University of Kaiserslautern
Department of Computer Science
Database and Information Systems

Seminar
Database and the Cloud

Winter Semester 2013/14

Table of Contents

Incremental Recomputations in Distributed Materialized Views

Sandy Ganza

University of Kaiserslautern

Abstract. Materialized views are like data caches; they provide fast access to data and significantly increase querying speed. Materialized views are therefore important for applications such as data warehouses that deal with huge amounts of data and perform complex computations, while requiring fast response times. To ensure consistency with the underlying data, materialized views have to be maintained. A straightforward way to maintain a view is to fully re-compute it from scratch each time the base tables change. This, however, is often inefficient and costly. The other approach, which in most cases is more efficient, is to compute the changes to a view in response to incremental changes to the base tables. In this case, only those portions of the underlying data that have been changed will be re-computed. This is known as incremental view maintenance, and is the focus of this paper.
We first discuss incremental view maintenance in traditional databases, where both the materialized view and the base tables are controlled by the same database system. We then provide an overview of view maintenance policies. Finally, we analyze materialized view maintenance algorithms in distributed environments, where the base tables are decoupled from the materialized view. The resulting distributed incremental view maintenance anomalies caused by concurrent updates are described, and approaches to overcome them.

1 Introduction

In database systems, incremental recomputations help to efficiently maintain materialized views when base tables are changing. An alternative approach is to fully recompute the view from scratch, which for complex recurrent queries is often extremely costly. For such queries, it is important to pre-compute and persist them in a view, in order to guarantee fast access to data and improve the performance of the system. Traditionally, views are not materialized but rather stored just as definitions, which are then computed by the query processor every time the view is invoked. Database operators such as aggregations and joins need much time and processing power to compute and could significantly be performed more efficiently when precomputed and stored in materialized views. Moreover, materialized views can be used to build index structures. However, materialized views are susceptible to staleness when base data changes. There is, therefore, need for *view maintenance* in order to keep track of the changes made at the base tables and reflect them in the materialized view. In cases where only small parts of the base data change, the more efficient way to maintain a materialized view is to capture such changes and only modify those parts of the view that have changed. This is known as *incremental view maintenance*.

The concept of *materialized views* traditionally applies to local systems, in which both base tables and the view are under the control of a single database system. Here, the base tables know the view definition and understand view management, making it easier to propagate modifications to the view. In section 2, a mechanism to efficiently update materialized views is presented. First, a method to detect irrelevant updates is given, which narrows down the number of tuples to be considered

while maintaining the view [BLLT86]. For the relevant updates, a differential algorithm is suggested to re-evaluate the view expression by identifying the correct insertions or deletions to be made in the current view instance. It should be noted, however, that materialized views assume that base tables are updated by transactions which contain the differential update mechanism as the last operation. The third part of section 2 deals with deriving production rules for incremental view maintenance [CW91]. These are set-oriented rules that are triggered by events occurring on the base tables to enable propagation of the changed portions to the materialized view, without completely re-computing the view. How and when such rules can be automatically generated, will be studied. Finally, in section 2, the concept of self-maintainability in views is introduced [GJM96]. These views can be maintained using only the content of the materialized view and the data changes at the sources, commonly known as *deltas*. Unlike previous approaches, underlying tables are not accessed.

Section 3 introduces view maintenance policies. When implementing materialized views, it is important to know *when* and *how* to maintain the views after the base tables have been modified. Different approaches to this problem are discussed. Some policies maintain the view *immediately* when changes at the sources are detected, while others maintain the views *on-demand* when the views are queried, or *periodically* [AL80,CGL$^+$97,CKL$^+$97].

Today, the concept of materialized views has been generalized to distributed environments, where base tables and materialized views are controlled by different database systems on a network. Here, both view definitions and view management are unknown to the remote sources. In a distributed environment, modifications on the sources may need to integrate with data from other sources before being propagated to the materialized view. An example of such an environment is a *data warehouse*. A data warehouse consists of integrated data from distributed, autonomous, and possibly heterogeneous sources. Maintaining the warehouse is therefore a challenging task and may result in *distributed incremental view maintenance anomalies*, which render traditional view maintenance algorithms inapplicable to distributed environments. Distributed materialized views and their incremental recomputations are the focus of section 4. We begin by defining the term consistency as applied to materialized views in distributed systems and which levels of consistency exist [ZGmJW98]. Then, two classes of view maintenance algorithms that are based on compensation queries are discussed. *Compensation queries* are used to overcome maintenance anomalies in distributed environments. The first class of algorithms applies to single-source systems and is known as Eager Compensating Algorithms (ECA) [ZGMHW95]. The second class can also be applied multi-source environments and is known as the Strobe Algorithms [ZGmJW98,ZGMW96].

2 Incremental Recomputations in Materialized Views

Regular database views are *virtual relations* that consist of fields from several or one relation of the database. They are called virtual because the physical data is stored in *base relation* and not in the views themselves. Views can be seen as saved queries, that enable dealing with the same set of data from different perspectives by different users. When a view is invoked, the query processor replaces it by its definition, which is an SQL query, and then executes the query.
For performance reasons, however, more sophisticated queries and complex data transformations need to be *materialized* or pre-computed at the expense of always being up-to-date as is the case when directly querying the base relations. Materialized views are especially important in data warehouse applications. In the following parts of this section, we discuss the view maintenance problem and how to classify

incremental view maintenance algorithms according to given dimensions. A mechanism for efficient materialized view updates is described, and we a set of rules for incremental view maintenance is studied. Furthermore, we define the concept of self-maintainability in views.

2.1 The View Maintenance Problem - Dimensions

Since the data residing in the base relations is bound to change with time, it is important that the changes be reflected in the respective materialized views. This process is known as *view maintenance* and can be achieved in two ways: naively, the materialized views can be fully re-computed from scratch or only the portions of data that have been altered will be re-computed in the materialized views. Since in most cases the amount of changed data is less compared to the existing data, the latter option is often the most efficient and is known as *incremental re-computation*. Gupta et al.[GM95] classify *incremental view maintenance algorithms* according to the following four dimensions. The problem space resulting from the use of three of the four dimensions is shown in Figure 1.

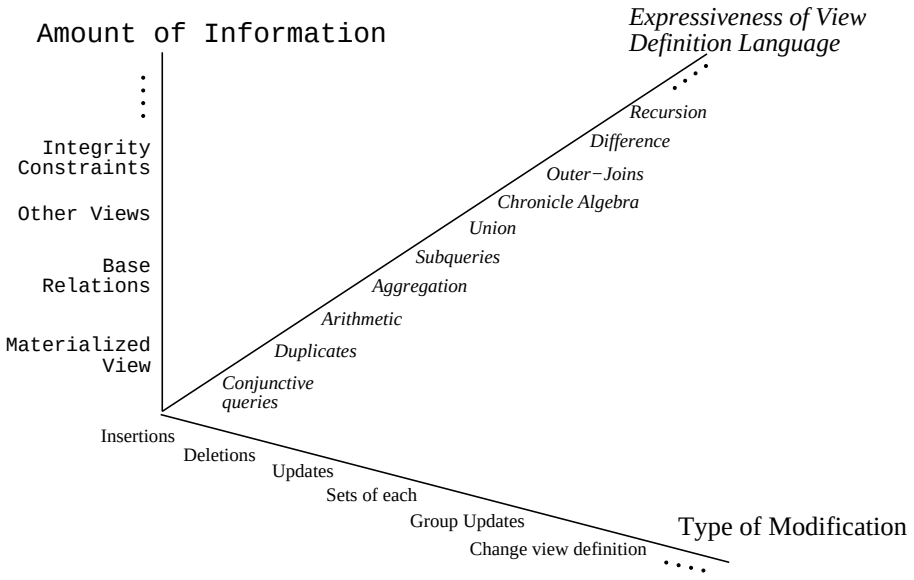

Fig. 1. The problem space (Source:[GM95])

- Information dimension. This deals with the amount of information available to determine the changes to the view. Whether there is access to the base relations, the materialized view, or whether information about constraints and keys is known, will determine which algorithm to use. Access to everything is not always guaranteed
- Modification dimension. Determines how the base relations and views are modified. The kind of modifications that can be performed and how to handle, for instance, insertions, deletions or updates determines the type of algorithm to apply. There are algorithms which implement updates directly and other which model them as deletions followed by insertions. Whether view maintenance is

performed by recomputing the whole view or only the changed portions also distinguishes algorithms
- Language dimension. Under this dimension, algorithms defer based on their syntactic representation of the view. Some might be based on relational algebra or subsets of it, SQL or a subset of it. The support for duplicates, recursion or aggregations can also be studied
- Instance dimension. This dimension can be analyzed in two ways. On the one hand, a view maintenance algorithm could be valid for all instances of a database or only for some, whereas on the other hand the algorithm could support all instances of modification or only a few instances of modification

Example 1. `course(course_code,course_grade,student_name)`

Take Example 1 of the course relation showing students' grade in respective university courses. Assuming, there is a view best_courses defined as:

$$\texttt{best\_course(course\_grade)} = \Pi_{\texttt{course\_code}}\ \sigma_{\texttt{course\_grade}>90}\texttt{(course)}$$

that returns distinct course codes in which a score of over 90 points has been obtained (duplicates are eliminated by projection).

View maintenance scenarios:
Insertion in relation course: If inserted tuple has *course_grade<=90*, then the view remains unchanged. If, however, a tuple *course(C123,95,John_Smith)* is inserted with *course_grade>90*, it is possible to update the view by applying different view maintenance algorithms depending upon which information is available: (a) If only the materialized view is available, compare this to the old materialized view to check whether the *course_grade* already exists in the view. If yes, no change is applied to the view, otherwise materialize course C123. (b) Considering only the base relation is available, if the *course* relation already has a tuple with the same *course_code* but the *course_grade* is equal or greater than that being inserted, then no changes are necessary to the materialized view. (c) If *course_code* is the key, then it must be inserted since *course_code* cannot have been present in the view.

2.2 A Mechanism for Efficient Materialized View Updates

We assume a database system that consists of *base relations* and *derived relations*. In this case, the derived relation is the materialized view, which stores its data physically in the database. When base relations change, there is need to update the materialized view such that its data is kept consistent with the base relations. Since the materialized view is basically defined by an SQL expression, this expression is completely re-evaluated to obtain the current state. This, however, is often a very slow and costly method.

A more efficient mechanism to update materialized views is proposed by Blakeley et al. [BLLT86]. In the following we describe the two main components of this mechanism: detecting updates that do not affect the view, and a method for differentially updating the materialized view. The view is expressed in relational algebra (language dimension) formed from combinations of selections, projections, and joins (SPJ expressions). Furthermore, *transactions* represent a sequence of updates on base tables, such that either all the update operations are correctly carried out or none of them is executed. A transaction may span several base relations.

Relevant and irrelevant updates. For efficiently updating the materialized view, it is important to identify those updates which , regardless of the database state,

have no effect on the state of the view. These are known as *irrelevant* updates and are helpful to avoid unnecessary re-evaluation of the view expression or at least to narrow down the number of involved tuples. The mechanism to identify irrelevant updates is illustrated by Example 2:

Example 2. Consider relations r and s with R = {A,B} and S = {C,D}. Let the view be defined as

$$v = \pi_{A,D}(\sigma_{(A>5)\wedge(C<10)\wedge(B=C)}(r \times s))$$

Let the selection condition be C(Y), where Y is a set of attributes from the relations. Then, C(A,B,C) = (A > 5) ∧ (C < 10) ∧ (B = C). Consider the following instances of the relations:

r:	A	B
	6	8
	2	20

s:	C	D
	11	30
	8	20

v:	A	D
	6	20
	2	30

Suppose tuple (7,8) is inserted into relation r, equation C(A,B,C) becomes C(7,8,C) = (7 > 5) ∧ (C < 10) ∧ (8 = C). We see that the condition C(7,8,C) is satisfiable, since relations R and S can have an instance containing tuples (7,8) and (8,δ) for some δ such that C(7,8,δ) = *True*. Inserting tuple (7,8) is hence *relevant* to view v. In a similar way, it can be seen that, for example, if tuple (1,5) was inserted into relation r, the selection condition C would be unsatisfiable regardless of the database state. Tuple (1,5) into r would therefore be *irrelevant* to view v. the same procedure applies for deletions.

Detection of relevant updates. Having known, from Example 2 what an irrelevant or relevant update is, it remains to be shown how algorithms can automatically determine the satisfiability of Boolean expressions. In general, this is in fact *NP-complete*. Rosenkrantz and Hunt [RHI80] however have shown that there is still a large class of Boolean expressions for which satisfiability can be efficiently decided. Such expressions are made of conjunctions of atomic formulae: x *op* y, x *op* c and x *op* $y + c$, where x and y are variables, c is a constant, and *op*∈{=, <, >, ≤, ≥ }. For better efficiency ≠ is not allowed in *op*. The algorithm can also be applied to disjunctions of conjunctions. Based on the algorithm of Rosenkrantz and Hunt, an algorithm for the detection of relevant updates is proposed by Blakeley et al. [BLLT86].

Differential re-evaluation of views. *Differential updates* on a materialized view decide the tuples that must be inserted or deleted from the current view instance when the base relations change. An algorithm by Blakeley et al. is briefly described here. The algorithm is designed for SPJ expressions and works under the assumption that base relations are updated by transactions and the *commit* of the transaction contains the differential update mechanism. Furthermore, it is assumed that the following information is known when invoking the differential view update mechanism: contents of base relations before the transaction is executed, inserted/deleted tuples into/from the base relation, view definition, and content of view that corresponds to that of the base relation before the transaction.

In the following, different types of views are explained. For each view we consider how its expression behaves when insert and delete operations are made to the base relations.

– **Select views:** The definition of a *select view* is $V = \sigma_{C(Y)}(R)$, with C, the selection condition which is given a Boolean expression on Y ⊆ R. If i_r and d_r

are inserted and deleted tuples respectively, the new view state v^i is given by: $v^i = v \cup \sigma_{C(y)}$ - $\sigma_{C(Y)}(d_r)$. The view is therefore updated by running insert and delete operations in sequence. This way, the view can be computed faster and cheaper than computing V from scratch, especially when the tuples being modified are far less then the total tuples that make up the view

– **Project views:** We define a *project view* by the expression $V = \pi_X(R)$, where $X \subseteq R$. Project operation is not as straightforward as the select operation. This can be shown by Example 3, when a deletion occurs in the base relation.

Example 3. Given a relation R = {A,B} and a view definition $\pi_A(R)$, with

r:

A	B
1	2
1	3
4	5

v:

A
1
4
2

The operation *delete(R,{(4, 5)})* when applied to relation r can result into the view operation *delete(V,{4})* with no problems. Trying to execute *delete(R,{(1, 2)}* on relation r though, leads to an inconsistent view if *delete(V,{1})* is to be applied there. The tuple to be deleted in the view is referenced by more than one tuple in the base relation. In a broader sense, it their could be several potential sources for a view tuple to be deleted. The reason for the inconsistence is that, over the difference operation, the distributive property of projection does not hold: $\pi_X(r_1 - r_2) \neq \pi_X(r_1)$ - $\pi_X(r_2)$.

Two solutions exist for this problem:

- Introduce an additional attribute in the view, which records for each tuple a multiplicity *counter*. When a tuple is inserted that already exists in the view, the counter value is incremented by one. On deletion it is decremented by one. The tuple can be deleted from the view when the counter becomes zero
- Alternatively, we could choose to project key attributes of the base tables in the view. In this case, every tuple in the view can be uniquely traced to its base table. Insertions and deletions are therefore safe

– **Join views:** Unlike select and project views, a *join view* is defined on multiple base relations. The view definition is given by: $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_P$. First, only insert operations to base relations are considered, then only deletions are considered, and finally changes caused by a combination of both operations are shown.

Example 4. Let R and S be two relation schemes with R = {A,B} and S = {B,C}. If a view V = R $\bowtie$ S is defined and a view v is materialized. Assume relation r is modified by inserting a set of tuples i_r. The modified relation $r^i = r \cup i_r$ and the new state of the view v^i is defined as follows:

$$v^i = r^i \bowtie s = (r \cup i_r) \bowtie s = (r \bowtie s) \cup (i_r \bowtie s)$$

If $i_v = i_r \bowtie s$, then we get $v^i = v \cup i_v$. This implies that the view can be modified by inserting only the new tuples in relation v. As a result, computing the view v^i is similar to adding i_v to v, which is cheaper than recomputing the whole join from scratch.

This idea can be expanded to insertions in an arbitrary number of base relations. The distributive property of join over union operator is used here. For a database $D = r_1, r_2, ..., r_p$ and a view definition $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_P$, assume a materialized view v and let tuples $i_{r1}, i_{r2}, ..., i_{rp}$ be inserted into relations $r_1, r_2, ..., r_p$. The new state of view v^i is computed as follows:

$$v^i = (r_1 \cup i_{r1}) \bowtie (r_2 \cup i_{r2}) \bowtie ... \bowtie (r_p \cup i_{rp})$$

We now introduce a binary variable B_i for each relation scheme $R_i : 1 \leq i \leq p$. B_i has a value *zero* when old tuples (before insertion into relation r_i) are considered, when it refers to tuples inserted into r_i since the latest materialization it acquires the value *one*. The expression v^i is illustrated by the truth table of variables B_i for $p = 3$.

B_1	B_2	B_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

From the truth table, the new view v^i can be obtained by making a union of all the 8 rows. This gives:

$$v^i = (r_1 \bowtie r_2 \bowtie r_3) \cup (r_1 \bowtie r_2 \bowtie i_{r3}) \cup (r_1 \bowtie i_{r2} \bowtie r_3) \cup (r_1 \bowtie i_{r2} \bowtie i_{r3}) \cup (i_{r1} \bowtie r_2 \bowtie r_3) \cup (i_{r1} \bowtie r_2 \bowtie i_{r3}) \cup (i_{r1} \bowtie i_{r2} \bowtie r_3) \cup (i_{r1} \bowtie i_{r2} \bowtie i_{r3})$$

We observe that not all rows are often relevant for computing the new view. The first row, for instance, only considers old tuples. Furthermore, it's not often that updates happen to all relations in a view definition. If we assume a transaction that only inserts tuples in r_1 and r_2, all rows in which B_3 has a value of one can be neglected. As a result, to compute the new view, only joins represented by rows 3, 5, and 7 are relevant. Therefore, this mechanism is cheaper than a full computation of the whole join.

Example 5. Now we consider an example that only illustrates the deletion of tuples from base relations. Let us consider two relation schemes again: $R = \{A, B\}$ and $S = \{B, C\}$, also let the view definition be $V = R \bowtie S$. If view v is materialized and relation r is updated by deletion of of tuples d_r. Let $r^i = r - d_r$. The new state v^i is given by:

$$v^i = r^i \bowtie s = (r - d_r) \bowtie s = (r \bowtie s) - (d_r \bowtie s)$$

If $d_v = d_r \bowtie s$, then $v^i = v - d_v$. We can therefore update the view by deleting the new set of tuples d_v from v. When the number of tuples of d_v are much less compared to the tuples of v, this is an easier way to compute the new materialized view v_i.

Analogously, this differential update can be expressed in terms of binary tables. In the following, we assume tuples are tagged as inserted, deleted, or old tuples. A tag of a tuple obtained from a join of two tuples is shown by the following table.

r_1	r_2	$r_1 \bowtie r_2$
insert	insert	insert
insert	delete	ignore
insert	old	insert
delete	insert	ignore
delete	delete	delete
delete	old	delete
old	insert	insert
old	delete	delete
old	old	old

When performing a join, tuples tagged as ignore are not computed. It is therefore, necessary to expand the semantics of the join operation to include tag values of tuples involved in a join. Furthermore, if there is a projection present, a count value for join tuples has to be maintained as explained earlier. The following table shows tag values for tuples resulting from selection or projection operations.

r	$\sigma_{C(Y)}(r)$	$\pi_X(r)$
insert	insert	insert
delete	delete	delete
old	old	old

Tables of 2^p rows are often not necessary in practice. Knowing which tables have been modified, only corresponding rows can be built for evaluation.

– **Select-Project-Join (SPJ) views:** The distributive property of join, select, and project over union is used to build a differential update algorithm for SPJ views. An SPJ view is defined as follows:

$$V = \pi_X(\sigma_{c(Y)}(R_1 \bowtie R_2 ... \bowtie R_P))$$

where X is a set of attributes and $C(Y)$ is a Boolean expression of the selection condition. Example 6 illustrates the idea behind the algorithm that updates SPJ views differentially.

Example 6. Consider two relation schemes $R = \{A, B\}$ and $S = \{B, C\}$ and a view $V = \pi_A(\sigma_{C(Y)}(R \bowtie S))$. If v is the materialized view and relation r is updated by insertion of tuples i_r. Assuming $r^i = r \cup i_r$. The new value of the view is given by:

$$v^i = \pi_A(\sigma_{C(Y)}(r^i \bowtie s)) = \pi_A(\sigma_{C(Y)}((r \cup i_r) \bowtie s)) = \pi_A(\sigma_{C(Y)}(r \bowtie s)) \cup \pi_A(\sigma_{C(Y)}(i_r \bowtie s)) = v \cup \pi_A(\sigma_{C(Y)}(i_r \bowtie s))$$

Let $i_v = \pi_A(\sigma_{C(Y)}(i_r \bowtie s))$, then $v^i = v \cup i_v$. It is therefore shown that only by inserting the new set of tuples into relation v, the view will be updated.

2.3 Production Rules for Incremental View Maintenance

The concept of production rules is important in database systems when dealing with a number of advanced database features such as enforcing integrity constraints, maintaining derived data, triggers, version control, and more. This section discusses production rules that are used to maintain materialized views. The rules are automatically generated basing on a view definition that is provided by a user.

When the base tables of a view are modified, maintenance rules will be triggered, often incrementally. To modify the materialized views, changes made to the base tables and stored in logical tables are required. In this section an overview of the production rule language that provides the logical tables will be given. In some cases, however, operations may require substantial recomputations. Syntactic analysis based on key information is performed to determine whether efficient view maintenance is possible or not.

When certain events or conditions are fulfilled, production rules specify manipulation operations to be carried out automatically. This principle, can therefore be used to maintain materialized views, whereby rules will be triggered in response to changing base tables. Obtaining the right set of rules for efficient view maintenance is not straightforward. There are two options to take: A complete recomputation of the view from the base tables can be done, which is often inefficient, or more efficiently, only portions of the base table that have changed are propagated to the

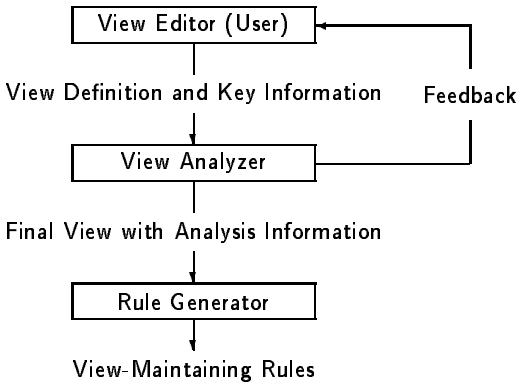

Fig. 2. Rule derivation system (Source:[CW91])

view, incrementally. Ceri and Widom [CW91] have developed a method that automatically derives incremental maintenance rules for a wide class of views. Figure 2 shows the structure of their system. It is invoked in response to view creation at compile time.

In the first step, a view definition in form of an SQL **select** expression is entered along with the keys of the base tables of the view. Next, a syntactic analysis is performed by the system on the view definition in order to answer the following two questions: Can the view contain duplicates? and are efficient view maintenance rules possible for operations on each base table referenced in the view? The analysis result is given to the user. At this point, a feedback to the system may be necessary before the rules are generated. For instance, if the system finds out that the view may contain duplicates, the user should add **distinct** to the view to preserve efficient maintenance. On the other hand, if the impossibility of efficient maintenance is detected by the system, the user may include more key information or modify the view definition. View analysis is repeated, in case of changes. The user weighs the results and the trade-offs for materializing the view or not and once the user is satisfied with the definition and properties of the view, the system generates the set of rules for maintaining the view. Rules produced include **insert**, **delete** and **update** on base tables of the view. A complete computation of the view is necessary and thereafter the view can be maintained automatically.

Overview of the production rule language. The rule language used is *set-oriented* and therefore sets of changes to the database trigger the rules. Moreover, the production rule language discussed here is SQL-based. Operations that can be maintained efficiently, changes made to base tables determine the incremental maintenance of the view by maintenance rules. *Transition tables* are used to store these changes. Only a subset of the rule language used by the view maintenance rules is described. Usual database functionality can be expressed by this set of rules, since they are fully integrated in Starburst database system[WCL91].

Rules result from database state changes called *transitions*, which are a result of executing a sequence of data manipulation operations. Production rules are described by the following syntax:

```
create rule name
when transition predicate
then action
```

[precedes *rule-list*]

The production rule is given an arbitrary *name*, whereas a *transition predicate* refers to one or more operations on tables, namely: **inserted into T**, **deleted from T**, or **updated T**. If at least one of the operations on the tables has taken place within a transition, then a rule is *triggered*. The SQL data manipulation operations that are executed when the rule is triggered are indicated by the *action*. Optionally, an ordering on the set of rules can be specified in the **precedes** list. If R_a has R_b in it's precedes list, then R_a is ordered before R_b.

In addition to the current state of the database through top-level or nested SQL select operations, a rule's action may also refer to *transition tables*. A transition table is a logical table that shows changes that occurred during a transition. The following transition tables exist: **inserted T** contains tuples of table **T** in the current state that were inserted by the transition, table **deleted T** contains pre-transition tuples of table **T** that were deleted by the transition, transition table **old updated T** contains pre-transition tuples of table **T** that were updated by the transition, and transition table, finally, transition table **new updated T** contains current values of the same tuples. *Rule assertion points* are points where rules are activated. Assertion points are found at transaction commit but users can also define other points within transactions. The user-generated database operations that are executed after the last assertion point result into a state change, which in turn creates the first relevant transition. This transition may also trigger other sets of rules. The highest ordered rule determines the triggered rule R that is then executed. R's actions become part of the initial transition to determine the rule to be triggered next. This leads to a new set of rules, in which a rule is chosen such that no other is higher in ordering. Transition predicates should also hold with respect to the transition since the action was executed or since the last rule assertion point. Rule processing terminates when the set of triggered rules is empty. When maintaining views, sometimes, there is need to consider the entire pre-transition value of a table. Although this can not be achieved directly through the rule facility, it can be derived from transition tables. The value of a table **T** at the start of the transition triggering rule is referred to as **"old T"** and is given by:

```
(T minus inserted T minus new updated T)
union deleted T union old updated T
```

Methods for view analysis and rule generation. From the rule derivation system described above, initially, the user defines the view and also provides information on keys for the view's base tables. The next step of view analysis relies on thorough information on keys from the initial step. The view analysis then computes *bound columns* from each list of table references, after this, the *safety* of the reference for each table reference is determined. A safe table reference enables generation of incremental view maintenance rules for that table. Bound columns are used by top-level tables to determine whether the view may or may not contain duplicates. In the presence of duplicates, the methods discussed here do not support efficient view maintenance. In the following, we use a view V with the form:

define view V(Col-List):
select $C_1, ..., C_n$ *from* $T_1, ..., T_m$ *where* P

where $T_1, ..., T_m$ are top-level table references,
$C_1, ..., C_n$ are columns of $T_1, ..., T_m$, and P is a predicate

Bound columns. *Bound columns* are essential for view analysis. According to [CW91], the following steps are taken to compute bound columns B(V) of the top-level table references in view V.

- Initialize B(V) to contain the columns C_1,...,C_n projected in the view definition
- Add to B(V) all columns of T_1,...,T_m such that predicate P includes an equality comparison between the column and a constant
- Repeat until B(V) is unchanged:
  - Add to B(V) all columns of T_1,...,T_m such that predicate P includes an equality comparison between the column and a column in B(V)
  - Add to B(V) all columns of any table T_i, $1 \leq i \leq m$, if B(V) includes a key for T_i

Bound columns guarantee the property: *If two tuples in the cross-product of top-level tables T_1,...,T_m satisfy predicate P and differ in their bound columns, then the tuples also must differ in view columns C_1,...,C_n.*

Duplicate analysis. Duplicate analysis is easy to perform, if the computation of bound columns for top-level table references has terminated. The system only does duplicate analysis if the view's definition does not include **distinct**. Once the possibility of occurrence of duplicates is detected by the system, the user is notified and unless V is modified to include **distinct**, maintenance rules will not be generated for V. The following theorem is true for duplicates: *If B(V) includes a key for every top-level table, then V will not contain duplicates.*

Safety analysis. For top-level table references, both safety analysis and duplicate analysis are the same. [CW91] show that if table T_i is safe, then **insert**, **update**, and **delete** operations on T_i can be propagated incrementally to V.

Rule generation. Having completed the view analysis phase, we focus on how maintenance rules are generated for top-level tables. Safe table references are studied first, followed by unsafe references. Four rules are generated for each table reference. The rules are triggered by: **insertion**, **deletion**, and two rules by **update**. How some rules can be combined and how the rule set is ordered, is discussed in the following.

Assuming T_i is a safe top-level table reference in view V. We aim at reflecting tuples inserted into T_i into V by using transition table **inserted** $\mathbf{T}_i$ instead of table T_i. The insertion theorem for top-level tables states that, such insertion cannot create duplicates in the view. Duplicates would however occur if tuples were inserted into a different top-level table applying a similar rule. It is therefore important to check if the tuple has not been inserted by a different rule already. To do this, table **inserted V** is used. The rule for **inserted** is:

```
create rule ins-T_i − V
when inserted into T_i
then insert into V
       (select C_1, ..., C_n
        from old T_1, .., inserted T_i, .., T_m
        where P and < C_1, ..., C_n > not in inserted V)
```

Similarly, when tuples are deleted from T_i, our aim is to delete them from V with help of **deleted** $\mathbf{T}_i$ instead of table T_i. The deletion theorem for top-level tables says that these tuples should no longer be in the view. It is possible, however, that

other tables in the top-level table list have been modified. For a correct set of tuples to delete from V, we therefore use pre-transition values of all the other tables. Furthermore, we use P-old which denotes predicate P with all table references **T** replaced by **old T**. The rule for **deleted** is:

```
create rule del-T_i − V
when deleted from T_i
then delete from V
     where < C_1, ..., C_n > in
        (select C_1, ..., C_n
         from old T_i, .., deleted T_i, old T_m
         where P-old)
```

Update operations on base tables, on the other side, are split into delete and/or insert operations on views. **Updated** in effect triggers two separate rules. These are similar to rules for **delete** and **insert** explained above. Delete is eventually followed by insert.
These rules are generated for each table reference, even when a table appears multiple times in the top-level table list. The combination of rules occurs when rules have identical triggering operations and their actions perform the same operation like delete or insert. When the whole set of rules has been generated, a **precedes** clause is used to order them, with deletions coming before insertions.

For unsafe top-level references T_i, the theorems given above are not valid anymore.

2.4 Self-Maintainability of Views

Views are said to be *self-maintainable* when they can be maintained using only the content of the materialized view and the data changes at the sources, commonly known as *deltas*. The underlying base tables should not be accessed. Self-maintainability in views leads to performance improvements when maintaining large views made up of data from multiple sources. The concept of self-maintainable views originates from [GJM96]. Select-Project-Join (SPJ) views are considered in response to insertions, deletions, or updates. As a result, the term self-maintainability is defined with respect to one of the three modification types, if, in response to the modification to base relations, for all database states the view can be self-maintained.

Conditions under which SPJ views are self-maintainable under insertions, deletions, and updates are presented in [GJM96] and some results are stated in the following. First, however, some terms are introduced: An attribute is said to be *distinguished* if it appears in the **select** clause of the view definition, and an attribute is said to be *exposed* if it is used in a predicate of the view definition. If it not exposed, it is said to be *unexposed*.

Self-maintainability with respect to insertions. Self-maintainability for insertions is very difficult to achieve. The following observations are true for insertions:

- It is impossible to self-maintain an SPJ view with respect to insertions. This is because inserted tuples may be tuples that haven't been in the view previously or even in the deltas. They could be tuples from any other base relation
- All SP views (do not involve joins) are self-maintainable with respect to insertions

Self-maintainability with respect to deletions. An SPJ view that joins base relations $R_1, R_2, ..., R_n$ is said to be self-maintainable with respect to deletions in R_1 if the following sufficient condition holds:

- For some key candidate of R_1, each key attribute is either retained in the view, or
- Each key attribute is equated to a constant in the view definition

The key attributes help us to identify the view tuples to be deleted.

Self-maintainability with respect to updates. To achieve self-maintainability with respect to updates, updates are modeled directly rather than in the conventional way as deletions followed by insertions. In so doing, valuable information is kept from the deleted tuple that is used to insert a new tuple into the view. This information from the deleted tuple also makes it possible for many views to be self-maintainable with respect to updates, despite not being self-maintainable with respect to insertions. Self-maintainability depends on the attributes being updated and an SPJ that joins two or more distinct relations $R_1, R_2, ..., R_n$ is said to be self-maintainable with respect to updates to R_1 *if and only if* either:

- The updated attributes are unexposed and not distinguished. In this case the updated attributes are irrelevant to the view. Or
- The updated attributes are unexposed and the view is self-maintainable with respect to updates. Here the updated attributes appear in the view and for view tuples to be updated, can be identified by their key attributes as in the case of deletions.

3 View Maintenance Policies

When implementing materialized views, it is important to know *when* and *how* to maintain the views after the base tables have been modified. To this end, several view maintenance policies have been proposed [AL80,CGL+97,CKL+97]. Although materialized views are often used to increase query performance, the maintenance process has a drawback of slowing down update transactions. This trade-off between the speed of queries and the speed of transactions has given rise to different approaches to this view maintenance problem. Some policies maintain the view *immediately* when changes at the sources are detected, while others maintain the views *on-demand* when the views are queried, or *periodically*. The process of updating the view is also known as *refresh*. Three major policies used to refresh materialized views are briefly explained below:

1. **Immediate Views:** The view is refreshed immediately within the transaction that updates the base table. The advantage of immediate maintenance is that it allows fast querying. On the other hand, however, update transactions incur a significant overhead, since each transaction can potentially update the view. Many applications can not tolerate the transaction overhead
2. **Deferred Views:** The view is refreshed when it is queried (on-demand) [CGL+97]. Unlike immediate updates, there is a separate transaction to maintain the view that is called when the view is queried or when certain conditions are fulfilled. As a result, updates are faster, and several updates may be batched together. In deferred views, however, a view may become inconsistent with it's definition. Furthermore, the querying process is slower

3. **Snapshot Views:** Snapshot views are refreshed periodically, for example, each day, week, or month [AL80]. They combine the two advantages of the previous methods by allowing fast querying and fast updates, however, the drawback is that queries may read obsolete data that doesn't match the current state of the base relations. Snapshot views are therefore suitable for those applications that may require or tolerate obsolete data and do not need to update the current state. They view the data "as of" a specific point in time

We note that immediate view maintenance may not be applicable in some environments. In data warehouses, for example, sources may lack information about the materialized view and, therefore, can not modify the update transactions so that they can refresh the materialized view. Deferred and snapshot maintenances, therefore remains the only viable options.

Determining a suitable view maintenance policy for one's view is not a straightforward process. A set of heuristics for policy selection for views with multiple heterogeneous are proposed by [ECL03]. The heuristics are based on the importance of the following service requirements for the application:

- *Consistency.* Determines whether a one-to-one correspondence between states of a materialized view and source tables is important
- *Data staleness.* A measure of elapsed time between receiving an answer to a query and the first source change to invalidate it
- *Response time.* A measure of the elapsed time until a query is answered by the view. In the worst case, the view must be fully re-computed for each update on the base tables
- *Storage capabilities.* This specifies how much data the system is capable of storing

Based on these service requirements and on the characteristics of underlying sources, several possible policies are are developed by the heuristic. Thereafter, a two-stage selection process determines the suitable maintenance policy.

4 Incremental Recomputations in Distributed Materialized Views

Until now, we have considered materialized views in traditional database environments, where both the the materialized view and the sources are controlled by the same database system. In such systems, the sources understand view management and have information regarding the view. Our focus is now turned to distributed systems like data warehouse, in which the view and the sources are decoupled. In a distributed environment, immediate view maintenance can not be achieved, since the sources are not able to invoke view maintenance through update transactions at commit. Maintaining views incrementally in distributed systems therefore poses new challenges and requires new algorithms. In this section, we begin by defining the term consistency as applied to materialized views in distributed environment and which levels of consistency exist. We then discuss two major classes of incremental view maintenance algorithms, namely: the Eager Compensating Algorithms (ECA), and the Strobe algorithms.

4.1 Consistency in Incremental View Maintenance

Maintaining the consistency of distributed materialized views is challenging, since the views may span several data sources that are autonomous and multiple sources

are bound to transactions containing multiple updates. View definitions are decoupled from data sources and this can result in anomalies when executing updates on data sources. To integrate new data into the view in a consistent way is a complex task and requires, first, to identify possible levels of consistency and correctness, and when to apply each of them. This section introduces the definition of consistency as applied to a materialized view and it's original source data. Since views may be contained in a *data warehouse*, we first describe the *states* at both the warehouse and the sources. A data warehouse integrates information from multiple sources, which may be distributed, autonomous or even heterogeneous. Furthermore, four levels of consistency for warehouse views are defined according to [ZGmJW98,ZGMHW95], in increasing order of difficulty to guarantee.

States of warehouses and sources. A warehouse state ws represents the contents of the warehouse, whereas source states ss represent the contents of source base relations. When updates occur on a system, at both the warehouse and data sources, events are triggered. These events yield different states on both sides. Let the corresponding resulting states at the warehouse be ws_1,ws_2,..,ws_f and those at the sources ss_1,ss_2,..,ss_q. ws_f and ss_q being the corresponding final states after all activity has completed. At the warehouse, the state of a materialized view V after event i has occurred is given by V(ws_i). Similarly, V(ss_i) is the state of V if it computed over the source after event i has been triggered. It is expected that each source transaction brings the sources from one consistent state to the other.

Consistency levels. The four levels of consistency for materialized views at warehouses are obtained under the assumption that at the warehouse, initially, source data and views are synchronized, or in other words: V(ss_0) = V(ws_0). Furthermore, each level entails all prior levels.

- **Convergence:** True for all finite executions, that is: V(ss_q) = V(ws_f). The view is consistent with the source data after the last update and all activity is complete
- **Weak consistency:** Convergence is true and for every state of the view, there is a valid source state, and in a corresponding order. In other words: for all ws_i there exists an ss_j such that V(ws_i) = V(ss_j)
- **Strong consistency:** Convergence holds and for every state of a view, there exists a valid source start. Furthermore, for all executions and for every pair ws_i < ws_j, there exists $ss_k \leq ss_l$ such that V(ws_i) = V(ss_k) and V(ws_j) = V(ss_l). Meaning, every state of the view corresponds to a valid source state, and in a corresponding order
- **Completeness:** Strong consistency holds, and it is also true that for each ss_i there exists a ws_j such that V(ws_j) = V(ss_i). In other words, between the states of the view and those of the states, there is a complete order-preserving mapping

In most practical warehouses, the completeness level is considered to be a requirement that is too strong and unnecessary. The view will not always need to know what exactly is happening to the base data. Sometimes, the convergence level is enough, which yields a final valid state of the warehouse although in between some states might be invalid. Most of the times, strong consistency is desirable, since it guarantees that the warehouse will always be valid with respect to some source state.

Several algorithms have been developed that support incremental view maintenance and guarantee consistency when updates are being executed in a system. We describe two of the them, below: the Strobe algorithms [ZGMW96], and the Eager Compensating Algorithms (ECA) [ZGMHW95].

4.2 Eager Compensating Algorithms

The Eager Compensating Algorithms (ECA) are incremental view maintenance algorithms that operate on a single data source. Traditional view maintenance algorithms assume that the source has information regarding the view definitions and view management. This information helps to make corresponding changes on the view when the source changes. In a warehouse environment, though, such algorithms fail because the view and the source are decoupled. The source might be a legacy system that has no understanding of the view mechanism.

In a warehousing environment, when the sources are updated, they send an update message to the warehouse. The warehouse determines which additional data may be required from the sources to make necessary changes in the views and issues queries to the sources. On receiving the queries, the sources evaluate them and replies back with answers. This process is shown in Figure 3.

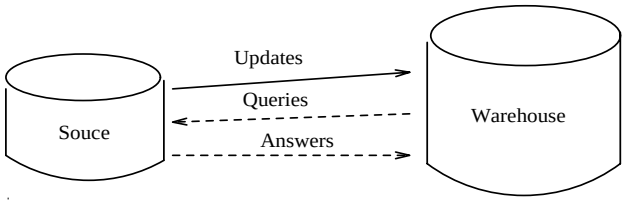

Fig. 3. Update processing in a single source model (Source:[ZGMHW95])

By the time the sources evaluate the query sent by the warehouse, it is however possible that further updates have already occurred that had not yet been captured by the warehouse before sending that query. This may result into incorrect views at the warehouse. ECA algorithms applies extra *compensating queries* that are used to eliminate such view maintenance anomalies. The view maintenance anomaly is illustrated using Example 7. For all examples in this section, the following assumptions are made: (a) relational model is used for data (b) relational algebra select-project-join queries are used for views (c) duplicates are retained in the materialized views. However, the algorithms can be extended to other data models and view specification languages.

Example 7 (View maintenance anomaly over a single source). Assume two relations r_1 and r_2 at the source with r_2 initially empty:

$$r_1\text{:}\ \begin{array}{cc} A & B \\ \hline 3 & 4 \end{array} \text{ and } r_2\text{:}\ \begin{array}{cc} B & C \\ \hline - & - \end{array}$$

The view definition is given by $V = \Pi_A(r_1 \bowtie r_2)$ and there are two consecutive updates happening at the source: $U_1 = insert(r_2,[4,8])$ and $U_2 = insert(r_1,[5,4])$. The materialized view (MV) is initially empty $MV = \emptyset$. Steps below show the update process by a conventional incremental algorithm.

1. Source executes U_1 and sends it to the warehouse
2. Warehouse receives U_1 and sends query $Q_1 = \Pi_A(r_1 \bowtie [4, 8])$ to the source
3. Source executes U_2 and sends it to the warehouse
4. Warehouse receives U_2 and sends query $Q_2 = \Pi_A([5, 4] \bowtie r_2)$ to the source
5. Source receives Q_1 and evaluates it basing on it's current relations $r_1 = ([3,4],[5,4])$ and $r_2 = ([4,8])$. The answer to this is $A_1 = ([3],[5])$ and is sent to the warehouse

6. Warehouse receives A_1 updates the view to MV $\cup$ A_1 = ([3],[5])
7. Source receives Q_2 and basing on current relations r_1 and r_2. The answer A_2 = ([5]) and is also sent to the warehouse
8. Warehouse receives A_2 and updates MV to MV $\cup$ A_2 = ([3],[5],[5])

The final view at the warehouse is incorrect. Note that if the source and warehouse were not decoupled, a conventional algorithm at the source would have yielded A_1 = ([3]) after U_1 and A_2 = ([3],[5]) after U_2. The problem results from the fact that when Q_1 is issued by the warehouse to the source, it is based on update U_1 from the source, yet by the time it is evaluated at the source update U_2 has already occurred.

Compensating Queries as a Solution. *Compensating queries* are added to queries that are sent to the source in order to offset the effect of concurrent updates. Considering Example 7, when the warehouse receives U_2 in step 4, and knows that the messages are supposed to be delivered in order, the warehouse infers that Q_1 will be evaluated on incorrect data, since otherwise, it would have received an answer to Q_1 before the notification U_2. The warehouse, therefore, sends a compensation query Q_2 to undo the effect of U_2 on A_1.

$$Q_2 = \Pi_A([5,4] \bowtie r_2) \text{ - } \Pi_A([5,4] \bowtie [4,8])$$

We observe that the A_1 received in step 6 is still A_1 = ([3],[5]), however, the compensation query leads to A_2 in step 8 being empty. Consequently, the final view is in a correct state. To achieve *strong consistency*, the ECA algorithm avoids updating the view after each answer is received from the source. This might result in the view temporarily assuming incorrect states. To implement this, the algorithm maintains a set called *unanswered query set* (UQS) which contains those queries that have been sent by the warehouse but remain unanswered when the warehouse receives the next update. Such queries see a source state that already contains the latest update. The warehouse therefore adds a compensating query to each one of the queries in the UQS. The intermediate answers received from the source are stored in a temporary relation called *COLLECT* and it is used to update the view when UQS = $\emptyset$. The ECA algorithm is not complete.

Algorithms that improve the basic ECA algorithm have been suggested: In the *ECA-Key Algorithm (ECA^K)*, the view must include keys from every base table. Having this full key information, *deletions* can directly be performed at the warehouse without sending queries to the source. Furthermore, although for *insertions* queries to the source are still required, there is no need for compensating queries anymore. On the other hand, the *ECA-Local Algorithm (ECA^L)* combines both the compensating queries of ECA with the local updates of ECA-Key Algorithm (ECA^K) and can therefore be applied to general views. The algorithm determines which updates can be processed locally or where to apply compensating queries.

4.3 The Strobe Algorithms

While the ECA algorithms operate in single-source environments, the Strobe algorithms are designed to maintain consistency as the warehouse is updated in a multi-source environment. In multi-source environments updates arriving at the warehouse may need to be integrated with updates from other sources before being materialized in the view. As several updates are made and the processing is going on, there is potential for the warehouse to become inconsistent.

An important factor to consider when processing updates at the warehouse is whether and how sources run transactions. Updates that are made at sources that run transactions, should be treated as atomic units. There are three main transaction scenarios that are considered depending on the scope of the transaction.

The three scenarios are: (a) *single update transactions*, in which each update comprises it's own transaction, (b) *source-local transactions*, in which a sequence of actions at the same source form one transaction, and finally (c) *global transactions*, in which actions performed at multiple sources comprise global transactions. The Strobe algorithms are designed to achieve a specific level of correctness for each of the transaction scenarios. As a result there are three algorithms, namely: the Strobe algorithm, which achieves strong consistency for the single update transactions. The Transaction-Strobe algorithm, which achieves strong consistency for the source-local transactions, and the Global-Strobe algorithm that achieves strong consistency for global transactions. Before delving into the algorithms, the following assumptions are made:

- We assume the projection list of the view contains key attributes for each relation. This is a major difference to the ECA algorithms and allows Strobe algorithms to be self-maintainable with respect to deletions
- Updates at one source could initiate a multi-source query Q at the warehouse. The warehouse decides the order of the sources to be visited. Function *next_source*(Q) returns the pair (x, Q^i) where x is the next source to visit, and Q^i the portion of Q to be evaluated there. A_i is the answer from the source to Q_i. Query $Q(A_i)$ is the remaining query after answer A_i has been incorporated into query Q
- Function *source_evaluate* returns the final result answer A after looping to evaluate the next portion of query Q. function *source_evaluate* may, however, return an incorrect answer when concurrent transactions exit at the sources. Strobe algorithms use compensation mechanisms to overcome this anomaly
- For simplicity only deletions and insertions are considered by the algorithms

Strobe: In order to maintain a consistent materialized view MV at the warehouse at all times, the Strobe algorithm does not perform each update directly on the view, instead the updates, although processed immediately, are kept in an actions lists AL. The actions in the list are applied to MV only when a consistent state can be guaranteed. At this time, there are no "pending" queries present and all updates have been processed.

The list of actions consists of both insert and delete actions but how are these actions generated? A *delete action* is straightforward, since it is generated when a deletion is received at the warehouse. On the other hand, an *insert action* may require issuing a *source_evaluate* procedure first. As stated earlier, when a query Q is being answered, updates may change source data and lead to inconsistency. To overcome this, a set called *pending*(Q) is kept that stores all those updates that occur during query processing. After the compensation of Q has terminated, an insert action for MV can now be generated and added to AL.

Example 8 (Strobe). Let UQS be the unanswered query set as applies to the ECA algorithms. Operation *key_delete*(R, U_i) deletes tuples from relation R whose key attributes are the same as U_i. V(U) is the view expression V with tuple U substituted for U's relation. If we have relations r_1, r_2 and r_3 residing on sources x, y and z respectively, let view V be defined as $V = r_1 \bowtie r_2 \bowtie r_3$. The initial relations are given by:

$$r_1\text{:}\ \begin{array}{cc} A & B \\ \hline 1 & 2 \end{array}\ r_2\text{:}\ \begin{array}{cc} B & C \\ \hline - & - \end{array}\ \text{and}\ r_3\text{:}\ \begin{array}{cc} C & D \\ \hline 3 & 4 \end{array}$$

Initially the materialized view is MV = $\emptyset$. Assume two update at the source: U_1 = *insert*(r_2,[2,3]) and U_2 = *delete*(r_1,[1,2]). The Strobe algorithm follows the steps:

1. AL = (). Warehouse receives U_1 from source y. Query $Q_1 = r_1 \bowtie [2,3] \bowtie r_3$ is generated. First, $Q_1^1 = r_1 \bowtie [2,3]$ is sent to source x

2. Warehouse receives A_1^1 = [1,2,3] from source x. $Q_1^2 = [1,2,3] \bowtie r_3$ is sent to z for evaluation
3. Warehouse receives U_2 from source x. U_2 is added to *pending*(Q_1) and *key_delete*(Mv, U_2) is added to AL. Therefore AL = (*key_delete*(MV,U_2)).
4. Warehouse receives A_1^2 = [1,2,3,4] from source z but since *pending*(Q) is not empty, the warehouse first applies *key_delete*(A_1^2, U_2). and as a result $A_2 = \emptyset$. Nothing, therefore, is to be added to AL and there are no pending queries. MV is therefore updated by the warehouse by applying AL = (*key_delete*(MV, U_2)). The final view state is MV = $\emptyset$ and it is correct and strongly consistent.

Strobe in this way, avoids the anomaly that a conventional algorithm would face. A conventional algorithm fails to apply the deletion since when the delete update arrives at warehouse, MV is empty and there is no mechanism of tracking pending updates. The Strobe algorithm provides strong consistency in single-update transaction environments.

Transaction-Strobe: Transaction-Strobe (T-Strobe) provides strong consistency for source-local transactions. All updates performed by one transaction are batched into a single unit for processing. This also reduces the number of message interchange between sources and the warehouse. The following definitions are are true for T-Strobe algorithms:

- The *update list* of transaction T is UL(T) and it contains both inserts and deletions performed by T. The *insertion list* of T, which is a subset of the update list is represented by IL(T). whereby IL(T) $\subseteq$ UL(T)
- key(U_i) represents key attributes of the inserted or deleted tuple U_i. U_i and U_j are equal when key(U_i) = key(U_j)

At the source, T-Strobe issues the same actions as Strobe algorithm, however, the behavior is different at the warehouse. The following warehouse actions take place in T-Strobe algorithm: An optimization step is carried out initially, where pairs of insertions and deletions that are removed, in which the same tuple was first inserted but later removed. Remaining deletions are added to AL. And finally, one query is generated for all insertions by the warehouse. Compensation is carried out as before, for those deletions that arrive after the query has been generated.

Example 9 (T-Strobe). This simple example shows how, for source-local transactions, Strobe may only achieve convergence where T-Strobe ensures strong consistency. The reason being that Strobe has no mechanism to handle transactions. This means Strobe will eventually provide a correct view but after going through invalid intermediate states.

Assume a relation r_1: $= \begin{array}{cc} A & B \\ 1 & 2 \end{array}$. If the view definition is V = r_1 and attribute A is the key, initially MV = ([1, 2]). If there is one source transaction: T_1 = (*delete*(r_1, [1,2]), *insert*(r_1, [3, 4])). Strobe algorithm would first add the deletion to AL but since there are no pending updates, AL will be applied to MV, resulting into MV = $\emptyset$. This is an incorrect intermediate state, since it is not consistent with r_1 both before or after T_1. After that the insertion is processed and yields a correct view MV = ([3, 4]).

For T-Strobe, since it understands local transactions, it will only update MV when both updates in the transaction have been processed. MV therefore is directly updated to the correct view MV = ([3, 4]).

Global-Strobe: For global transactions, T-Strobe only achieves weak consistency, while GLobal-Strobe (G-Strobe) guarantees strong consistency. G-Strobe resembles T-Strobe, however, it only updates MV when three conditions have all been met. Of the three conditions, T-Strobe only requires condition one. Assume TT is the set of all transaction identifiers after the last update of MV.

1. UQS = $\emptyset$
2. If transaction T_i in TT depends on another transaction T_j, T_j is also in TT
3. All updates in TT have been received and processed

5 Conclusion

In this paper, the importance of incremental recomputations in both traditional databases systems and in distributed systems was discussed. In the absence of incremental maintenance mechanisms, materialized views have to be fully re-computed from scratch, which in most cases is inefficient. We saw different approaches to the view maintenance problem in both environments, however it is evident that in the problem space of the view maintenance problem, many gaps still remain that have not yet been explored. Each of the four dimensions introduced still has unconsidered points.

Production rules for incremental view maintenance were covered and it was shown that it is possible to automatically derive production rules to incrementally maintain a materialized view basing on a user's view definition in SQL. The full power of SQL queries can, nevertheless, not yet be exploited.

The concept of self-maintainable views was introduced. A guidance was provided on how to define views such that they can be maintained without accessing any underlying database. In practice, not all database modifications are self-maintainable. Insertions in particular are impossible to self-maintain when joins are involved. Deletions are self-maintainable when attribute keys are kept track of.

View maintenance policies were described. These policies tell us when and how to maintain the materialized view when the underlying data changes. It was shown that the decision to apply a certain policy is often a compromise between the speed of the queries on the one hand, and the speed of the update transactions on the other.

View maintenance policies also highlighted a fundamental difference between traditional databases and distributed database systems. In distributed systems, sources may lack information about the materialized view and, therefore, can not modify the update transactions so that they can refresh the materialized view immediately.

The fact that in distributed systems materialized views and sources are decoupled, means that view maintenance algorithms defined for traditional databases can not be applied to distributed systems. Algorithms in a distributed system should be able to handle the distributed view maintenance anomalies resulting from concurrent updates that were presented. Compensation approaches are used to overcome such anomalies. The ECA and Strobe algorithms were described. Furthermore, a definition for consistency as applied to materialized views in distributed systems was given.

A newer and more challenging field of application for incremental recomputations that was not handled in this paper is about supporting materialized data integration [JD09,Jör13,DHW+08], for example, within Extract-Transform-Load (ETL) or MapReduce environments. Such environments are not relational based and differ also in many other aspects from the relational environments we considered in this paper. However, an initial approach has been made by [Jör13] to transfer incremental recomputational techniques into the ETL and MapReduce environments.

References

[AL80] Michel E. Adiba and Bruce G. Lindsay. Database snapshots. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*, VLDB '80, pages 86–91. VLDB Endowment, 1980.

[BLLT86] Jose A. Blakeley, Per ke Larson, Per-Ake Larson, and Frank Wm. Tompa. Efficiently updating materialized views. pages 61–71, 1986.

[CGL+97] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In *In SIGMOD*, pages 469–480, 1997.

[CKL+97] Latha Colby, Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Supporting multiple view maintenance policies. In *IN THE ACM SIGMOD CONF*, pages 405–416, 1997.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *In Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, 1991.

[DHW+08] Stefan Dessloch, Mauricio A. Hernández, Ryan Wisnesky, Ahmed Radwan, and Jindan Zhou. Orchid: Integrating schema mapping and etl. In *ICDE*, pages 1307–1316, 2008.

[ECL03] H. Engström, S. Chakravarthy, and B. Lings. Maintenance policy selection in heterogeneous data warehouse environments: A heuristics-based approach. In *Proceedings of the 6th ACM International Workshop on Data Warehousing and OLAP*, DOLAP '03, pages 71–78, New York, NY, USA, 2003. ACM.

[GJM96] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *EDBT*, pages 140–144, 1996.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[JD09] Thomas Jörg and Stefan Deßloch. Formalizing etl jobs for incremental loading of data warehouses. In *BTW*, pages 327–346, 2009.

[Jör13] Thomas Jörg. *Incremental Recomputations in Materialized Data Integration.* PhD thesis, 2013.

[RHI80] Daniel J. Rosenkrantz and Harry B. Hunt III. Processing conjunctive predicates and queries. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*, VLDB '80, pages 64–72. VLDB Endowment, 1980.

[WCL91] Jennifer Widom, Roberta Jo Cochrane, and Bruce G. Lindsay. Implementing set-oriented production rules as an extension to starburst. In *In Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 275–285, 1991.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *IN PROCEEDINGS OF SIGMOD*, pages 316–327, 1995.

[ZGmJW98] Yue Zhuge, Hector Garcia-molina, Janet, and Janet L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Distributed and Parallel Databases*, 6:7–40, 1998.

[ZGMW96] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. The strobe algorithms for multi-source warehouse consistency. In *PDIS*, pages 146–157, 1996.