, and the point in time changes are captured and visible in the corresponding change relation. Therefore, the ETL system may already see the first deletion $\nabla C = \{[\mathsf{adam}, \mathsf{kl}]\}$ but it may not see the second deletion yet, i.e. $\nabla S = \{\}$. When the ETL job $\mathcal{E}_\nabla$ is executed it returns an empty set $\nabla V = \{\}$. The reason is that a matching tuple for $\nabla C = \{[\mathsf{adam}, \mathsf{kl}]\}$ is neither found in S_{new} nor in ∇S since both relations are empty when the ETL job is executed. At some later point in time, the remaining deletion will get visible, i.e. ∇S will turn to $\{[\mathsf{bob}, \mathsf{kl}]\}$. However, because ∇C is now empty, the execution of $\mathcal{E}_\nabla$ will again result in an empty set $\nabla V = \{\}$. Relation V at the data warehouse is therefore left unchanged in both loading cycles. This result is incorrect and we speak of a *deletion anomaly*. Deletion anomalies arise when base tables are affected by deletions that have not been captured by the time incremental loading is performed.

Example 3. Data warehouse refreshment with an update anomaly.
Again, suppose the initial states of the base relations are $C_{old} = \{[\mathsf{adam}, \mathsf{kl}]\}$ and $S_{old} = \{[\mathsf{bob}, \mathsf{kl}]\}$. Now suppose that the tuple $[\mathsf{adam}, \mathsf{kl}]$ in C is updated to $[\mathsf{adam}, \mathsf{mz}]$. The current state of C is hence $C_{new} = \{[\mathsf{adam}, \mathsf{mz}]\}$. Additionally, a new tuple $[\mathsf{carl}, \mathsf{mz}]$ is inserted into S, i.e. $S_{new} = \{[\mathsf{bob}, \mathsf{kl}], [\mathsf{carl}, \mathsf{mz}]\}$. At some point in time the change to S is captured and available in $\triangle S = \{[\mathsf{carl}, \mathsf{mz}]\}$. However, suppose the change capture at C is delayed and both, $\triangle C$ and ∇C are empty up to now. When incremental loading is started in this situation the ETL jobs $\mathcal{E}_\triangle$ and $\mathcal{E}_\nabla$ will result in $\triangle V = \{[\mathsf{adam}, \mathsf{mz}, \mathsf{carl}]\}$ and $\nabla V = \{\}$, respectively. In consequence, the new state of V after data warehouse refreshment is $V_{new} = \{[\mathsf{adam}, \mathsf{kl}, \mathsf{bob}], [\mathsf{adam}, \mathsf{mz}, \mathsf{carl}]\}$. Recall that the *name* attribute of C is assumed to be unique. Considering this, no state of the base relations exist that yields to the state observed for V. Thus, V is *inconsistent* after data warehouse refreshment and we speak of an *update anomaly*. Update anomalies arise when base tables are affected by updates that have not been captured by the time incremental loading is performed. Note that the resulting inconsistencies are a temporary issue. Given that no other updates occur, the inconsistencies are resolved in the subsequent loading cycle. Note that this is not the case for inconsistencies arising from deletion anomalies.

After having seen an example for deletion and update anomalies one may ask if there are *insertion anomalies* as well. In the strict sense, insertion anomalies do exist. They arise from insertions that affected the base table but have not been captured by the time incremental loading is performed. Insertion anomalies cause the same tuple to be sent to the data warehouse multiple times in successive loading cycles. Under set semantics, however, this does not lead to an inconsistent data warehouse state. Therefore anomalies caused by insertions may not be regarded as actual anomalies.

4 Properties of Operational Data Sources

Incremental loading is the preferred approach to data warehouse refreshment because it generally reduces the amount of data that has to be extracted, transformed, and loaded by the ETL system. ETL jobs for incremental loading require access to source data that has been changed since the previous loading cycle. For this purpose, so called Change Data Capture (CDC) mechanisms at the sources can be exploited, if available. Additionally, ETL jobs for incremental loading potentially require access to the overall data content of the operational sources.

Operational data sources differ in the way data can be accessed. Likewise, different CDC mechanisms may be available. In the remainder of this section we present a classification of operational sources with regard to these properties based on [4] and [6].

Snapshot sources Legacy and custom applications often lack a general purpose query interface but allow for dumping data into the file system. The resulting files provide a *snapshot* of the source's state at the time of data extraction. Change data can be inferred by comparing successive snapshots. This approach is referred to as *snapshot differential* [5].

Logged sources There are operational sources that maintain a *change log* that can be queried or inspected, so changes of interest can be retrieved. Several implementation approaches for log-based CDC exist: If the operational source provides active database capabilities such as triggers, change data can be written to dedicated log tables. Using triggers, change data may be logged as part of the original transaction that introduced the changes. Alternatively, triggers can be specified to be deferred causing change data to be written in a separate transaction.

Log-based CDC can also be implemented by means of application logic. In this case, the application program that changes the back-end database is responsible for writing the respective change data to the log table. Again, logging can be performed either as part of the original transaction or on its own in a separate transaction.

Database log scraping or log sniffing are two more CDC implementation approaches worth being mentioned here [4]. The idea is to exploit the transaction logs kept by the database system for backup and recovery. Using database-specific utilities, changes of interest can be extracted from the transaction log. The idea of log scraping is to parse archive log files. Log sniffing, in contrast, polls the active log file and captures changes on the fly. While these techniques have little impact on the source database, they involve some latency between the original transaction and the changes being captured. Obviously, this latency is higher for the log scraping approach.

In the remainder of this paper we will refer to those sources that log changes as part of the original transaction as *synchronously* logged sources while we refer to sources that do not have this property as *asynchronously* logged sources.

Timestamped sources Operational source systems often maintain timestamp columns to indicate the time tuples have been created or updated, i.e. whenever a tuple is changed it receives a fresh timestamp. Such timestamp columns are referred to as *audit columns* [4]. Audit columns may serve as the selection criteria to extract just those tuples that have been changed since the last loading cycle. Note that deletions remain undetected though.

Lockable sources Operational sources may offer mechanism to lock their data to prevent it from being modified. For instance, database table locks or file locks may be used for this purpose.

5 Preventing Refreshment Anomalies

In Section 3 we have shown that refreshment anomalies cause the data warehouse to become inconsistent with its sources. Analysis based on inconsistent data will likely lead to wrong decisions being made, thus an inconsistent data warehouse is of no use.

In this section we discuss approaches to prevent refreshment anomalies and keep the data warehouse consistent. Refreshment anomalies occur for two reasons.

- The ETL system sees base tables in a changed state but it does not see the complete change data that lead to this state. Thus, there is a mismatch between the base table and its change data. Such a *change data mismatch* may occur for two reasons. First, for several CDC techniques there is some latency between the original change in the base relation and the change being captured. Second, even in case the change is captured as part of the original transaction, the ETL system may still see a mismatch: ETL jobs for incremental loading often evaluate joins between base relations and change data in a nested loop fashion. That is, the change data is first extracted and then used in the outer loop. Subsequently, the operational source is queried for matching tuples. When the base relation is not locked, it may be changed in the meantime and the ETL system effectively sees a mismatch between the extracted change data and the current base relation.
- The ETL jobs for incremental loading presented in Section 3 are based on traditional change propagation principles. In particular, a mismatch between the base relations and its change data is not anticipated.

Considering the two reasons that cause refreshment anomalies, there are two basic approaches to prevent them: Either the ETL jobs can be prevented from seeing a mismatch between a base relation and its change data or advanced ETL jobs for incremental loading can be developed that work correctly in spite of the change data mismatch. We will discuss both options in the remainder of this section.

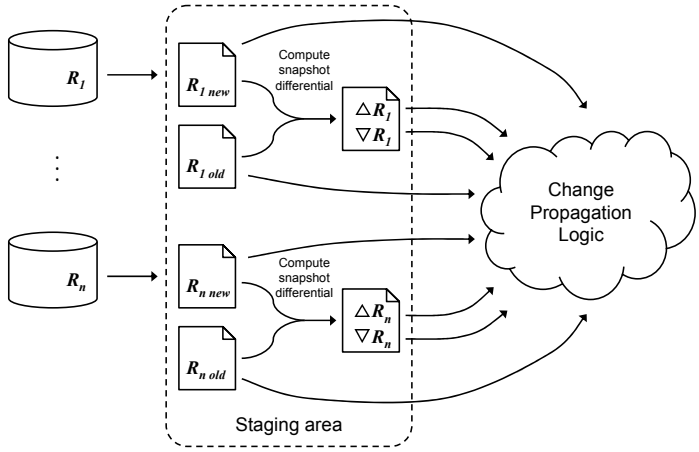

Fig. 3. Computing snapshot differentials in the staging area

5.1 Preventing a Change Data Mismatch

There are several approaches to prevent the ETL jobs from seeing a change data mismatch. Which approach is applicable is largely determined by the properties of the operational sources. We discuss options for each of the source classes introduced in Section 4.

Snapshot Sources For snapshot sources the problem is trivially solved: In each loading cycle, the ETL system request a snapshot of the sources' current state, i.e. the source data is extracted completely. The snapshot is stored at the ETL tool's working area, often referred to as *staging area*. The snapshot taken during the previous loading cycle has been kept in the staging area and the ETL system can now compute the snapshot differential by comparing the successive snapshots. The process is depicted in Figure 3.

For incremental loading the ETL system does not query the operational sources directly. Instead, queries are issued against the snapshots in the staging area. Once taken, snapshots obviously remain unchanged. Therefore, the ETL jobs will not see change data mismatches and data warehouse refreshment anomalies will not occur.

In the discussion on incremental loading in Section 3 we assumed that the base relations are available in their current state only. Hence, we designed ETL jobs in a way such that access to the initial state is not required. Here, snapshots of the current and the initial state are available in the staging area. Thus, we can design ETL jobs for incremental loading that rely on both states. The benefit is that the required change propagation logic is generally simpler in this case, i.e. the ETL job can be implemented using fewer operators as suggested in Section 3.

Computing snapshot differentials is straightforward and prevents refreshment anomalies. However, this approach has severe drawbacks: Taking snapshots is expensive; large volumes of data have to be extracted and sent over the network.

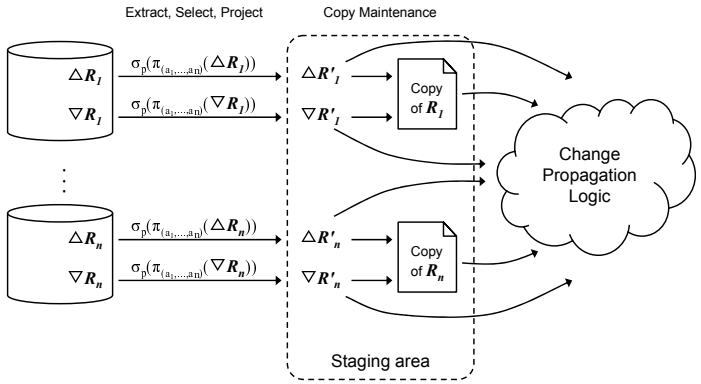

Fig. 4. Staging copies of the base relations

This may be acceptable in off-peak hours but is not an option when the operational systems are busy. Furthermore the ETL system is required to compute snapshot differentials which is again expensive [5] and the storage cost at the staging area is high; roughly double the size of all relevant base relations is required. In summary, the snapshot differentials approach does not scale well to short loading cycles that facilitate near real-time data warehousing.

Logged Sources Logged sources maintain a change log that can be queried by the ETL system. In this way, the ETL system can extract the changes that occurred since the previous loading cycle.

As we have seen, refreshment anomalies arise from a mismatch between the state of the base relations and the change data in the log. That is, there are two options to avoid a change data mismatch and thus rule out refreshment anomalies: It can either be ensured that 1) the operational sources are not changed during incremental loading, or 2) a copy of the base relation can be maintained in the staging area.

The first approach is feasible when the logged source is *lockable*. Special care must be taken when the source is logged *asynchronously*. Then there is some latency between the original change and the corresponding log entry. Thus, simply locking the base table cannot avoid a change data mismatch because changes that occurred before the lock was placed may not have been written to the change capture log yet. If there is no mechanism to "flush" the change log after the base relations have been locked, this approach cannot avoid refreshment anomalies in the general case. The drawback of locking operational sources is obvious: For the duration of incremental loading, all writing transactions at the sources are blocked. This may not be acceptable apart from off-peak hours.

The second strategy to avoid a change data mismatch for logged sources is to maintain copies of the relevant base relations in the staging area. This comes at the cost of additional storage space but minimizes the impact on the operational sources.

At the beginning of a loading cycle the ETL system queries the sources for change data. No other queries are issued towards the sources for the rest of the loading cycle. The change data is used by the ETL system in two ways as shown in Figure 4. First, it serves as the input for the ETL jobs for incremental loading. Second, it is used to maintain the local copy of the base relation. The maintenance can either be performed right away before the ETL jobs are started or after the ETL jobs are finished. In the former case the ETL jobs see a copy of the initial state of the base relations, in the latter case the ETL jobs see a copy of the current state of the base relations. The ETL jobs need to be tailored to one or the other case.

Keeping copies of base relations in the staging area avoids refreshment anomalies for both, synchronous and asynchronous logged sources. In the asynchronous case there may be some latency between the base relation change and the corresponding log entry. Consequently, changes that have not been logged by the time the loading cycle begins will not be considered for maintaining the staged copy. That is, the state of the copy may lag behind the state of the base relation. However, the copies are always consistent with the extracted change data, thus a change data mismatch cannot occur.

In many cases it is not required to stage copies of entire base relations: The base relations may contain attributes that are not included in the data warehouse schema. Such columns are dropped during ETL processing by means of a projection operator. Furthermore, only source tuples that satisfy given predicates may be relevant to the data warehouse. In this case, the ETL job contains a selection operator that discards tuples not satisfying the predicate. To save storage space in the staging area the copies of base relations can be restricted to relevant attributes and tuples. Therefore, the ETL job's projection and selection operators are "pushed down" and directly applied to the change data while it is transferred to the staging area as depicted in Figure 4. The staged copies are Select-Project (SP) views in the sense of [1] and must be maintainable using only the change data extracted from the sources. In [1] it has been shown that SP views are always self-maintainable with respect to insertions. A sufficient condition for self-maintainability of SP views with regard to deletions is to retain the key attributes in the view. Therefore any staged copy should contain the key attributes of its base relation even if they are not part of the data warehouse schema.

Compared to other approaches discussed so far, staging copies of base relations has several advantages: Most importantly, the impact on the operational sources is minimal. Only small volumes of data need to be extracted in each loading cycle and the sources are not burdened in any other way. The disadvantage is the additional storage space required at the staging area.

Timestamped Sources In timestamped sources, changes are captured by querying for tuples with a timestamp later than the latest timestamp, seen during the last loading cycle. Recall that deletions cannot be detected in this way.

Thus, only insertions (and updates[4]) can be propagated to the data warehouse. This restriction is well acceptable when historical data is kept in the data warehouse as is most often the case. A change data mismatch can occur when the ETL system needs to query the operational sources during incremental loading. The ETL system may then see changes to the base relations that occurred after the change data was extracted.

If the timestamped source is lockable, the change data mismatch can be avoided by locking the base relations while incremental loading is performed. Locks must be acquired before the change data is extracted and must not be released until all queries towards the respective base relation have been answered. As mentioned before, locking operational systems seriously interferes with business transaction processing.

To minimize the impact on the operational systems and avoid refreshment anomalies at the same time we proposed to stage copies of the base relations in the discussion on logged sources before. This approach, however, poses problems for timestamped sources. Recall that deletions remain undetected when audit columns are used for change capture. Hence deletions cannot be propagated to the staged copies and the staged copies grow steadily. Even worse, change propagation is skewed in a subtle way: Tuples that have been deleted from the base relations remain in the staged copies and thus influence the change propagation. In this way, changes propagated to the warehouse may partly arise from tuples that do no longer exist in the sources. If the data warehouse keeps a history of changes this is undesirable. We illustrate this effect with an example.

Example 4. Reconsider the sample source and target schemas introduced in Section 3. Again, suppose the initial states of the base relations are $C_{old} = \{[\mathsf{adam}, \mathsf{kl}]\}$ and $S_{old} = \{[\mathsf{bob}, \mathsf{kl}]\}$. Now suppose that the tuple $[\mathsf{adam}, \mathsf{kl}]$ is deleted. Since deletion cannot be detected here, no change is propagated to the warehouse. This is all right if the warehouse is supposed to keep historical data. Say, a new tuple $\triangle S = \{[\mathsf{charly}, \mathsf{kl}]\}$ is inserted into S. Then the ETL job $\mathcal{E}_{\triangle}$ will result in $\triangle V = \{[\mathsf{adam}, \mathsf{kl}, \mathsf{charly}]\}$ because the deleted tuple is retained in the staged copy of C. However, Adam was never responsible for Charly, thus the data warehouse's history is falsified.

In summary, staging copies of timestamped sources should be used with caution. First, the staged copies grow in size steadily and second, change propagation may be skewed in a subtle way.

5.2 Making Change Propagation Anomaly-Proof

In the beginning of this section we identified two reasons that cause refreshment anomalies. First, anomalies may arise from a change data mismatch; we discussed approaches to avoid this in the previous section. Second, the ETL jobs

[4] The insertion of a tuple with a primary key value that already exists in the warehouse relation simply overwrites the existing tuple and can hence be seen as an update, though it lacks its deletions counterpart here.

for incremental loading rely on traditional change propagation mechanisms. In this section we propose "anomaly-proof" change propagation approaches that work correctly in spite of a change data mismatch and can be implemented using state-of-the-art ETL tools. In particular, we are interested in solutions that neither lock operational sources nor maintain data copies in the staging area.

All solutions discussed in the previous section guarantee that data warehouse refreshment is done *correctly*. Without having defined it explicitly, by *correctness* we mean that incremental loading always leads to the same data warehouse state as full reloading would do. Some approaches proposed in this section do not achieve this levels of correctness. Depending on the data warehousing application, lower levels of correctness may be acceptable. Therefore we define a hierarchy of correctness levels based on [7] that allows us to classify the approaches proposed in the remainder of this section.

- *Convergence:* For each sequence of source changes and each sequence of incremental loads, after all changes have been captured and no other changes occurred in the meantime, a final incremental load leads to the same data warehouse state as a full reload would do. However, the data warehouse may pass through intermediary states that would not appear, if it was fully reloaded in each loading cycle.
- *Weak Consistency:* Convergence holds and for each data warehouse state reached after incremental loading, there are valid source states such that full reloading led to this state of the data warehouse.
- *Consistency:* For each sequence of source changes and each sequence of loading cycles, incremental loading leads to the same data warehouse state as full reloading would do.

To satisfy the convergence property a data warehouse refreshment approach must avoid deletion anomalies. However, it may permit for update anomalies because they appear only temporarily and are resolved in subsequent loading cycles. To satisfy the weak consistency property a refreshment approach must not allow for update anomalies. As demonstrated in Example 3 in Section 3 an update anomaly may lead to a data warehouse state that does not correspond to any valid state of the sources. This is contradictory to the definition above. Note that all data warehouse refreshment approaches discussed in the previous section satisfy the consistency property.

Logged Sources Synchronously logged sources capture changes as part of the original transaction. A change data mismatch may still occur, when the ETL system runs separate transactions to extract change data and query the base relations. Using global transactions instead, the change data mismatch can be avoided. However, global transactions acquire locks on the base relations for the duration of incremental loading. We discussed this approach in the previous section and identified the drawbacks of locking.

Reconsider the sample ETL job for incremental loading presented in Section 3, $\mathcal{E}_{\triangle} : \triangle V = (C_{new} \ltimes \triangle S) \cup (\triangle C \ltimes S_{new})$. Since $\triangle C$ and $\triangle S$ are typically much smaller than C and S, it is appropriate to evaluate the joins in a nested

loop fashion.[5] In this way only matching tuples need to be extracted from the base relations. When the ETL job is started, the ETL system first extracts the change data $\triangle C$ and $\triangle S$. These datasets are used in the outer loop of the join operators. Hence, for each tuple in $\triangle C$ and $\triangle S$, one query is issued towards the base relations S and C, respectively. Each query is evaluated in a separate transaction, i.e. the locks acquired at the operational sources are released early. Changes to C and S that occur after the change data has been extracted and before the last query was answered, result in a change data mismatch and may thus lead to refreshment anomalies.

To avoid the change data mismatch, the ETL system may use information from the change log to "compensate" for base relation changes that happen concurrently with incremental loading. Say, the previous incremental load was performed at time t_1 and the current incremental load is started at time t_2. When the ETL job $\mathcal{E}_\triangle$ is started, the ETL system first extracts the changes to C and S for the time interval from t_1 to t_2, denoted as $\triangle C\,[t_1, t_2]$ and $\triangle S\,[t_1, t_2]$, respectively. Once this is done, the ETL system starts to issue queries against the base relations C and S to evaluate the joins. The state of C and S may change at any time, thus query answers may contain unexpected tuples (inserted after t_2) or lack expected tuples (deleted after t_2). To avoid this, the ETL system can use the change log to compensate for changes that occurred after t_2. Instead of querying C and S directly, the ETL system can issue queries against the expressions $C \setminus \triangle C\,[t_2, \mathsf{now}] \cup \nabla C\,[t_2, \mathsf{now}]$ and $S \setminus \triangle S\,[t_2, \mathsf{now}] \cup \nabla S\,[t_2, \mathsf{now}]$, respectively. In this way, the query answers will neither contain tuples inserted after t_2 nor lack tuples deleted after t_2.

For this approach to be feasible, the source system has to meet several prerequisites: It must be capable of evaluating the compensation expression locally and in a single transaction. Furthermore, the source must be logged synchronously and it must be possible to "browse" the change log instead of reading it in a destructive manner. If these prerequisites are met, the outlined approach avoids refreshment anomalies and satisfies the consistency property.

For synchronously logged sources that do not meet these prerequisites or asynchronously logged sources, we do not see any possibility to achieve consistency using state-of-the-art ETL tools, unless staging copies of base relations is an option. However, there is a way to achieve convergence. Recall that the convergence property precludes deletion anomalies while it allows for update anomalies. Thus, making the deletion propagation anomaly-proof is sufficient to achieve convergence. No modifications with regard to the propagation of insertions are required. Consider the sample ETL jobs for incremental loading presented in Section 3 again. To achieve convergence, we need to modify $\mathcal{E}_\nabla$ in a way such that deletions are correctly propagated in spite of a change data mismatch.

In [1] it has been shown that a sufficient condition for SPJ views to be self-maintainable with respect to deletions is to retain all key attributes in the view. Thus, deletions can be propagated to a data warehouse relation V, using only the

[5] ETL tools typically allow the ETL developer to choose the physical join operator.

change data and V itself, if V contains all key attributes of the base relations and the ETL transformation logic consists of selection, projection, and join operators only. In particular, querying base relations is not required for change propagation and hence, a change data mismatch cannot occur.

Example 5. Reconsider Example 2 presented in Section 3 that shows a deletion anomaly. The initial situation is given by $C_{old} = \{[\mathsf{adam}, \mathsf{kl}]\}$, $S_{old} = \{[\mathsf{bob}, \mathsf{kl}]\}$, $V_{old} = \{[\mathsf{adam}, \mathsf{kl}, \mathsf{bob}]\}$, $C_{new} = \{\}$, $S_{new} = \{\}$, $\triangledown C = \{[\mathsf{adam}, \mathsf{kl}]\}$, and $\triangledown S = \{\}$. Note that there is a change data mismatch because the tuple $[\mathsf{bob}, \mathsf{kl}]$ has been deleted from S but $\triangledown S$ is empty as yet. Since V includes the key attributes *cname* and *sname* of both base relations, it is self-maintainable with respect to deletions, thus deletions can be propagated using only $\triangledown C$, $\triangledown S$, and V itself. In response to the deletion $\triangledown C = \{[\mathsf{adam}, \mathsf{kl}]\}$ all tuples from V where $cname = \prime\mathsf{adam}\prime$ are deleted. In the example, $[\mathsf{adam}, \mathsf{kl}, \mathsf{bob}]$ is deleted from V. When the deletion to S is eventually captured, $\triangledown S$ turns into $[\mathsf{bob}, \mathsf{kl}]$. Now all tuples where $sname = \prime\mathsf{bob}\prime$ are deleted from V. However, no such tuple is found in V. Finally V is empty, which is the correct result.

In summary, for logged sources it is possible to refresh the data warehouse incrementally and satisfy the convergence property, if the data warehouse relation includes all base relation key attributes.

Timestamped Sources As discussed before, change capture based on timestamps cannot detect deletions. This restriction is acceptable if we refrain from propagating deletions to the data warehouse and keep historical data instead. Deletion anomalies are not an issue in this case. However, update anomalies may occur when traditional change propagation techniques are used as shown in Section 3. Recall that update anomalies arise from base relation updates that occur in-between the time change data is fully extracted and the time change propagation is completed. During change propagation, the ETL system issues queries towards the base relations and such updates may influence the query results in an unexpected way and cause update anomalies.

Update anomalies can be avoided by exploiting timestamp information during change propagation. Say, the previous incremental load was performed at time t_1 and the next incremental load is started. The ETL system first extracts all tuples with a timestamp greater than t_1. These tuples make up the change data. The biggest timestamp seen during the extraction determines the current time t_2. When the ETL system queries the base relations, the answers may include tuples that have been updated after t_2. Using timestamps, such "dirty" tuples can easily be detected but it is not possible to find out about the state of these tuples before t_2. However, ignoring dirty tuples already avoids update anomalies. Note that ignoring dirty tuples does not prevent any changes from being propagated. In fact, the propagation is just postponed. All dirty tuples carry a timestamp greater than t_2 and will thus be part of the change data in the subsequent incremental load. However, because changes may be propagated with a delay, this approach satisfies the weak consistency property only.

6 Conclusion

Near real-time data warehousing reduces the latency between business transaction at the operational sources and their appearance at the data warehouse. It facilitates the analysis of more recent data and thus, timelier decision making. The advantage of near real-time data warehousing over "true" real-time solutions is that it builds on the mature and proven ETL system and does not require a re-implementation of the ETL transformation logic on another platform.

Care must be taken when a traditional data warehouse is refreshed in near real-time. One consequence of shortening the loading intervals is that refreshment may no longer happen at off-peak hours only. In fact, the operational source data may change while incremental loading is performed. We showed that refreshment anomalies may arise and cause the data warehouse to end up in an inconsistent state.

We identified two ways to tackle this problem: First, the ETL system can be prevented from seeing a change data mismatch. Second, advanced change propagation approaches can be employed that work correctly in spite of a change data mismatch. We considered both options and proposed several approaches to avoid refreshment anomalies that can be implemented using state-of-the-art ETL tools. For each of these approaches we discussed their impact on the operational sources, storage cost, level of consistency, and prerequisites with regard to change data capture properties. We believe that our results are valuable for ETL architects planning to migrate to data warehouse refreshment in near real-time.

References

1. Gupta, A., Jagadish, H. V., Mumick, I. S.: Data Integration using Self-Maintainable Views. EDBT, 1996, 140-144
2. Jörg, T., Dessloch, S.: Towards generating ETL processes for incremental loading. IDEAS, 2008, 101-110
3. Jörg, T., Dessloch, S.: Formalizing ETL Jobs for Incremental Loading of Data Warehouses. BTW, 2009, 327-346
4. Kimball, R., Caserta, J.: The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data. John Wiley & Sons, 2004
5. Labio, W., Garcia-Molina, H.: Efficient Snapshot Differential Algorithms for Data Warehousing. VLDB, 1996, 63-74
6. Widom, J.: Research Problems in Data Warehousing. CIKM, 1995, 25-30
7. Zhuge, Y., Garcia-Molina, H., Hammer, J., Widom, J.: View Maintenance in a Warehousing Environment. SIGMOD Conference, 1995, 316-327
8. Zhuge, Y., Garcia-Molina, H., Wiener, J. L.: Consistency Algorithms for Multi-Source Warehouse View Maintenance. Distributed and Parallel Databases, 1998, 6, 7-40