Maintenance of Materialized Views: Problems, Techniques, and Applications

In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views & Data Warehousing*, 18(2), June 1995

Ashish Gupta
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
ashish@almaden.ibm.com

Inderpal Singh Mumick
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
mumick@research.att.com

Abstract

In this paper we motivate and describe materialized views, their applications, and the problems and techniques for their maintenance. We present a taxonomy of view maintenance problems based upon the class of views considered, upon the resources used to maintain the view, upon the types of modifications to the base data that are considered during maintenance, and whether the technique works for all instances of databases and modifications. We describe some of the view maintenance techniques proposed in the literature in terms of our taxonomy. Finally, we consider new and promising application domains that are likely to drive work in materialized views and view maintenance.

1 Introduction

What is a view? A view is a derived relation defined in terms of base (stored) relations. A view thus defines a function from a set of base tables to a derived table; this function is typically recomputed every time the view is referenced.

What is a materialized view? A view can be materialized by storing the tuples of the view in the database. Index structures can be built on the materialized view. Consequently, database accesses to the materialized view can be much faster than recomputing the view. A materialized view is thus like a cache – a copy of the data that can be accessed quickly.

Why use materialized views? Like a cache, a materialized view provides fast access to data; the speed difference may be critical in applications where the query rate is high and the views are complex so that it is not possible to recompute the view for every query. Materialized views are useful in new applications such as data warehousing, replication servers, chronicle or data recording systems [JMS95], data visualization, and mobile systems. Integrity constraint checking and query optimization can also benefit from materialized views.

What is view maintenance? Just as a cache gets *dirty* when the data from which it is copied is updated, a materialized view gets dirty whenever the underlying base relations are modified. The process of updating a materialized view in response to changes to the underlying data is called view maintenance.

What is incremental view maintenance? In most cases it is wasteful to maintain a view by recomputing it from scratch. Often it is cheaper to use the heuristic of inertia (only a part of the view changes in response to changes in the base relations) and thus compute only the changes in the view to update its materialization. We stress that the above is only a heuristic. For example, if an entire base relation is deleted, it may be cheaper to recompute a view that depends on the deleted relation (if the new view will quickly evaluate to an empty relation) than to compute the changes to the view. Algorithms that compute changes to a view in response to changes to the base relations are called *incremental view maintenance* algorithms, and are the focus of this paper.

Classification of the View Maintenance Problem There are four dimensions along which the view maintenance problem can be studied:

- Information Dimension: The amount of information available for view maintenance. Do you have access to all/some the base relations while doing the maintenance? Do you have access to the materialized view? Do you know about integrity constraints and keys? We note that the amount of information used is orthogonal to the incrementality of view maintenance. Incrementality refers to a computation that only computes that part of the view that has changed; the information dimension looks at the data used to compute the change to the view.

- Modification Dimension: What modifications can the view maintenance algorithm handle? Insertion and deletion of tuples to base relations? Are updates to tuples handled directly or are they modeled as deletions followed by insertions? What about changes to the view definition? Or sets of modifications?

- Language Dimension: Is the view expressed as a select-project-join query (also known as a SPJ views or as a conjunctive query), or in some other subset of relational algebra? SQL or a subset of SQL? Can it have duplicates? Can it use aggregation? Recursion? General recursions, or only transitive closure?

- Instance Dimension: Does the view maintenance algorithm work for all instances of the database, or only for some instances of the database? Does it work for all instances of the modification, or only for some instances of the modification? Instance information is thus of two types - *database instance*, and *modification instance*.

We motivate a classification of the view maintenance problem along the above dimensions through examples. The first example illustrates the information and modification dimensions.

Example 1: (Information and Modification Dimensions) Consider relation

$$\texttt{part}(\texttt{part\_no}, \texttt{part\_cost}, \texttt{contract})$$

listing the cost negotiated under each contract for a part. Note that a part may have a different price under each contract. Consider also the view `expensive_parts` defined as:

$$\texttt{expensive\_parts}(\texttt{part\_no}) = \Pi_{\texttt{part\_no}}\ \sigma_{\texttt{part\_cost} > 1000}(\texttt{part})$$

The view contains the **distinct** part numbers for parts that cost more than \$1000 under at least one contract (the projection discards duplicates). Consider maintaining the view when a tuple is inserted into relation `part`. If the inserted tuple has $\texttt{part\_cost} \leq 1000$ then the view is unchanged.

However, say $\texttt{part}(p1, 5000, c15)$ is inserted that does have cost > 1000. Different view maintenance algorithms can be designed depending upon the information available for determining if $p1$ should be inserted into the view.

- The materialized view alone is available: Use the old materialized view to determine if `part_no` already is present in the view. If so, there is no change to the materialization, else insert part $p1$ into the materialization.

- The base relation part alone is available: Use relation part to check if an existing tuple in the relation has the same part_no but greater or equal cost. If such a tuple exists then the inserted tuple does not contribute to the view.
- It is known that part_no is the key: Infer that part_no cannot already be in the view, so it must be inserted.

Another view maintenance problem is to respond to deletions using only the materialized view. Let tuple part($p1, 2000, c12$) be deleted. Clearly part $p1$ must be in the materialization, but we cannot delete $p1$ from the view because some other tuple, like part($p1, 3000, c13$), may contribute $p1$ to the view. The existence of this tuple cannot be (dis)proved using only the view. Thus there is no algorithm to solve the view maintenance problem for deletions using only the materialized view. Note, if the relation part was also available, or if the key constraint was known, or if the counts of number of view tuple derivations were available, then the view could be maintained. ■

With respect to the information dimension, note that the view definition and the actual modification always have to be available for maintenance. With respect to the modification dimension, updates typically are not treated as an independent type of modification. Instead, they are modeled as a deletion followed by an insertion. This model loses information thereby requiring more work and more information for maintaining a view than if updates were treated independently within a view maintenance algorithm [BCL89, UO92, GJM94].

The following example illustrates the other two dimensions used to characterize view maintenance.

Example 2: (Language and Instance Dimensions) Example 1 considered a view definition language consisting of selection and projection operations. Now let us extend the view definition language with the join operation, and define the view supp_parts as the equijoin between relations supp(supp_no, part_no, price) and part ($\bowtie_{\texttt{part\_no}}$ represents an equijoin on attribute part_no):

$$\texttt{supp\_parts(part\_no)} = \Pi_{\texttt{part\_no}}(\texttt{supp} \bowtie_{\texttt{part\_no}} \texttt{part})$$

The view contains the **distinct** part numbers that are supplied by at least one supplier (the projection discards duplicates). Consider using only the old contents of supp_parts for maintenance in response to insertion of part($p1, 5000, c15$). If supp_parts already contains part_no $p1$ then the insertion does not affect the view. However, if supp_parts does not contain $p1$, then the effect of the insertion cannot be determined using only the view.

Recall that the view expensive_parts was maintainable in response to insertions to part using only the view. In contrast, the use of a join makes it impossible to maintain supp_parts in response to insertions to part when using only the view.

Note, view supp_parts is maintainable if the view contains part_no $p1$ but not otherwise. Thus, the maintainability of a view depends also on the particular instances of the database and the modification. ■

Figure 1 shows the problem space defined by three of the four dimensions; namely the information, modification, and language dimensions. The instance dimension is not shown here so as to keep the figure manageable. There is no relative ordering between the points on each dimension; they are listed in arbitrary order. Along the language dimension, *chronicle algebra* [JMS95] refers to languages that operate over ordered sequences that may not be stored in the database (see Section 4.3). Along the modification dimension, *group updates* [GJM94] refers to insertion of several tuples using information derived from a single deleted tuple.

We study maintenance techniques for different points in the shown problem space. For each point in this 3-D space we may get algorithms that apply to all database and modification instances or that may work only for some instances of each (the fourth dimension).

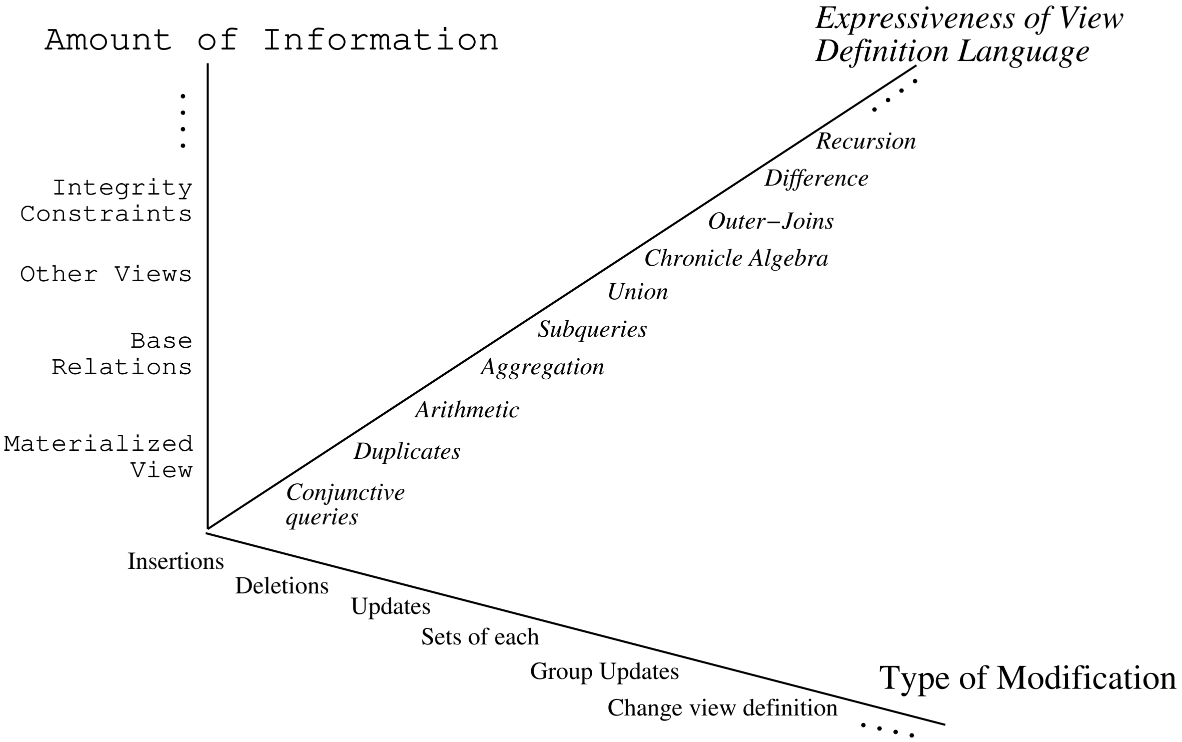

Figure 1: The problem space

Paper Outline

We study the view maintenance problem with respect to the space of Figure 1 using the "amount of information" as the first discriminator. For each point considered on the information dimension, we consider the languages for which view maintenance algorithms have been developed, and present selected algorithms in some detail. Where appropriate, we mention how different types of modifications are handled differently. The algorithms we describe in some detail address the following points in the problem space.

- (Section 3:) Information dimension: Use *Full Information* (all the underlying base relations and the materialized view). Instance dimension: Apply to all instances of the database and all instances of modifications. Modification dimension: Apply to all types of modifications. Language dimension: Consider the following languages —
  - SQL views with duplicates, `UNION`, negation, and aggregation (*e.g.* `SUM, MIN`).
  - Outer-join views.
  - Recursive Datalog or SQL views with `UNION`, stratified aggregation and negation, but no duplicates.
- (Section 4:) Information dimension: Use *partial information* (materialized view and key constraints – views that can be maintained without accessing the base relations are said to be *self-maintainable*). Instance dimension: Apply to all instances of the database and all instances of modifications. Language dimension: Apply to SPJ views. Modification dimension: Consider the following types of modifications —
  - Insertions and Deletions of tuples.
  - Updates and group updates to tuples.

  We also discuss maintaining SPJ views using the view and some underlying base relations.

2 The Idea Behind View Maintenance

Incremental maintenance requires that the change to the base relations be used to compute the change to the view. Thus, most view maintenance techniques treat the view definition as a mathematical formula and apply a differentiation step to obtain an expression for the change in the view. We illustrate through an example:

Example 3: (Intuition) Consider the base relation $\texttt{link}(S, D)$ such that $\texttt{link}(a, b)$ is true if there is a link from source node a to destination node b. Define view hop such that $\texttt{hop}(c, d)$ is true if c is connected to d using two links, via an intermediate node:

$$\mathcal{D}: \ \texttt{hop}(X, Y) = \Pi_{X,Y}(\texttt{link}(X, V) \ \bowtie_{V=W} \ \texttt{link}(W, Y))$$

Let a set of tuples $\Delta(\texttt{link})$ be inserted into relation link. The corresponding insertions $\Delta(\texttt{hop})$ that need to be made into view hop can be computed by mathematically differentiating definition $\mathcal{D}$ to obtain the following expression:

$$\begin{aligned}\Delta(\texttt{hop}) = \Pi_{X,Y}((\Delta(\texttt{link})(X, V) \ &\bowtie_{V=W} \ \texttt{link}(W, Y)) \cup \\ (\texttt{link}(X, V) \ &\bowtie_{V=W} \ \Delta(\texttt{link})(W, Y)) \cup \\ (\Delta(\texttt{link})(X, V) \ &\bowtie_{V=W} \ \Delta(\texttt{link})(W, Y)))\end{aligned}$$

The second and third terms can be combined to yield the term $\texttt{link}^{\nu}(X, V) \ \bowtie_{V=W} \ \Delta(\texttt{link})(W, Y)$ where $\texttt{link}^{\nu}$ represents relation link with the insertions, *i.e.*, $\texttt{link} \cup \Delta(\texttt{link})$. ■

In the above example, if tuples are deleted from link then too the same expression computes the deletions from view hop. If tuples are inserted into and deleted from relation link, then $\Delta(\texttt{hop})$ is often computed by separately computing the set of deletions $\Delta^{-}(\texttt{hop})$ and the set of insertions $\Delta^{+}(\texttt{hop})$ [QW91, HD92]. Alternatively, by differently tagging insertions and deletions they can be handled in one pass as in [GMS93].

3 Using Full Information

Most work on view maintenance has assumed that all the base relations and the materialized view are available during the maintenance process, and the focus has been on efficient techniques to maintain views expressed in different languages – starting from select-project-join views and moving to relational algebra, SQL, and Datalog, considering features like aggregations, duplicates, recursion, and outer-joins. The techniques typically differ in the expressiveness of the view definition language, in their use of key and integrity constraints, and whether they handle insertions and deletions separately or in one pass (Updates are modeled as a deletion followed by an insertion). The techniques all work on all database instances for both insertions and deletions. We will classify these techniques broadly along the language dimension into those applicable to nonrecursive views, those applicable to outer-join views, and those applicable to recursive views.

3.1 Nonrecursive Views

We describe the counting algorithm for view maintenance, and then discuss several other view maintenance techniques that have been proposed in the literature.

The counting Algorithm [GMS93]: applies to SQL views that may or may not have duplicates, and that may be defined using `UNION`, negation, and aggregation. The basic idea in the counting algorithm is to keep a count of the number of derivations for each view tuple as extra information in the view. We illustrate the counting algorithm using an example.

Example 4: Consider view hop from Example 3 now written in SQL.

CREATE VIEW hop(S, D) **as**
(**select distinct** l1.S, l2.D **from** link l1, link l2 **where** l1.D = l2.S)

Given link $= \{(a,b),(b,c),(b,e),(a,d),(d,c)\}$, the view hop evaluates to $\{(a,c),(a,e)\}$. The tuple hop(a,e) has a unique derivation. hop(a,c) on the other hand has two derivations. If the view had duplicate semantics (did not have the **distinct** operator) then hop(a,e) would have a count of 1 and hop(a,c) would have a count of 2. The counting algorithm pretends that the view has duplicate semantics, and stores these counts.

Suppose the tuple link(a,b) is deleted. Then we can see that hop can be recomputed as $\{(a,c)\}$. The counting algorithm infers that one derivation of each of the tuples hop(a,c) and hop(a,e) is deleted. The algorithm uses the stored counts to infer that hop(a,c) has one remaining derivation and therefore only deletes hop(a,e), which has no remaining derivation. ■

The counting algorithm thus works by storing the number of alternative derivations, count(t), of each tuple t in the materialized view. This number is derived from the multiplicity of tuple t under duplicate semantics [Mum91, MS93]. Given a program T defining a set of views $V_1, \ldots, V_k$, the counting algorithm uses the differentiation technique of Section 2 to derive a program T_Δ. The program T_Δ uses the changes made to base relations and the old values of the base and view relations to produce as output the set of changes, $\Delta(V_1), \ldots, \Delta(V_k)$, that need to be made to the view relations. In the set of changes, insertions are represented with positive counts, and deletions by negative counts. The count value for each tuple is stored in the materialized view, and the new materialized view is obtained by combining the changes $\Delta(V_1), \ldots, \Delta(V_k)$ with the stored views $V_1, \ldots, V_k$. Positive counts are added in, and negative counts are subtracted. A tuple with a count of zero is deleted. The count algorithm is optimal in that it computes exactly those view tuples that are inserted or deleted. For SQL views counts can be computed at little or no cost above the cost of evaluating the view for both set and duplicate semantics. The counting algorithm works for both set and duplicate semantics, and can be made to work for outer-join views (Section 3.2).

Other Counting Algorithms: [SI84] maintain select, project, and equijoin views using counts of the number of derivations of a tuple. They build data structures with pointers from a tuple τ to other tuples derived using the tuple τ. [BLT86] use counts just like the counting algorithm, but only to maintain SPJ views. Also, they compute insertions and deletions separately, without combining them into a single set with positive and negative counts. [Rou91] describes "ViewCaches," materialized views defined using selections and one join, that store only the TIDs of the tuples that join to produce view tuples.

Algebraic Differencing: introduced in [Pai84] and used subsequently in [QW91] for view maintenance differentiates algebraic expressions to derive the relational expression that computes the change to an SPJ view without doing redundant computation. [GLT95] provide a correction to the minimality result of [QW91], and [GL95] extend the algebraic differencing approach to multiset algebra with aggregations and multiset difference. They derive two expressions for each view; one to compute the insertions into the view, and another to compute the deletions into the view.

The Ceri-Widom algorithm [CW91]: derives production rules to maintain selected SQL views - those without duplicates, aggregation, and negation, and those where the view attributes functionally determine the key of the base relation that is updated. The algorithm determines the SQL query needed to maintain the view, and invokes the query from within a production rule.

Recursive Algorithms: The algorithms described in Section 3.3 for recursive views also apply to nonrecursive views.

3.2 Outer-Join Views

Outer joins are important in domains like data integration and extended relational systems [MPP+93]. View maintenance on outer-join views using the materialized view and all base relations has been discussed in [GJM94].

In this section we outline the algorithm of [GJM94] to maintain incrementally full outer-join views. We use the following SQL syntax to define a view V as a full outer-join of relations R and S:

`CREATE` **view** `V` **as select** $X_1, \ldots, X_n$ **from** R **full outer join** S **on** $g(Y_1, \ldots, Y_m)$

where $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_m$ are lists of attributes from relations R and S. $g(Y_1, \ldots, Y_m)$ is a conjunction of predicates that represent the outer-join condition. The set of modifications to relation R is denoted as $\Delta(R)$, which consists of insertions $\Delta^+(R)$ and deletions $\Delta^-(R)$. Similarly, the set of modifications to relation S is denoted as $\Delta(S)$. The view maintenance algorithm rewrites the view definition to obtain the following two queries to compute $\Delta(V)$.

(a): **select** $X_1, \ldots, X_n$
from $\Delta(R)$ **left outer join** S
on $g(Y_1, \ldots, Y_m)$

(b): **select** $X_1, \ldots, X_n$
from R^ν **right outer join** $\Delta(S)$
on $g(Y_1, \ldots, Y_m)$.

R^ν represents relation R after modification. All other references in queries (a) and (b) refer either to the pre-modified extents or to the modifications themselves. Unlike with SPJ views queries (a) and (b) do not compute the entire change to the view, as explained below.

Query (a) computes the effect on V of changes to relation R. Consider a tuple r^+ inserted into R and its effect on the view. If r^+ does not join with any tuple in s, then r^+.`NULL` (r^+ padded with nulls) has to be inserted into view V. If instead, r^+ does join with some tuple s in S, then $r^+.s$ (r^+ joined with tuple s) is inserted into the view. Both these consequences are captured in Query (a) by using the left-outer-join. However, query (a) does not compute a possible side effect if r^+ does join with some tuple s. The tuple `NULL`.s (s padded with nulls) may have to be deleted from the view V if `NULL`.s is in the view. This will be the case if previously tuple s did not join with any tuple in R.

Similarly, a deletion r^- from R not only removes a tuple from the view, as captured by Query (a), but may also precipitate the insertion of a tuple `NULL`.s if before deletion r^- is the only tuple that joined with s. Query (b) handles the modifications to table S similar to the manner in which query (a) handles the modifications to table R, with similar possible side-effects. The algorithm of [GJM94] handles these side effects.

3.3 Recursive Views

Recursive queries or views often are expressed using rules in Datalog [Ull89], and all the work on maintaining recursive views has been done in the context of Datalog. We describe the DRed **(Deletion and Rederivation)** algorithm for view maintenance, and then discuss several other recursive view maintenance techniques that have been proposed in the literature.

The DRed Algorithm [GMS93]: applies to Datalog or SQL views, including views defined using recursion, `UNION`, and stratified negation and aggregation. However, SQL views with duplicate semantics cannot be maintained by this algorithm. The DRed algorithm computes changes to the view relations in three steps. First, the algorithm computes an overestimate of the deleted derived tuples: a tuple t is in this overestimate if the changes made to the base relations invalidate *any* derivation of t. Second, this overestimate is pruned by removing (from the overestimate) those tuples that have alternative derivations in the new database. A version of the original view restricted to compute only the tuples in the overestimated set is used to do the pruning. Finally, the new tuples that need to be inserted are computed using the partially updated materialized view and the insertions made to the base relations. The algorithm can also maintain materialized views incrementally when rules defining derived relations are inserted or deleted. We illustrate the DRed algorithm using an example.

Example 5: Consider the view hop defined in Example 4. The DRed algorithm first deletes tuples hop(a, c) and hop(a, e) since they both depend upon the deleted tuple. The DRed algorithm then looks for alternative derivations for each of the deleted tuples. hop(a, c) is rederived and reinserted into the materialized view in the second step. The third step of the DRed algorithm is empty since no tuples are inserted into the `link` table. ■

None of the other algorithms discussed in this section handle the same class of views as the DRed algorithm; the most notable differentiating feature being aggregations. However, some algorithms derive more efficient solutions for special subclasses.

The PF (Propagation/Filtration) algorithm [HD92]: is very similar to the DRed algorithm, except that it propagates the changes made to the base relations on a relation by relation basis. It computes changes in *one* derived relation due to changes in *one* base relation, looping over all derived and base relations to complete the view maintenance. In each loop, an algorithm similar to the delete/prune/insert steps in DRed is executed. However, rather than running the deletion step to completion before starting the pruning step, the deletion and the pruning steps are alternated after each iteration of the semi-naive evaluation. Thus, in each semi-naive iteration, an overestimate for deletions is computed and then pruned. This allows the PF algorithm to avoid propagating some tuples that occur in the over estimate after the first iteration but do not actually change. However, the alternation of the steps after each semi-naive iteration also causes some tuples to be rederived several times. In addition, the PF algorithm ends up fragmenting computation and rederiving changed and deleted tuples again and again. [GM93] presents improvements to the PF algorithm that reduce rederivation of facts by using memoing and by exploiting the stratification in the program. Each of DRed and the PF algorithms can do better than the other by a factor of n depending on the view definition (where n is the number of base tuples in the database). For nonrecursive views, the DRed algorithm always works better than the PF algorithm.

The Kuchenhoff algorithm [Kuc91]: derives rules to compute the difference between consecutive database states for a stratified recursive program. The rules generated are similar in spirit to those of [GMS93]. However, some of the generated rules (for the *depends* predicates) are not safe, and the delete/prune/insert three step technique of [GMS93, HD92] is not used. Further, when dealing with positive rules, the Kuchenhoff algorithm does not discard duplicate derivations that are guaranteed not to generate any change in the view as early as the DRed algorithm discards the duplicate derivations.

The Urpi-Olive algorithm [UO92]: for stratified Datalog views derives transition rules showing how each modification to a relation translates into a modification to each derived relation, using existentially quantified subexpressions in Datalog rules. The quantified subexpressions may go through negation, and can be eliminated under certain conditions. Updates are modeled directly; however since keys need to be derived for such a modeling, the update model is useful mainly for nonrecursive views.

Counting based algorithms can sometimes be used for recursive views. The counting algorithm of [GKM92] can be used effectively only if every tuple is guaranteed to have a finite number of derivations[1], and even then the computation of counts can significantly increase the cost of computation. The BDGEN system [NY83] uses counts to reflect not all derivations but only certain types of derivations. Their algorithm gives finite even counts to all tuples, even those in a recursive view, and can be used even if tuples have infinitely many derivations.

Transitive Closures [DT92] derive nonrecursive programs to update right-linear recursive views in response to insertions into the base relation. [DS93] give nonrecursive programs to update the transitive closure of specific kinds of graphs in response to insertions and deletions. The algorithm does not apply to all graphs or to general recursive programs. In fact, there does not exist a nonrecursive program to maintain the transitive closure of an arbitrary graph in response to deletions from the graph [DLW95].

Nontraditional Views [LMSS95a] extends the DRed algorithm to views that can have nonground tuples. [WDSY91] give a maintenance algorithm for a rule language with negation in the head and body of rules, using auxiliary information about the number of certain derivations of each tuple. They do not consider aggregation, and do not discuss how to handle recursively defined relations that may have an infinite number of derivations.

4 Using Partial Information

As illustrated in the introduction, views may be maintainable using only a subset of the underlying relations involved in the view. We refer to this information as *partial information*. Unlike view maintenance using full information, a view is not always maintainable for a modification using only partial information. Whether the view can be maintained may also depend upon whether the modification is an insertion, deletion, or update. So the algorithms focus on checking whether the view can be maintained, and then on how to maintain the view.

We will show that treating updates as a distinct type of modification lets us derive view maintenance algorithms for updates where no algorithms exist for deletions+insertions.

4.1 Using no Information: Query Independent of Update

There is a lot of work on optimizing view maintenance by determining when a modification leaves a view unchanged [BLT86, BCL89, Elk90, LS93]. This is known as the "query independent of update", or the "irrelevant update" problem. All these algorithms provide checks to determine whether a particular modification will be irrelevant. If the test succeeds, then the view stays unaffected by the modification. However, if the test fails, then some other algorithm has to be used for maintenance.

[BLT86, BCL89] determine irrelevant updates for SPJ views while [Elk90] considers irrelevant updates for Datalog. Further, [LS93] can determine irrelevant updates for Datalog with negated base relations and arithmetic inequalities.

4.2 Using the Materialized View: Self-Maintenance

Views that can be maintained using only the materialized view and key constraints are called *self-maintainable* views in [GJM94]. Several results on self-maintainability of SPJ and outer-join views in response to insertions, deletions, and updates are also presented in [GJM94]. Following [GJM94], we define:

[1]An algorithm to check finiteness appears in [MS93, MS94].

Definition 1: (Self Maintainability With Respect to a Modification Type) A view V is said to be self-maintainable with respect to a modification type (insertion, deletion, or update) to a base relation R if for all database states, the view can be self-maintained in response to all instances of a modification of the indicated type to the base relation R.

Example 6: Consider view `supp_parts` from Example 2 that contains all **distinct** `part_no` supplied by at least one supplier. Also, let `part_no` be the key for relation `part` (so there can be at most one contract and one `part_cost` for a given part).

If a tuple is deleted from relation `part` then it is straightforward to update the view using only the materialized view (simply delete the corresponding `part_no` if it is present). Thus, the view is self-maintainable with respect to deletions from the `part` relation.

By contrast, let tuple supp$(s1, p1, 100)$ be deleted when the view contains tuple $p1$. The tuple $p1$ cannot be deleted from the view because `supp` may also contain a tuple supp$(s2, p1, 200)$ that contributes $p1$ to the view. Thus, the view is not self-maintainable with respect to deletions from `supp`. In fact, the view is not self-maintainable for insertions into either `supp` or `part`. ∎

Some results from [GJM94] are stated after the following definitions.

Definition 2: (Distinguished Attribute) An attribute A of a relation R is said to be distinguished in a view V if attribute A appears in the **select** clause defining view V.

Definition 3: (Exposed Attribute) An attribute A of a relation R is said to be exposed in a view V if A is used in a predicate. An attribute that is not exposed is referred to as being non-exposed.

Self-Maintainability With Respect to Insertions and Deletions [GJM94] shows that most SPJ views are not self-maintainable with respect to insertions, but they are often self-maintainable with respect to deletions and updates. For example:

- An SPJ view that takes the join of two or more distinct relations is not self-maintainable with respect to insertions.
- An SPJ view is self-maintainable with respect to deletions to R_1 if the key attributes from each occurrence of R_1 in the join are either included in the view, or are equated to a constant in the view definition.
- A left or full outer-join view V defined using two relations R and S, such that:
  - The keys of R and S are distinguished, and
  - All exposed attributes of R are distinguished.

  is self-maintainable with respect to all types of modifications to relation S.

Self-Maintainability With Respect to Updates By modeling an update independently and not as a deletion+insertion we retain information about the deleted tuple that allows the insertion to be handled more easily.

Example 7: Consider again relation `part(part_no, part_cost, contract)` where `part_no` is the key. Consider an extension of view `supp_parts`:

$$\texttt{supp\_parts}(\texttt{supp\_no}, \texttt{part\_no}, \texttt{part\_cost}) = \Pi_{\texttt{part\_no}}(\texttt{supp} \bowtie_{\texttt{part\_no}} \texttt{part})$$

The view contains the `part_no` and `part_cost` for the parts supplied by each supplier. If the `part_cost` of a part $p1$ is updated then the view is updated by identifying the tuples in the view that have `part_no` $= p1$ and updating their `part_cost` attribute. ■

The ability to self-maintain a view depends upon the attributes being updated. In particular, updates to non-exposed attributes are self-maintainable when the key attributes are distinguished. The complete algorithm for self-maintenance of a view in response to updates to non-exposed attributes is described in [GJM94] and relies on (a) identifying the tuples in the current view that are potentially affected by the update, and (b) computing the effect of the update on these tuples.

The idea of self-maintenance is not new — Autonomously computable views were defined by [BCL89] as the views that can be maintained using only the materialized view for all database instances, but for a given modification instance . They characterize a subset of SPJ views that are autonomously computable for insertions, deletions, and updates, where the deletions and updates are specified using conditions. They do not consider views with self-joins or outer-joins, do not use key information, and they do not consider self-maintenance with respect to all instances of modifications. The characterization of autonomously computable views in [BCL89] for updates is inaccurate — For instance, [BCL89] determines, incorrectly, that the view "**select** X **from** r(X)" is not autonomously computable for the modification "Update(R(3) to R(4))".

Instance Specific Self-Maintenance For insertions and deletions only, a database instance specific self-maintenance algorithm for SPJ views was discussed first in [BT88]. Subsequently this algorithm has been corrected and extended in [GB95].

4.3 Using Materialized View and Some Base Relations: Partial-reference

The partial-reference maintenance problem is to maintain a view given only a subset of the base relations and the materialized view. Two interesting subproblems here are when the view and all the relations except the modified relation are available, and when the view and modified relation are available.

Modified Relation is not Available (Chronicle Views) A chronicle is an ordered sequence of tuples with insertion being the only permissible modification [JMS95]. A view over a chronicle, treating the chronicle as a relation, is called a chronicle view. The chronicle may not be stored in its entirety in a database because it can get very large, so the chronicle view maintenance problem is to maintain the chronicle view in response to insertions into the chronicle, but without accessing the chronicle. Techniques to specify and maintain such views efficiently are presented in [JMS95].

Only Modified Relation is Available (Change-reference Maintainable) Sometimes a view may be maintainable using only the modified base relation and the view, but without accessing other base relations. Different modifications need to be treated differently.

Example 8: Consider maintaining view `supp_parts` using relation `supp` and the old view in response to deletion of a tuple t from relation `supp`. If t.`part_no` is the same as the `part_no` of some other tuple in `supp` then the view is unchanged. If no remaining tuple has the same `part_no` as tuple t then we can deduce that no supplier supplies t.`part_no` and thus the part number has to be deleted from the view. Thus, the view is change-reference-maintainable.

A similar claim holds for deletions from `part` but not for insertions into either relation. ■

Instance Specific Partial-reference Maintenance [GB95, Gup94] give algorithms that successfully maintain a view for some instances of the database and modification, but not for others. Their algorithms derive conditions to be tested against the view and/or the given relations to check if the information is adequate to maintain the view.

5 Applications

New and novel applications for materialized views and view maintenance techniques are emerging. We describe a few of the novel applications here, along with a couple of traditional ones.

Fast Access, Lower CPU and Disk Load: Materialized views are likely to find applications in any problem domain that needs quick access to derived data, or where recomputing the view from base data may be expensive or infeasible. For example, consider a retailing database that stores several terabytes of point of sale transactions representing several months of sales, and supports queries giving the total number of items sold in each store for each item the company carries. These queries are made several times a day, by vendors, store managers, and marketing people. By defining and materializing the result, each query can be reduced to a simple lookup on the materialized view; consequently it can be answered faster, and the CPU and disk loads on the system are reduced. View maintenance algorithms keep the materialized result current as new sale transactions are posted.

Data Warehousing: A database that collects and stores data from several databases is often described as a data warehouse.

Materialized views provide a framework within which to collect information into the warehouse from several databases without copying each database in the warehouse. Queries on the warehouse can then be answered using the materialized views without accessing the remote databases. Provisioning, or changes, still occurs on the remote databases, and are transmitted to the warehouse as a set of modifications. Incremental view maintenance techniques can be used to maintain the materialized views in response to these modifications. While the materialized views are available for view maintenance, access to the remote databases may be restricted or expensive. Self-Maintainable views are thus useful to maintain a data warehouse [GJM94]. For cases where the view is not self-maintainable and one has to go to the remote databases, besides the cost of remote accesses, transaction management is also needed [ZG$^+$95].

Materialized views are used for data integration in [ZHKF95, GJM94]. Objects that reside in multiple databases are integrated to give a larger object if the child objects "match." Matching for relational tuples using outer-joins and a *match* operator is done in [GJM94], while more general matching conditions are discussed in [ZHKF95]. The matching conditions of [ZHKF95] may be expensive to compute. By materializing the composed objects, in part or fully, the objects can be used inexpensively.

[LMSS95b] presents another model of data integration. They consider views defined using some remote and some local relations. They materialize the view partially, without accessing the remote relation, by retaining a reference to the remote relation as a constraint in the view tuples. The model needs access to the remote databases during queries and thus differs from a typical warehousing model.

Chronicle Systems: Banking, retailing, and billing systems deal with a continuous stream of transactional data. This ordered sequence of transactional tuples has been called a chronicle [JMS95]. One characteristic of a chronicle is that it can get very large, and it can be beyond the capacity of any database system to even store, far less access, for answering queries. Materialized views provide a way to answer queries over the chronicle without accessing the chronicle.

Materialized views can be defined to compute and store summaries of interest over the chronicles (the balance for each customer in a banking system, or the profits of each store in the retailing system). View maintenance techniques are needed to maintain these summaries as new transactions are added to the chronicle, but without accessing the old entries in the chronicle [JMS95].

Data Visualization: Visualization applications display views over the data in a database. As the user changes the view definition, the display has to be updated accordingly. An interface for such queries in a real estate system is reported in [WS93], where they are called *dynamic queries*. Data archaeology [BST+93] is a similar application where an archaeologist discovers rules about data by formulating queries, examining the results, and then changing the query iteratively as his/her understanding improves. By materializing a view and incrementally recomputing it as its definition changes, the system keeps such applications interactive. [GMR95] studies the "view adaptation problem," *i.e.*, how to incrementally recompute a materialized view in response to changes to the view definition.

Mobile Systems: A common query in a personal digital assistant (PDA) is of the form "Which freeway exits are within a 5 mile radius". One model of computation sends the query to a remote server that uses the position of the PDA to answer the query and sends the result back to the PDA. When the PDA moves and asks the same query, data transmission can be reduced by computing only the change to the answer and designing the PDA to handle answer differentials.

Integrity Constraint Checking: Most static integrity constraints can be represented as a set of views such that if any of the views is nonempty then the corresponding constraint is violated. Then checking constraints translates to a view maintenance problem. Thus, view maintenance techniques can be used to incrementally check integrity constraints when a database is modified. The expression to check integrity constraints typically can be simplified when the constraint holds before the modification, *i.e.*, the corresponding views initially are empty [BC79, Nic82, BB82, BMM92, LST87, CW90].

Query Optimization: If a database system maintains several materialized views, the query optimizer can use these materialized views when optimizing arbitrary queries, even when the queries do not mention the views. For instance, consider a query in a retailing system that wants to compute the number of items sold for each item. A query optimizer can optimize this query to access a materialized view that stores the number of items sold for each item and store, and avoid access to a much larger sales-transactions table.

[RSU95, LMSS95a] discuss the problem of answering a conjunctive query (SPJ query) given a set of conjunctive view definitions. Optimization of aggregation queries using materialized views is discussed in [CKPS95, DJLS95, GHQ95]. The view adaptation results of [GMR95] can be used to optimize a query using only one materialized view.

6 Open Problems

This section describes some open problems in view maintenance, in the context of Figure 1. Many points on each of the three dimensions remain unconsidered, or even unrepresented. It is useful to extend each dimension to unconsidered points and to develop algorithms that cover entirely the resulting space because each point in the space corresponds to a scenario of potential interest.

View maintenance techniques that use all the underlying relations, *i.e.* full-information, have been studied in great detail for large classes of query languages. We emphasize the importance of developing comprehensive view maintenance techniques that use different types of partial information. For instance:

- Use information on functional dependencies, multiple materialized views, general integrity constraints, horizontal/vertical fragments of base relations (*i.e.*, simple views).
- Extend the view definition language to include aggregation, negation, outer-join for all instances of the other dimensions. The extensions are especially important for using partial information.
- Identify *subclasses* of SQL views that are maintainable in an instance independent fashion.

The converse of the view maintenance problem under partial information, as presented in Section 4 is to identify the information required for efficient view maintenance of a given view (or a set of views). We refer to this problem as the "information identification (II)" problem. Solutions for view maintenance with partial information indirectly apply to the II problem by checking if the given view falls into one of the classes for which partial-information based techniques exist. However, direct and more complete techniques for solving the II problem are needed.

An important problem is to implement and incorporate views in a database system. Many questions arise in this context. When are materialized views maintained – before the transaction that updates the base relation commits, or after the transaction commits? Is view maintenance a part of the transaction or not? Should the view be maintained before the update is applied to the base relations, or afterwards? Should the view be maintained after each update within the transaction, or after all the updates? Should active rules (or some other mechanism) be used to initiate view maintenance automatically or should a user start the process? Should alternative algorithms be tried, based on a cost based model to choose between the options? Some existing work in this context is in [NY83, CW91, GHJ94, RC+95]. [CW91] considers using production rules for doing view maintenance and [NY83] presents algorithms in the context of a deductive DB system. [GHJ94] does not discuss view maintenance but discusses efficient implementation of deltas in a system that can be used to implement materialized views. [RC+95] describes the ADMS system that implements and maintains simple materialized views, "ViewCaches," in a multi-database environment. The ADMS system uses materialized views in query optimization and addresses questions of caching, buffering, access paths, *etc.*.

The complexity of view maintenance also needs to be explored. The dynamic complexity classes of [PI94] and the incremental maintenance complexity of [JMS95] characterize the computational complexity of maintaining a materialized copy of the view. [PI94] show that several recursive views have a first order dynamic complexity, while [JMS95] define languages with constant, logarithmic, and polynomial incremental maintenance complexity.

Acknowledgements

We thank H. V. Jagadish, Leonid Libkin, Dallan Quass, and Jennifer Widom for their insightful comments on the technical and presentation aspects of this paper.

References

[BB82] P. A. Bernstein and B. T. Blaustein. Fast Methods for Testing Quantified Relational Calculus Assertions. In *SIGMOD 1982*, pages 39–50.

[BBC80] P. A. Bernstein, B. T. Blaustein, and E. M. Clarke. Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data. In *6th VLDB, 1980*, pages 126–136.

[BC79] Peter O. Buneman and Eric K. Clemons. *Efficiently Monitoring Relational Databases.* In *ACM Transactions on Database Systems*, Vol 4, No. 3, 1979, 368-382.

[BCL89] J. A. Blakeley, N. Coburn, and P. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Transactions on Database Systems*, 14(3):369–400, 1989.

[BLT86] J. A. Blakeley, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *SIGMOD 1986.*

[BMM92] F. Bry, R. Manthey, and B. Martens. Integrity Verification in Knowledge Bases. In *Logic Programming, LNAI 592*, pages 114–139, 1992.

[BST+93] R. J. Brachman, et al.. Integrated support for data archaeology. In *International Journal of Intelligent and Cooperative Information Systems*, 2:159–185, 1993.

[BT88] J. A. Blakeley and F. W. Tompa. Maintaining Materialized Views without Accessing Base Data. In *Information Systems*, 13(4):393–406, 1988.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, K. Shim. Query Optimization in the presence of Materialized Views. In *11th IEEE Intl. Conference on Data Engineering*, 1995.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In *VLDB* 1990.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB* 1991.

[DJLS95] S. Dar, H.V. Jagadish, A. Y. Levy, and D. Srivastava. Answering SQL queries with aggregation using views. Technical report, AT&T, 1995.

[DLW95] G. Dong, L. Libkin and L. Wong. On Impossibility of Decremental Recomputation of Recursive Queries in Relational Calculus and SQL. In *Proc. of the Intl. Wksp. on DB Prog. Lang*, 1995.

[DS93] G. Dong and J. Su. Incremental and Decremental Evaluation of Transitive Closure by First-Order Queries. In Proceedings of the 16th Australian Computer Science Conference, 1993.

[DT92] G. Dong and R. Topor. Incremental Evaluation of Datalog Queries. In *ICDT*, 1992.

[Elk90] C. Elkan. Independence of Logic Database Queries and Updates. In *9th PODS*, pages 154–160, 1990.

[GHJ94] S. Ghandeharizadeh, R. Hull, and D Jacobs. Heraclitus[Alg,C]: Elevating Deltas to be First-Class Citizens in a Database Programming Language. Tech. Rep. # USC-CS-94-581, USC, 1994.

[GB95] A. Gupta and J. A. Blakeley. Maintaining Views using Materialized Views . *Unpublished document.*

[GHQ95] A. Gupta, V. Harinarayan and D. Quass. Generalized Projections: A Powerful Approach to Aggregation. In *VLDB*, 1995.

[GJM94] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. Technical Memorandum 113880-941101-32, AT&T Bell Laboratories, November 1994.

[GKM92] A. Gupta, D. Katiyar, and I. S. Mumick. Counting Solutions to the View Maintenance Problem. In *Workshop on Deductive Databases, JICSLP*, 1992.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD 1995*.

[GLT95] T. Griffin and L. Libkin and H. Trickey. A correction to "Incremental recomputation of active relational expressions" by Qian and Wiederhold. To appear in *IEEE TKDE*.

[GM93] A. Gupta and I. S. Mumick. Improvements to the PF Algorithm. TR STAN-CS-93-1473, Stanford.

[GMR95] A. Gupta, I. Singh Mumick, and K. A. Ross. Adapting materialized views after redefinitions. In *Columbia University TR CUCS-010-95*, March 1995. Also in *SIGMOD 1995*, pages 211-222.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD 1993*, pages 157–167. (Full version in AT&T technical report # 9921214-19-TM.)

[GSUW94] A. Gupta, S. Sagiv, J. D. Ullman, and J. Widom. Constraint Checking with Partial Information. In *13th PODS*, 1994, pages 45-55.

[Gup94] A. Gupta. *Partial Information Based Integrity Constraint Checking.* Ph.D. Thesis, Stanford (CS-TR-95-1534).

[HD92] J. V. Harrison and S. Dietrich. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Workshop on Deductive Databases, JICSLP*, 1992.

[JMS95] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues in the chronicle data model. In *14th PODS*, pages 113–124, 1995.

[KSS87] R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. In *VLDB*, 1987.

[Kuc91] V. Kuchenhoff. On the Efficient Computation of the Difference Between Consecutive Database States. In *DOOD, LNCS 566*, 1991.

[LMSS95a] A. Y. Levy and A. O. Mendelzon and Y. Sagiv and D. Srivastava. Answering Queries Using Views. In *PODS* 1995, pages 95-104.

[LMSS95b] J. Lu, G. Moerkotte, J. Schu, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *SIGMOD 1995*, pages 340–351.

[LS93] A.Y. Levy and Y. Sagiv. Queries Independent of Updates. In *19th VLDB*, pages 171–181, 1993.

[LST87] J.W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity Constraint Checking in Stratified Databases. *Journal of Logic Programming*, 4(4):331–343, 1987.

[MPP+93] B. Mitschang, H. Pirahesh, P. Pistor, B. Lindsay, and N. Sudkamp. SQL/XNF - Processing Composite Objects as Abstractions over Relational Data. In *Proc. of* 9^{th} *IEEE ICDE*, 1993.

[MS93] I. S. Mumick and O. Shmueli. Finiteness properties of database queries. In *Advances in Database Research: Proc. of the* 4^{th} *Australian Database Conference*, pages 274–288, 1993.

[MS94] I. S. Mumick and O. Shmueli. Universal Finiteness and Satisfiability. In *PODS 1994*, pages 190-200.

[Mum91] I. S. Mumick. *Query Optimization in Deductive and Relational Databases.* Ph.D. Thesis, Stanford University, Stanford, CA 94305, USA, 1991.

[Nic82] J. M. Nicolas. Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, 1982.

[NY83] J. M. Nicolas and Yazdanian. An Outline of BDGEN: A Deductive DBMS. In *Information Processing*, pages 705–717, 1983.

[Pai84] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In *Advances in Database Theory*, pages 170–209, Plenum Press, New York, 1984.

[PI94] S. Patnaik and N. Immerman. Dyn-fo: A parallel, dynamic complexity class. In *PODS*, 1994.

[QW91] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. In *IEEE TKDE*, 3(1991), pages 337-341.

[RSU95] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995, pages 105–112.

[RC+95] N. Roussopoulos, C. Chun, S. Kelley, A. Delis, and Y. Papakonstantinou. The ADMS Project: Views "R" Us. In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2), June 1995.

[Rou91] N. Roussopoulos. The Incremental Access Method of View Cache: Concept, Algorithms, and Cost Analysis. In *ACM-TODS*, 16(3):535-563, 1991.

[SI84] O. Shmueli and A. Itai. *Maintenance of Views.* In *SIGMOD 1984*, pages 240–255.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, Vol 2. Computer Science Press.

[UO92] T. Urpi and A. Olive. A Method for Change Computation in Deductive Databases. In *VLDB* 1992.

[WDSY91] O. Wolfson, H. M. Dewan, S. J. Stolfo, and Y. Yemini. *Incremental Evaluation of Rules and its Relationship to Parallelism.* In *SIGMOD 1991*, pages 78–87.

[WS93] C. Williamson and B. Shneiderman. The Dynamic HomeFinder: evaluating Dynamic Queries in a real- estate information exploration system. In Ben Shneiderman, editor, *Sparks of Innovation in Human-Computer Interaction.* Ablex Publishing Corp, 1993.

[ZHKF95] G. Zhou, R. Hull, R. King, J-C. Franchitti. Using Object Matching and Materialization to Integrate Heterogeneous Databases. In *Proc. of* 3^{rd} *Intl. Conf. on Cooperative Info. Sys.*, 1995, pp. 4–18.

[ZG+95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD 1995*, pages 316–327.