View Maintenance using Partial Deltas

Thomas Jörg and Stefan Dessloch
University of Kaiserslautern, Germany
{joerg|dessloch}@cs.uni-kl.de

Abstract: This paper addresses maintenance of materialized views in a warehousing environment, where views reside on a remote database. We analyze so called Change Data Capture techniques used to capture changes (also referred to as deltas) at the source systems. We show that many existing CDC techniques do not provide complete deltas but rather incomplete (or partial) deltas. Traditional view maintenance techniques, however, require complete deltas as input. We propose a generalized technique that allows for maintaining a class of materialized views using partial deltas.

1 Introduction

Materialized views are used to pre-compute (intermediary) query results to speed up query evaluation [GM95]. Upon updates to the base data, materialized views need to be maintained to regain consistency. Assuming that updates affect just a small part of the base data, it seems wasteful to maintain a view by recomputing it from scratch. It is often more efficient to compute only the changes required to update the view. This approach is referred to as incremental view maintenance.

The concept of materialized views has been applied in distributed environments, where base tables and materialized views reside on different machines connected by a network [ZGMHW95, ZGMW98, AASY97, AAM+02]. A machine hosting materialized views is usually called data warehouse (DWH). In a DWH environment, views are typically maintained in a deferred manner, i.e. deltas are gathered at the sources and propagated periodically in batches.

It has been shown that traditional view maintenance techniques can be applied in DWH environments. However, since global transactions are prohibitively expensive, special care must be taken w.r.t. synchronization. Previous research focused on this aspect only (cf. Sec. 6).

Our work addresses an orthogonal problem. In a DWH environment, so called Change Data Capture (CDC) techniques are used to gather deltas at the source systems [KC04]. The captured deltas are often partial (or incomplete). Partial deltas may lack attribute values – the initial state of an updated tuple may not be available, for instance. Furthermore, the type of partial deltas may be uncertain, i.e. inserted tuples may not be distinguishable from updated ones.

The reasons for deltas being partial are twofold. First, there are CDC approaches that cannot deliver non-partial (or complete) deltas due to principal restrictions. Second, the

CDC process may become more efficient if partial deltas are acceptable. Traditional view maintenance techniques, however, require complete deltas and cannot be used in such an environment. In this paper we give answers to the following questions: Is it possible to maintain materialized views using partial deltas? How can traditional view maintenance techniques be generalized such that partial deltas can be propagated?

The remainder of this paper is organized as follows. In Sec. 2 we give an overview of CDC techniques used in practice. We explain why many CDC techniques provide partial deltas. Traditional view maintenance techniques presume the availability of non-partial deltas and thus, cannot be applied in many DWH setups. Hence, we aim at generalizing these techniques such that partial deltas can be propagated. To this end, we propose a formal model for partial deltas in Sec. 3. This model will provide a basis for our generalized update propagation approach. In Sec. 4 we discuss techniques to apply partial deltas to materialized views. In Sec. 5 we first review a view maintenance algorithm by Griffin et al. based on algebraic differencing, which provides the basis for our work. We then discuss view maintainability in the context of partial deltas. As we will see, views cannot be maintained using partial deltas in general. However, we will identify an important class of views, which we will call dimension views, that is maintainable here. In the remainder of this section, we show how the Griffin et al. algorithm can be generalized for the propagation of partial deltas. We discuss related work in Sec. 6 and conclude in Sec. 7.

2 Change Data Capture

Change Data Capture (CDC) is a general term for techniques that gather change information (or deltas) at source systems [KC04]. We analyzed existing CDC modules and identified four main approaches, namely utilization of audit columns, log-based CDC, change tracking, and computing snapshot differentials.

Audit columns: Source systems may maintain dedicated columns (so called audit columns) to store timestamps or version numbers for individual tuples. Whenever a tuple is changed, it is assigned a fresh timestamp. Audit columns can serve as selection criteria to retrieve tuples that have been updated since the last CDC cycle.

In its most simplistic form, a single audit column is appended to each base table. A new timestamp is assigned whenever a tuple is either inserted or updated. Hence, insertions cannot be distinguished from updates when deltas are extracted. To work around this limitation, two audit columns can be appended to base tables. The first audit column is used to store the time of insertion while the second stores the time of the last update. However, deletions remain undetected when tuples are physically deleted. Tuples can instead be logically deleted by adding yet another audit column to store the time of the (logical) deletion. However, CDC techniques backed by audit columns are generally unable to capture the initial state of updated tuples. This is obvious considering that updates are performed in-place and previous values are overwritten.

Log-based CDC: Source systems may keep a log of changes that is appended in the event of an update. Several implementation approaches for log-based CDC exist: If the source system provides active database capabilities such as triggers, deltas can be written to dedicated log tables. Log-based CDC can also be implemented by means of application logic. Database log scraping is another common CDC approach. The idea is to exploit the transaction logs kept by the database system for backup and recovery. Deltas can be extracted using database-specific utilities.

Log-based CDC mechanisms are generally capable of providing complete deltas. However, their efficiency can be improved if partial deltas are acceptable. For view maintenance the so called net effect of changes is required as input. To obtain the net effect, the change log needs to be post-processed. When a tuple has been changed multiple times, the effects of these changes are combined to produce a single delta tuple. If a tuple has been inserted and subsequently updated, for instance, a delta tuple of type insertion with the updated values is produced.

The net-effect computation is more efficient if partial output deltas are acceptable. The following quote has been taken from the SQL Server 2008 documentation on the change capture feature [Mic].

> Because the logic to determine the precise operation for a given change adds to query complexity, this option is designed to improve query performance when it is sufficient to indicate that [...] the change is either an insert or an update, but it is not necessary to explicitly distinguish between the two.

Note that not being able to distinguish between insertions and updates means that the initial state of updated tuples is also not available.

Change Tracking: Change Tracking is an alternative change capture feature of SQL Server 2008 built into the database engine [Mic]. Change tracking is being advertised as light-weight change capture solution that offers better scalability than audit column or trigger-based solutions.

Change tacking is done by making a note of the primary key of the tuple that changed, along with the type of the change (insert, update, or delete) and a version number in an internal table. To retrieve deltas, the change tracking table needs to be joined to the corresponding base table, because it does not store any non-key attributes. More precisely, an outer join needs to be used, because deleted tuples are no longer found in the base tables. Thus, deltas produced by change tracking do not contain any information about deleted tuples except for the primary keys. Furthermore, the initial state of updated tuples cannot be reconstructed, because it has been overwritten in the base table.

Snapshot Differentials: Legacy and custom applications often lack a general purpose query interface. However, it is often possible to dump a system snapshot into the file system. Deltas can then be inferred by comparing successive snapshot files.

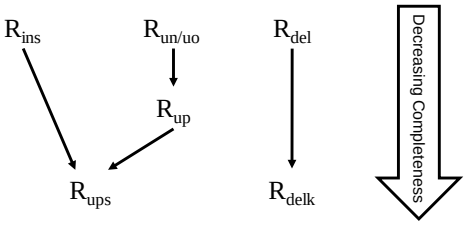

Figure 1: Delta sets with decreasing Completeness

In summary, several CDC approaches are used in practice and many of them produce partial deltas. Since there are CDC techniques that do not have this restriction, one could argue that only these techniques should be used for view maintenance in DWH environments. However, partial deltas can be captured more efficiently. Furthermore, source systems typically remain autonomous. Often, the system owners are reluctant to changes, thereby limiting the choice of practicable CDC techniques.

3 A Model for Partial Deltas

This section introduces a formal model for partial deltas. The analysis of CDC techniques in the previous section revealed different kinds of partial deltas that need to be captured in this model. First, it may not be possible to distinguish insertions from updates. Second, deltas may lack the initial state of updated tuples. Third, only the primary key of deleted tuples may be known.

Definition 1 (Partial Deltas) *Let $R(pk, a)$ be a relation with primary key pk and a set of attributes a. Let R_{old} be the state of R before it is changed and R_{new} the state of R hereafter. Partial deltas are a six-tuple of sets $(R_{ins}, R_{un/uo}, R_{del}, R_{up}, R_{ups}, R_{delk})$ where*

- *$R_{ins} \subseteq R_{new}$ denotes a set of tuples inserted into R (referred to as insertions),*
- *$R_{del} \subseteq R_{old}$ denotes a set of tuples deleted from R (referred to as deletions),*
- *$R_{un/uo}(pk, a_{un}, a_{uo})$ with $\pi_{pk,a_{un}}(R_{un/uo}) \subseteq R_{new}$ and $\pi_{pk,a_{uo}}(R_{un/uo}) \subseteq R_{old}$ denotes a set of tuples updated in R (referred to as update pairs). The initial state and the current state of updated tuples is given by (pk, a_{uo}) and (pk, a_{un}), respectively,*
- *$R_{up} \subseteq R_{new}$ denotes a set of tuples updated in R in their current state only (referred to as partial updates),*
- *$R_{ups} \subseteq R_{new}$ denotes a set of tuples either inserted or updated in R (referred to as upserts),*

- $R_{delk} \subseteq \pi_{pk}(R_{old})$ *denotes a set of primary keys of tuples deleted from* R *(referred to as partial deletions)*

such that each change at the tuple level from R_{old} *to* R_{new} *is reflected by exactly one tuple in one of the delta sets* R_{ins}*,* R_{del}*,* $R_{un/uo}$*,* R_{up}*,* R_{ups}*, or* R_{delk}*. That is, the primary key values are pairwise disjoint across the delta sets and*

$$\pi_{pk}(R_{new} - R_{old}) = \pi_{pk}(R_{ins} \cup R_{un/uo} \cup R_{up} \cup R_{ups}) \text{ and}$$
$$\pi_{pk}(R_{old} - R_{new}) = \pi_{pk}(R_{del} \cup R_{un/uo} \cup R_{delk}).$$

Figure 1 depicts the connection between the six delta sets. For each change in R there is a delta tuple in one of the delta sets. However, a delta tuple may appear in different delta sets. The alternative placements are indicated by the arrows in Fig. 1. The completeness decreases while moving from the upper to the lower delta sets. Decreasing completeness means that either attribute values become unavailable (update pairs to partial updates and deletions to partial deletions) or the type of the change becomes uncertain (partial updates to upserts and insertions to upserts).

The CDC techniques introduced in the previous section, can be characterized using our model for partial deltas. Figure 2 depicts the delta sets provided by different CDC techniques.

4 Change Data Application

The purpose of change propagation is maintaining the (remote) materialized view to resynchronize it with the source data. In this paper we show that change propagation is closed under the model for partial deltas (for a certain class of view definitions). That is, given partial input deltas the process of change propagation results in partial output deltas again, that comply with the model introduced in Sec. 3. The completeness of the output deltas may however differ from the completeness of the input deltas. The output deltas may contain upserts, for instance, even when none of the input delta sets did. Once the deltas have been propagated to the DWH, they are applied to the view. Depending on their completeness, different techniques can be used for delta application. An overview is provided in the following.

	ins	del	un/uo	up	ups	delk
Audit columns	✓	✓		✓		
Log-based CDC	✓	✓	✓			
Log-based CDC (efficient net-effect computation)		✓			✓	
Change Tracking	✓			✓		✓
Snapshot differentials	✓	✓	✓			

Figure 2: Delta sets provided by different CDC techniques

Insertions To apply insertions, the SQL interface provides the INSERT statement. Furthermore many databases are equipped with bulk loading facilities to insert larger batches of records in an efficient manner.

(Partial) deletions Assuming that the view has a primary key column, just key values are required to apply deletions (whereas attribute values are not). Thus partial deletions are sufficient for view maintenance in such cases. Non-partial deletions are however relevant for change propagation. In general, the resulting deltas are less partial when non-partial deletions are provided as input. The interrelationship will be examined in Sec. 5.

Partial updates Much like partial deletions, partial updates are sufficient for maintaining views with key columns.

Update pairs In contrast to partial updates, update pairs include the old state of updated tuples. To update a tuple in place, the old state is not required. However, the DWH often keeps historical data. Data historization is typically done using the so called Slowly Changing Dimensions technique [KR02]. To do so, it is important to understand which attributes have been changed. Given an update pair this can be found out easily. Given a partial update, however, a warehouse lookup is required to find out about the initial values. The latter approach is obviously less efficient.

Upserts To apply upserts, one can either attempt an UPDATE first and issue an INSERT if no rows were affected or else run an INSERT first and issue an UPDATE if the inserted key violates the uniqueness constraint. This method has been criticized as being rather inefficient [KC04]. In the latest SQL standard the MERGE statement has been introduced to work around this issue. MERGE can be used to insert or update tuples depending on whether a user-defined condition matches. While MERGE is more efficient than the former approach, it performs worse than a sequence of INSERT and UPDATE statements. However, the latter approach is only possible when inserts and updates are given in two distinct delta sets (i.e. deltas are less partial).

It is interesting to understand the relation between change capture and change application in the context of partial deltas: While more partial deltas can be captured more efficiently, the application of more partial deltas is less efficient. Thus, there is a trade-off between change capture and change application. Note that these steps are performed at distinct systems. It is thus possible to shift workload from the source systems to the DWH (by capturing more partial deltas) or vice versa (by capturing less or non-partial deltas).

5 View Maintenance using Partial Deltas

A number of approaches to incremental view maintenance have been proposed in literature (cf. [GM95] for an overview). Our work is based on an approach known as algebraic

V	ΔV	∇V
$\sigma_p(S)$	$\sigma_p(\Delta S)$	$\sigma_p(\nabla S)$
$\pi_A(S)$	$\pi_A(\Delta S) - \pi_A(S_{old})$	$\pi_A(\nabla S) - \pi_A(S_{new})$
$S \bowtie T$	$(S_{new} \bowtie \Delta T) \cup (\Delta S \bowtie T_{new})$	$(S_{old} \bowtie \nabla T) \cup (\nabla S \bowtie T_{old})$

Table 1: Delta Rules by Griffin et al.

differencing that was introduced in [KP81] and subsequently used for view maintenance in [QW91]. Some corrections to the minimality results of [QW91] and further improvements have been presented in [GLT97]. The basic idea is to differentiate the view definition to derive expressions that compute the change to the view without doing redundant computations.

The remainder of the section is structured as follows. In Sec. 5.1 we recall a conventional view maintenance algorithm by Griffin et al. [GLT97]. We proceed with a discussion on view maintainability in the context of partial deltas in Sec. 5.2. We will identify a class of views, which we call dimension views, that are maintainable in this context. The Griffin et al. algorithm uses so called delta rules to derive incremental expressions for view maintenance from view definitions. We propose generalized delta rules in Sec. 5.3. These rules allow for deriving incremental expressions to maintain views using partial deltas.

5.1 Algebraic Differencing for View Maintenance

In this section, we recall the algorithm proposed by Griffin et al. [GLT97] that provided the base for our work. Objects of interest are relations and relational expression presented in relational algebra. Relational expressions are used to define derived relations (or views). Changes to base relations are modeled as two sets – the set of deleted tuples and the set of inserted tuples. For a relation R the set of deleted tuples is denoted by ∇R and the set of inserted tuples is denoted by ΔR. Updates are not modeled explicitly but represented by delete-insert-pairs, i.e. for each update in R there is a corresponding delta tuple in ∇R and in ΔR.

Given a relational expression that defines a view, incremental expressions are derived by recursively applying so called delta rules. The delta rules[1] defined in [GLT97] are depicted in Tab. 1. From a relational expression V two incremental expressions ∇V and ΔV are derived that compute the deletions and insertions to the view, respectively. To this end, subexpressions in V that match the "patterns" shown in the left column of Tab.1 are recursively replaced by incremental counterparts found in the middle column or the right column to obtain ΔV or ∇V, respectively. Intuitively, the delta rules in Tab. 1 can be understood as follows.

- Selection: An inserted tuple is propagated through a selection, if it satisfies the filter

[1] Since our work is focused on Select-Project-Join (SPJ) views, delta rules for union, intersection, and set difference have been omitted.

predicate. A deletion is propagated through a selection, if the tuple used to satisfy the filter predicate.

- Projection: An inserted tuple is propagated through a projection, if no alternative derivation existed before the change. A deletion is propagated through the projection, if no alternative derivation remains after the change.
- Join: New tuples appear in the join of two relations, if a tuple inserted into one relation joins to tuples in the other one. Tuples disappear from the join, if a tuple deleted from one relation used to join to tuples in the other one before the change.

The view maintenance algorithm by Griffin et al. requires complete change information. Thus, it cannot be applied if deltas are partial as described in Sec. 3. We propose a generalized view maintenance algorithm that gracefully deals with partial deltas for a restricted (but important) class of view definitions.

5.2 Dimension Views

In general, materialized views cannot be maintained using partial deltas. Consider the following example. Say there is a base relation $R(pk, a)$ with pk being the primary key column and a simple derived view $V(a) := \pi_a(R)$. Say we use a CDC mechanism that does not provide the initial state of updated tuples (such as audit columns or change tracking). Obviously, V cannot be maintained in case of an update to R. While it is straightforward to add the updated tuple to V, it is unclear which tuple in V needs to be discarded (or overwritten) in return. Similar considerations hold for the other kinds of partial deltas, i.e. partial deletions and upserts.

Note, that V was maintainable if it included the primary key column pk. Including primary keys is thus a necessary condition for views to be maintainable using partial deltas. However, not all primary keys from the source relations need to be retained in the view definition. We will discuss the selection of keys in the following. At first, we define a class of views, which we call dimension views, that has interesting properties w.r.t. maintenance using partial deltas.

Definition 2 (Dimension View) *Let V be a relational expression defining a view that contains projections, selections, and joins only. V is called dimension view, if each join $R \bowtie_p S$ has a join predicate of the form $(R.a = S.pk)$ where a is a (set of) attributes of R and pk is the (composite) primary key of S.*

In the subsequent sections, we will show that dimension views are maintainable using partial deltas, if they include those primary key attributes that are not functionally dependent on any other key attributes. With other words, all key attributes used in join predicates do not need to be included in the view.

Dimension views are commonly found in DWHs. While they are usually called dimension tables here, they store derived data and can thus be seen as views. DWHs typically

$Cust_{old}$

CID	CName	CDiscount	CAddr
1	Adam	0%	1
2	Bob	0%	2
3	Carl	0%	3

$Addr_{old}$

AID	ACity	ACountry
1	Austin	US
2	Berlin	DE
3	Chemnitz	DE

$Cust_{new}$

CID	CName	CDiscount	CAddr
1	Adam	5%	1
2	Bob	0%	4
4	Dave	0%	4

$Addr_{new}$

AID	ACity	ACountry
1	Aachen	DE
2	Berlin	DE
4	Dresden	DE

Figure 3: Sample base tables in the old and new state

use a star schema to store multi-dimensional data that consists of fact and dimension tables [KC04, KR02]. Dimension tables are used to join together data on business entities that originates from multiple source systems. For improved query performance, dimension tables are typically denormalized. Dimension tables include a unique identifier for business entities referred to as business key. Typically, no other keys originating from the sources are stored here. Note that these keys would be functionally dependent on the business key in the denormalized dimension table. Our work is thus directly applicable to incremental maintenance of dimension tables.

Example 1 *Figure 3 depicts two relations that are going to be used as a running example throughout the paper. The Cust relation stores the ID, name, and discount of customers and a reference to an address, which is stored in the Addr relation. The idea is to derive a dimension table D from these base tables. Dimension tables are typically de-normalized. Consider the sample view definition.*

$$D := \pi_{CID,CName,CAddr}(Cust) \bowtie_{(Cust.Addr=Addr.AID)} \sigma_{ACountry='\mathsf{DE}'}(Addr)$$

The view is restricted to German customers, furthermore the customer discount column is dropped. Note that the view is a dimension view w.r.t. Def. 2.

5.3 A Generalized View Maintenance Algorithm

We propose a generalization of the algorithm by Griffin et al. that allows for maintaining dimension views using partial deltas. We proceed as follows: First, we explain how partial deltas can be represented by means of delete-insert sets used by the original algorithm. Second, we propose generalized delta rules for projection, selection, and join. We show that these operators are closed under the model for partial deltas. Third, we conclude that dimension views can be maintained by our algorithm.

View maintenance algorithms (including the one by Griffin et al.) model deltas by two sets – the set of deleted tuples and the set of inserted tuples. We are going to refer to this model as delete-insert delta model or delete-insert model for short. This model does not directly match our model for partial deltas introduced in Sec. 3. The latter uses a six-tuple

$$\Delta R(pk, a) := R_{ins} \cup \pi_{pk,a_{un}}(R_{un/uo}) \cup R_{up} \cup R_{ups}$$
$$\nabla R(pk, a, flag) := \pi_{pk,a,\mathsf{comp}}(R_{del}) \cup \pi_{pk,a_{uo},\mathsf{comp}}(R_{un/uo}) \cup$$
$$\pi_{pk,\mathsf{NULL},\mathsf{up}}(R_{up}) \cup \pi_{pk,\mathsf{NULL},\mathsf{ups}}(R_{ups}) \cup \pi_{pk,\mathsf{NULL},\mathsf{up}}(R_{delk})$$
$$\text{with } flag \in \{\mathsf{comp}, \mathsf{up}, \mathsf{ups}\}$$

Figure 4: Conversion from six-tuple model to delete-insert model

representation instead and we will therefore refer to it as six-tuple delta model or six-tuple model for short.

While the six-tuple model allows for a natural representation of partial deltas, it is more complex to handle six distinct sets during update propagation. We experienced that delta rules become rather complex. In particular, the join delta rules require a large number of joins to capture the interactions between the different delta sets.

Overly complex incremental expressions can be avoided by sticking to the delete-insert delta model for the update propagation. To do so, partial deltas need to be transferred to the delete-insert model. Furthermore, the result of the update propagation needs to be converted back into the six-tuple model. The general idea here is to extend the schema of the delta sets by adding a type flag column. This flag is used to indicate the type of individual delta tuples. Unknown attribute values (of partial deltas) are padded with NULL values. One could say that partial deltas are "encoded" as special kinds of complete deltas.

We proceed by describing the conversion from six-tuple deltas to delete-insert deltas, and continue with the conversion in opposite direction.

Six-tuple model to delete-insert model The equations for converting from the six-tuple model to the delete-insert model are given in Fig. 4. For a relation R it is straightforward to express the delta sets R_{ins}, $R_{un/uo}$, and R_{del} by means of the delete-insert model, because these sets are non-partial. To distinguish complete delta tuples from partial ones, they are assigned a type flag of value comp.

The remaining delta sets are treated as follows: Partial updates R_{up} and upserts R_{ups} are added to the insert set ΔR. Note that we need to distinguish them from "regular" insertions and updates, however. To this end, we add tuples with the same primary key value to the delete set ∇R. All other attribute values are padded with NULLs, because the initial attribute values of these tuples are unknown. Additionally we add a flag to indicate the type of the delta tuple. Note that the schema of ∇R is extended to accommodate the type flag. The partial deletions R_{delk} are also added to ∇R. Since R_{delk} contains primary key values only, the missing attributes are padded with NULLs. Note that a up flag is used for partial deletions. Partial deletions can however be distinguished from partial updates. While partial updates have a matching tuple in ΔR, partial deletions do not.

Example 2 *Recall the running example introduced in Sec. 5.2. Fig. 3 depicts both, the old and the new state of the base relations Cust and Addr. Assume that a log-based CDC technique is used for Cust providing insertions, update pairs, and deletions. Further assume*

ΔCust

CID	CName	CDiscount	CAddr
1	Adam	5%	1
2	Bob	0%	4
4	Dave	0%	4

ΔAddr

AID	ACity	ACountry
1	Aachen	DE
4	Dresden	DE

∇Cust

CID	CName	CDiscount	CAddr	flag
1	Adam	0%	1	comp
2	Bob	0%	2	comp
3	Carl	0%	3	comp

∇Addr

AID	ACity	ACountry	flag
1	-	-	up
3	-	-	up

Figure 5: Sample deltas converted to the delete-insert model

$$
\begin{aligned}
R_{ins} &:= \Delta R \,\overline{\ltimes}_{pk}\, \nabla R \\
R_{un/uo} &:= \Delta R \bowtie_{pk} \sigma_{(flag=\mathsf{comp})} \nabla R \\
R_{up} &:= \Delta R \ltimes_{pk} \sigma_{(flag=\mathsf{up})} \nabla R \\
R_{ups} &:= \Delta R \ltimes_{pk} \sigma_{(flag=\mathsf{ups})} \nabla R \\
R_{del} &:= \nabla R \,\overline{\ltimes}_{pk}\, \sigma_{(flag=\mathsf{comp})} \Delta R \\
R_{delk} &:= \pi_{pk}(\nabla R \,\overline{\ltimes}_{pk}\, \sigma_{(flag\neq\mathsf{comp})} \Delta R)
\end{aligned}
$$

Figure 6: Conversion from delete-insert model to six-tuple model

that change tracking is used for Addr providing insertions, partial updates, and partial deletions. The deltas converted to the delete-insert model are depicted in Fig. 5.

Delete-insert model to six-tuple model The equations for converting from the delete-insert model back to the six-tuple model are given in Fig. 6. Note that the symbols $\ltimes$ and $\overline{\ltimes}$ are used to denote a semi join and a anti join, respectively. The R_{ins} delta set consists of those tuples in ΔR that have a primary key value not existent in ∇R. The $R_{un/uo}$ delta set consists of pairs of tuples in ΔR and ∇R having equal primary key values and being "complete". The completeness is checked by means of the type flag in ∇R. The R_{up} and R_{ups} delta sets consist of tuples in ΔR that join to tuples in ∇R having a up or ups type flag, respectively. The R_{del} and R_{delk} delta set consist of tuples in ∇R having a primary key that does not exist in ΔR and being complete or incomplete, respectively.

5.4 Projection

The original delta rules for projection depicted in Tab. 1 are repeated for the reader's convenience.

$$
\begin{aligned}
\Delta(\pi_A(S)) &\equiv \pi_A(\Delta S) - \pi_A(S_{old}) \\
\nabla(\pi_A(S)) &\equiv \pi_A(\nabla S) - \pi_A(S_{new})
\end{aligned}
$$

The projection delta rules contain a so-called effectiveness test to prevent redundant updates from being propagated. A set difference is used to discard insert delta tuples if an alternative derivation of the same tuple existed in the old database state. Similarly, delete delta tuples are discarded if an alternative derivation continues to exist in the new database state.

We can represent the new and old relation states S_{new} and S_{old} using the so called preserved state $S_o := S_{new} - \Delta S = S_{old} - \nabla S$, i.e. the set of tuples that have not been changed.

$$\Delta(\pi_A(S)) \equiv \pi_A(\Delta S) - \pi_A(S_o \cup \nabla S)$$
$$\nabla(\pi_A(S)) \equiv \pi_A(\nabla S) - \pi_A(S_o \cup \Delta S)$$

Recall that dimension views contain primary key attributes that must not be dropped by a projection. Since primary key values are unique, $\pi_A(\Delta S)$ and $\pi_A(S_o)$ are obviously disjoint. Similarly $\pi_A(\nabla S)$ and $\pi_A(S_o)$ are disjoint. Given this, the delta rules can be simplified for dimension views as follows.

$$\Delta(\pi_A(S)) \equiv \pi_A(\Delta S) - \pi_A(\nabla S)$$
$$\nabla(\pi_A(S)) \equiv \pi_A(\nabla S) - \pi_A(\Delta S)$$

The effectiveness test in the above equations can be understood as follows. An alternative derivation of a delta tuple must have the same primary key value. An alternative derivation of an insert delta tuple can thus only be found among the delete delta tuples and vice versa (recall that updates are represented as delete-insert pairs).

The aim of the effectiveness test is to discard so called ineffective updates, i.e. updates that do not change the view. In the presence of keys, an ineffective update occurs when all updated attributes are dropped by the projection. In this case, the initial state of the propagated attributes is equal to their current state. Thus, the update is ineffective w.r.t. the view.

We now discuss the implications of partial deltas w.r.t. the effectiveness test. In fact, the test may not work as expected here. The reason is that the initial state of an updated tuple may not be available. Hence, its effectiveness cannot be tested. Without having the initial state available, we do not know which attributes have been updated. Hence, we cannot know whether the update will affect the view[2]. However, propagating ineffective updates is not problematic, because a view is not changed when an ineffective update is applied. While ineffective updates cause some overhead, the view does not become inconsistent.

Delta rules for projection can be generalized to handle partial deltas. To this end, the effectiveness test is only done for complete delta tuples and omitted for partial ones.

$$\Delta(\pi_A(S)) \equiv \pi_A(\Delta S) - \pi_A(\sigma_{flag=\mathsf{comp}} \nabla S)$$
$$\nabla(\pi_A(S)) \equiv \pi_{A,flag}(\nabla S) - \pi_{A,\mathsf{comp}}(\Delta S)$$

[2]Note that our notion of ineffective updates is related to the notion of safe updates studied mainly in the context of integrity checking [Beh09]. Safe updates are an overestimation of true updates that can be computed more efficiently. For integrity checking, safe updates are often sufficient. The computation of safe updates has been proposed to improve efficiency. In contrast, our ineffective updates necessarily occur in the context of partial deltas.

ΔCust'

CID	CName	CAddr
2	Bob	4
4	Dave	4

∇Cust'

CID	CName	CAddr	flag
2	Bob	2	comp
3	Carl	3	comp

Figure 7: Result of the sample incremental expressions $\Delta Cust'$ and $\nabla Cust'$

In the second delta rule, the schema of ΔS is extended by adding a type flag column which is assigned the value comp. Note that the effectiveness test may safely be omitted. Doing so may result in a larger number of ineffective updates being propagated. However, the rules become even simpler and may be evaluated more efficiently. Furthermore, note that the delta rules are closed under the model for partial deltas. That is, the result of the operation is again partial deltas.

Example 3 *Reconsider the running example. The first term of the dimension view definition D is $Cust' := \pi_{CID,CName,CAddr}(Cust)$. We can apply the delta rules given above to derive incremental expressions $\Delta Cust'$ and $\nabla Cust'$.*

$$\Delta Cust' \equiv \pi_{CID,CName,CAddr}(\Delta Cust) - \pi_{CID,CName,CAddr}(\sigma_{flag=\mathsf{comp}}(\nabla Cust))$$
$$\nabla Cust' \equiv \pi_{CID,CName,CAddr,flag}(\nabla Cust) - \pi_{CID,CName,CAddr,\mathsf{comp}}(\Delta Cust)$$

The result deltas are depicted in Fig. 7. Note that the update to the customer tuple with ID 1 is ineffective and thus discarded.

5.5 Selection

The original delta rules for selection found in Tab. 1 are repeated here for the reader's convenience.

$$\Delta(\sigma_p(S)) \equiv \sigma_p(\Delta S) \qquad \nabla(\sigma_p(S)) \equiv \sigma_p(\nabla S)$$

These equations need to be adapted to handle partial deltas. We will discuss each type of delta separately in the following.

The inserted tuples in S_{ins} are propagated if they satisfy the selection predicate and discarded otherwise. The deleted tuples in S_{del} are treated similarly. For the update pairs in $S_{un/uo}$ both, the initial and the current state have to be considered. If the initial and the current state satisfy the selection predicate the delta tuple is passed on as an update, i.e. it remains in $S_{un/uo}$. If neither the initial nor the current state satisfy the selection predicate the update pair is discarded. If the initial state did satisfy the predicate but the current state no longer does, the initial state is propagated as a deletion, i.e. it becomes part of S_{del}. Similarly, if the initial state did not satisfy the predicate but the current state does, the current state is propagated as an insertion, i.e. it becomes part of S_{ins}.

For partial updates S_{up} the initial state is not known. It could have either satisfied the selection predicate or not. Thus, given that the current state does satisfy the predicate, the

ΔAddr'

AID	ACity	ACountry
1	Aachen	DE
4	Dresden	DE

∇Addr'

AID	ACity	ACountry	flag
1	-	-	ups
3	-	-	ups

Figure 8: Result of the sample incremental expressions $\Delta Addr'$ and $\nabla Addr'$

resulting delta is either an insert or an update. Since we cannot distinguish these cases, the delta tuple becomes part of S_{ups}, i.e. the delta tuple changes its type and becomes an upsert. Given that the current state of an partial update does not satisfy the selection predicate, a deletion needs to be propagated. Since the initial state of the updated tuple is unavailable, a partial deletion (S_{delk}) is propagated. Note that this deletion may be ineffective, i.e. the tuple to be deleted may not be found in the view, because it did not satisfy the predicate in its initial state either.

The upsert delta set S_{ups} is handled in a very similar way as partial updates. Again, the initial state of delta tuples is unavailable. Given that the current delta tuple satisfies the selection predicate, it remains in S_{ups}. If it does not, it becomes part of S_{delk}. Again, the partial deletion may be ineffective.

Partial deletions contain primary key values only. Obviously, the selection predicate can generally not be checked without having the non-key attribute values. However, tuples in S_{delk} may safely be propagated in all cases. If a tuple with the same primary key value exists in the view, it is deleted. If no such tuple exists, the view remains unchanged, i.e. the deletion turns out to be ineffective. Ineffective deletions occur, when the original tuple did not satisfy the selection predicate and therefore never appeared in the view.

$$\Delta(\sigma_p(S)) \equiv \sigma_p(\Delta S) \qquad \nabla(\sigma_p(S)) \equiv \sigma_{p \vee flag \neq \mathsf{comp}}(\pi_{a,s(flag)} \nabla S)$$

$$\text{with } s(flag) := \begin{cases} \mathsf{ups} & \text{if } flag = \mathsf{up} \\ flag & \text{else} \end{cases}$$

Given these considerations, the original delta rules for selection can be adapted to partial change data. The rule to compute the insert set does not need to be changed. The rule to compute the delete set needs some adaptations though, because the selection predicate can only be checked for non-partial delta tuples (with $flag = \mathsf{comp}$). All partial delta tuples are simply passed on. As mentioned before, partial updates may become upserts and the type flag needs to be changed accordingly. Note that the delta rules are closed under the model for partial deltas.

Example 4 *Reconsider the running example. The second term of the dimension view definition D is $Addr' := \sigma_{ACountry='\mathsf{DE}'}(Addr)$. By applying the above delta rules the incremental expressions $\Delta Addr'$ and $\nabla Addr'$ can be derived.*

$$\Delta Addr' \equiv \sigma_{ACountry='\mathsf{DE}'}(\Delta Addr)$$
$$\nabla Addr' \equiv \sigma_{(ACountry='\mathsf{DE}') \vee flag \neq \mathsf{comp}}(\nabla Addr)$$

The result deltas are depicted in Fig. 8. Note that partial updates are turned into upserts and effective partial deletions into possibly ineffective partial deletions.

		S					
		pre	ins	del	up	ups	delk
T	pre	-	ins	del	ups	ups	delk
	ins	ins	ins	-			
	del	del	-	del			
	un	un	ins	-			
	uo	uo	-	del			
	up	up	ins	-			
	ups	ups	ins	-			
	delk	delk	-	delk			

Figure 9: Join Matrix 1

5.6 Join

The original join delta rules found in Tab. 1 are repeated here for the readers convenience. Note that both, the new and the old state of the base relations are required to incrementally maintain join views.

$$\Delta(S \bowtie T) \equiv (S_{new} \bowtie \Delta T) \cup (\Delta S \bowtie T_{new})$$
$$\nabla(S \bowtie T) \equiv (S_{old} \bowtie \nabla T) \cup (\nabla S \bowtie T_{old})$$

In the DWH environment source systems are decoupled and base relations are usually available in their new state only. However, the old state can be reconstructed using the new state and the deltas. Given a relation R, the preserved state R_o can be computed by subtracting the insert delta set from the new state ($R_o := R_{new} - \Delta R$). The old state R_{old} can then be computed by adding the delete delta set to the preserved state. In the light of partial deltas, it may not be possible to fully reconstruct the old state, because delta tuples in ∇R may be partial. Recall that a flag is used to indicate the type of delta tuples in ∇R. We hence use a type flag in the old state R_{old} as well; the preserved tuples R_o are assigned with the distinct type flag pre ($R_{old} := \pi_{\ldots,\mathsf{pre}} R_o \cup \nabla R$).

In this paper, we focus on the maintenance of so called dimension views defined in Sec. 5.2. All join predicates used in dimension views follow a common pattern. They are equality predicates and involve the primary key attribute of at least one relation. In the following, we consider the join of two relations S and T with the join predicate $(S.a = T.pk)$ where $S.a$ is an arbitrary attribute of S and $T.pk$ the primary key attribute of T. It is important to understand that the type of the resulting (joined) delta tuples depend on the type of both input delta tuples. To adapt the join delta rules, all possible combinations of delta types need to be considered. The different combinations are represented by the matrices in Fig. 9 and Fig. 10. Consider the matrix in Fig. 9. The column headings represent the different delta sets of S participating in the join. From left to right, there are preserved tuples, insertions, deletions, partial updates, upserts, and partial deletions. For the sake of clarity, update pairs are shown in a separate matrix (Fig. 10). The row headings in the matrix represent the different delta sets of R participating in the join. The cells of the matrix indicate the delta type resulting from a join between the corresponding delta sets of S and R.

Consider the matrix cell at the intersection of the S_{ins} column and the T_{up} row, for in-

S	T_{new}	S	T_{old}	result
un	-	uo	-	-
un	-	uo	pre	del
un	-	uo	uo	del
un	-	uo	del	del
un	-	uo	delk	delk
un	-	uo	up	delk
un	-	uo	ups	delk
un	pre	uo	-	ins
un	pre	uo	pre	un/uo
un	pre	uo	uo	un/uo
un	pre	uo	del	un/uo
un	pre	uo	delk	up
un	pre	uo	up	up
un	pre	uo	ups	ups
un	ins	uo	-	ins
un	ins	uo	pre	un/uo
un	ins	uo	uo	un/uo
un	ins	uo	del	un/uo
un	ins	uo	delk	up
un	ins	uo	up	up
un	ins	uo	ups	ups

S	T_{new}	S	T_{old}	result
un	un	uo	-	ins
un	un	uo	pre	un/uo
un	un	uo	uo	un/uo
un	un	uo	del	un/uo
un	un	uo	delk	up
un	un	uo	up	up
un	un	uo	ups	ups
un	up	uo	-	ins
un	up	uo	pre	un/uo
un	up	uo	uo	un/uo
un	up	uo	del	un/uo
un	up	uo	delk	up
un	up	uo	up	up
un	up	uo	ups	ups
un	ups	uo	-	ins
un	ups	uo	pre	un/uo
un	ups	uo	uo	un/uo
un	ups	uo	del	un/uo
un	ups	uo	delk	up
un	ups	uo	up	up
un	ups	uo	ups	ups

Figure 10: Join Matrix 2

stance. The cell indicates that the join result of these delta sets is to be propagated as insertion. This is obvious considering that any tuple added to S has a key that is unique in S. Thus, the key cannot be in the view yet. Hence, the result of the join is an insertions w.r.t. the view.

Consider the three right-most columns in the matrix referring to partial updates, upserts, and partial deletions in S. These deltas lack certain attribute values. They hence cannot be joined to T_{old}, because the join predicate cannot be evaluated. Consider a partial update in S, for instance. Recall, that the initial state of the updated tuple is not available. Hence, it is unclear whether the updated tuple used to find a join partner in T before the update. The joined tuple is either an update w.r.t. the view (if it used to find a join partner) or an insertion (if it did not). Since these cases cannot be distinguished for partial updates, an upsert has to be propagated.

Upserts in S are propagated as upserts and partial deletions as (possibly ineffective) partial deletions. Note that partial S deltas are handled for joins in a similar way than partial deltas are handled for selections (see Sec. 5.5). The function s defined in Sec. 5.5 to translate type flags can thus be reused in the generalized delta rules for joins.

The matrix in Fig.10 represents joins involving update pairs in S. Recall that the new state of an updated tuple is joined to the new state of T (T_{new}) while the old state of an updated tuple is joined to the old state of T (T_{old}) in the delta rules for update propagation. The matrix shows all possible join combinations. Let s be an update pair in $S_{un/uo}$, s_{un} the new state of s, and s_{uo} the old state of s. The first two columns of the matrix indicate where s_{un} finds a join partner in T_{new}. There are the following possibilities: A join partner may not exists, it may be a preserved tuple, an inserted tuple, an updated tuple in its new

state, a partial update, or an upsert.

The third and fourth column indicate where S_{uo} finds a join partner in T_{old}. A join partner may not exists, it may be a preserved tuple, an updated tuple in its old state, a deleted tuple, a partial deletion, a partial update, or an upsert.

The fifth column indicates the type of the delta resulting from the joins. As an example, consider the second row of the matrix. It treats the case where s_{un} does not find a join partner in T_{new}, while s_{uo} used to join to a preserved tuple (i.e. the join attribute of s was updated). Hence, a tuple $S_{uo} \bowtie T_o$ used to be in the view and needs to be discarded now. Thus, the resulting delta is of type deletion.

The matrix in Fig. 10 reveals a pattern. Whenever s_{uo} joins to a complete tuple in T_{old}, namely a preserved tuple, an updated tuple, or a deleted tuple, the resulting delta tuple is again complete, i.e. an update pair or a deletion. When s_{uo} joins to a partial update or a partial deletion in T_{old}, the resulting delta tuple is either an partial update or a partial deletion. The distinction is made based on the existence of a corresponding delta tuple in the insert set $\Delta(S \bowtie T)$, when the deltas are converted to the six-tuple model (see Sec. 5.3). When s_{uo} joins to an upsert, the resulting delta tuple is either an upsert or a partial deletion. Based on these considerations and the considerations that lead to the first join matrix, a function j is defined to derive type flags for joined tuples from the type flags of the input tuples.

$$j(flag_s, flag_t) := \begin{cases} flag_t & \text{if } flag_s = \mathsf{pre} \\ \mathsf{comp} & \text{if } flag_s = \mathsf{comp} \wedge (flag_t = \mathsf{pre} \vee flag_t = \mathsf{comp}) \\ \mathsf{up} & \text{if } flag_s = \mathsf{comp} \wedge flag_t = \mathsf{up} \\ \mathsf{ups} & \text{if } flag_s = \mathsf{comp} \wedge flag_t = \mathsf{ups} \end{cases}$$

The join delta rules are adapted as follows to handle partial deltas.

$$\begin{aligned} \Delta(S \bowtie T) &\equiv (S_{new} \bowtie \Delta T) \cup (\Delta S \bowtie T_{new}) \\ \nabla(S \bowtie T) &\equiv \pi_{\ldots, j(S.flag, T.flag)}(S_{old} \bowtie \nabla T) \cup \\ &\quad \pi_{\ldots, j(S.flag, T.flag)}(\nabla S \bowtie T_{old}) \cup \\ &\quad \pi_{\ldots, s(flag)}(\sigma_{flag \neq \mathsf{comp}}(\nabla S)) \end{aligned}$$

Once again, the delta rule for computing the insert delta set remains unchanged. The delta rule for the delete delta set is changed in two ways. First, a function j is used to derive the type of the resulting delta tuples from the type flags of joining tuples in S and T. Second, an additional term is added to the rule to handle partial tuples in ∇S. In this additional term the function s (defined in Sec. 5.5) is used to modify the type flag as needed. Note that the join delta rules propagate partial deltas as defined in Sec. 3. The join operation is thus closed under this model.

In Sec. 5.2 we have shown that views need to include primary key attributes to be maintainable using partial deltas. A simple join view includes the primary key attributes of both base relations. As we will see, not all of these key attributes are required to maintain dimension views though. In dimension view definitions, all join predicates have a common form. They are equality predicates involving the primary key of at least one base relation.

ΔD

CID	CName	CAddr	ACity	ACountry
1	Adam	1	Aachen	DE
2	Bob	4	Dresden	DE
4	Dave	4	Dresden	DE

∇D

CID	CName	CAddr	ACity	ACountry	flag
1	Adam	1	-	-	ups
2	Bob	2	Berlin	DE	comp
3	Carl	3	-	-	ups

Figure 11: Result of the sample incremental expressions ΔD and ∇D

Reconsider the join of S and T with the join predicate ($S.a = T.pk$) with $S.a$ being an arbitrary attribute of S and $S.pk$ and $T.pk$ being primary key attributes of S and T, respectively. Obviously $S.a$ is functionally dependent on its key $S.pk$. Thus, in the join view $T.pk$ is functionally dependent on $S.pk$. Thus each join view tuple is uniquely identified by $S.pk$ alone. Hence, partial updates, upserts, or partial deletions can be applied based on $S.pk$ only. In summary, dimension views remain maintainable when key attributes used in a join predicate are projected out hereafter.

Example 5 *Reconsider the running example. The sample dimension view was defined as* $D := Cust' \bowtie_{(Cust.Addr=Addr.AID)} Addr'$. *The incremental expressions* ΔD *and* ∇D *can be derived using the delta rules given above.*

$$\Delta D \equiv ({Cust'}_{new} \bowtie \Delta Addr') \cup (\Delta Cust' \bowtie {Addr'}_{new})$$

$$\begin{aligned}\nabla D \equiv\ & \pi_{CID,CName,CAddr,ACity,ACountry,j(Cust'.flag,Addr'.flag)}({Cust'}_{old} \bowtie \nabla Addr') \\ & \cup \pi_{CID,CName,CAddr,ACity,ACountry,j(Cust'.flag,Addr'.flag)}(\nabla Cust' \bowtie {Addr'}_{old}) \\ & \cup \pi_{CID,CName,CAddr,NULL,NULL,s(flag)}(\sigma_{flag \neq \mathsf{comp}}(\nabla Cust'))\end{aligned}$$

The result deltas are depicted in Fig. 11. When ΔD *and* ∇D *are converted back to the six-tuple model, one obtains an insertion (ID 4), an upsert (ID 1), an update pair (ID 2), and a partial deletion (ID 3).*

5.7 Putting it all together

In the previous sections, it has been shown that projection, selection, and join (with restricted join predicates) are closed under the model for partial deltas. Furthermore it has been shown that join views are maintainable if they include all non-functional dependent key attributes. Recall that dimension view definitions are assembled from these operations. We can thus conclude that dimension views are maintainable using partial deltas if all non-functional dependent key attributes are included.

6 Related work

View maintenance techniques have been adapted in several ways to deal with situations where input data is not or only partially available (cf. [GM95] for a survey). Work on

self-maintainable views aimed at maintaining a materialized view using just the deltas and the view itself, i.e. without accessing the base relations. *Partial-reference maintenance* considers only a subset of the base relations and the materialized view to be available. The *irrelevant update problem* means to decide whether an specific update leaves a view unchanged looking at the deltas and the view definition only, i.e. neither accessing the view nor the base relations. Interestingly, previous work has not considered deltas to be partial themselves, as we did here. This is probably because change capture is much less of a problem in the non-distributed environment.

Previous work on view maintenance in a *warehousing environment* [ZGMHW95, ZGMW98, AASY97, AAM+02] was focused on synchronization issues arising when base relations and materialized views reside on distributed systems. So called maintenance anomalies may occur when base relations are updated while view maintenance is performed concurrently. To prevent maintenance anomalies the Eager Compensating Algorithm (ECA) [ZGMHW95], the Strobe family of algorithms [ZGMW98], and the SWEEP algorithm [AASY97, AAM+02] have been proposed for SPJ views.

Preventing maintenance anomalies is an orthogonal problem to our work. ECA, Strobe, and Sweep make use of standard rules of algebraic differencing and tacitly assume non-partial deltas to be available. However, as we have seen, many CDC techniques used for warehousing provide incomplete deltas only. We believe that both, synchronization and handling of partial deltas are very relevant in the DWH environment. Thus, we feel our work is complementary.

We investigated view maintenance in the context of partial deltas in earlier work of ours [JD08, JD09]. In our previous work, we analyzed the impact of partial source deltas on view maintenance to understand which delta types (insert, update, delete, upsert) can still be reliably propagated. Such an analysis could, for example, reveal that a given view is maintainable w.r.t. insertions but not w.r.t. deletions for source deltas of a certain completeness. In our current work we took a different approach. We identified a restricted class of views (dimension views) that can be fully maintained using partial source deltas of any kind. We furthermore proposed a generalized algorithm for maintaining dimension views using partial deltas.

7 Conclusion

Maintenance of materialized views is an established research topic. More recently it has been proposed to use view maintenance techniques in the DWH environment where base relations and materialized views reside on different machines. However, previous work tacitly presumed that deltas captured at the sources are “complete”. We analyzed existing change capture modules and discovered that this assumption does often not hold in practice. In fact, change capture techniques may be unable to provide complete deltas or may provide partial deltas more efficiently. Thus, conventional maintenance techniques cannot be used in common DWH environments.

In this paper we studied view maintenance using partial deltas. At first, we introduced a

formal model for partial deltas. As we have shown, in general views cannot be maintained using partial deltas but there is a class of view that is maintainable. We referred to this class as dimension views, because of their close relation to dimension tables, which are typically used in star schemas. Based on our formal model for partial deltas, we then proposed a new view maintenance algorithm. To our knowledge, our algorithm is the first that allows for maintaining (a class of) views using partial deltas.

References

[AAM+02] Divyakant Agrawal, Amr El Abbadi, Achour Mostéfaoui, Michel Raynal, and Matthieu Roy. The Lord of the Rings: Efficient Maintenance of Views at Data Warehouses. In *DISC*, pages 33–47, 2002.

[AASY97] Divyakant Agrawal, Amr El Abbadi, Ambuj K. Singh, and Tolga Yurek. Efficient View Maintenance at Data Warehouses. In *SIGMOD Conference*, pages 417–427, 1997.

[Beh09] Andreas Behrend. A Classification Scheme for Update Propagation Methods in Deductive Databases. In *International Workshop on Logic in Databases (LID)*, 2009.

[GLT97] Timothy Griffin, Leonid Libkin, and Howard Trickey. An Improved Algorithm for the Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.*, 9(3):508–511, 1997.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[JD08] Thomas Jörg and Stefan Dessloch. Towards generating ETL processes for incremental loading. In *IDEAS*, pages 101–110, 2008.

[JD09] Thomas Jörg and Stefan Dessloch. Formalizing ETL Jobs for Incremental Loading of Data Warehouses. In *BTW*, pages 327–346, 2009.

[KC04] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. John Wiley & Sons, Inc., 2004.

[KP81] Shaye Koenig and Robert Paige. A Transformational Framework for the Automatic Control of Derived Data. In *VLDB*, pages 306–318, 1981.

[KR02] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., 2002.

[Mic] Microsoft. *SQL Server 2008*. http://www.microsoft.com/sqlserver/2008/.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.*, 3(3):337–341, 1991.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *SIGMOD Conference*, pages 316–327, 1995.

[ZGMW98] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency Algorithms for Multi-Source Warehouse View Maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998.