Bulletin of the Technical Committee on

Data Engineering

June, 1995 Vol. 18 No. 2

IEEE Computer Society

Letters

Special Issue on Materialized Views and Data Warehousing

Conference and Journal Notices

Letter from the Editor-in-Chief

About this issue

Data warehousing has become an increasingly important part of how users do information management. It solves two problems: how to off-load decision support applications from the on-line transaction system; and how to bring together information from multiple sources so as to provide a consistent database source for decision support queries. The fact that data in the warehouse is replicated data raises interesting issues in how it should be maintained and what level of timeliness and consistency should be guaranteed. The current issue of the Bulletin explores this subject, providing overviews and specifics about ongoing research efforts. I would like to thank Jennifer Widom for assembling the issue on this timely and interesting topic. Jennifer deserves a substantial dose of extra credit for this endeavor, as she has fit this in around having a baby.

State of the Bulletin

This is the second issue of the Bulletin for which there will not be hardcopy distribution. This situation does a disservice to editors and authors who have worked hard on producing the Bulletin and who deserve to see their work get the widest possible distribution. It is also a disservice to the membership of the Technical Committee, some of whom have difficulty accessing the electronic form of the Bulletin , either because of lack of ftp connectivity to the net, or because of difficulties in transmission, or because of an inability to print or display the postscript that is sent. I consider the current situation to be highly unsatisfactory. I believe that continued publication of the Bulletin is at risk should this persist.

In previous years, hardcopy publication was made possible by using the income from the Data Engineering Conference. Last year there was no surplus from that conference and hence the Technical Committee on Data Engineering has no budget for this year. We, the TC on Data Engineering, have requested permission to charge a hardcopy subscription fee for the Bulletin as a way to cover the cost of its printing and distribution. The IEEE Computer Society has never consented to this.

The only way to ensure the continued publication of the Bulletin is to provide a solid and durable way of covering the cost of hardcopy publication. Publication of the Bulletin stopped once before because such support was not forthcoming. The risk is now very large that that will be the result again. If you want to save the Bulletin, you need to express to the Computer Society your desire to see the Bulletin continue publication. The Technical Committee operates under the Technical Activities Board (TAB), and the person in charge of the TAB is Computer Society VP Paul Borrill (paul.borrill@eng.sun.com).

News Flash: The Computer Society has once again provided the Technical Committee on Data Engineering with funding, including funding for the Bulletin. While this is not the preferred long term solution, I want to thank the TAB for approving of the budget which enables us to continue the hardcopy distribution of the Bulletin. This is a token of their recognition of the usefulness of the Bulletin and their desire to be supportive of it. This is much appreciated.

David Lomet
Microsoft Corporation
lomet@microsoft.com

Letter from the Special Issue Editor

The topics of this special issue are *materialized views* and *data warehousing*. Interest in view maintenance has been reemerging in the database research community, while data warehousing has become one of the key buzzwords in industry.

Data warehousing encompasses frameworks, architectures, algorithms, tools, and techniques for bringing together selected data from multiple databases or other information sources into a single repository—called a *data warehouse*—suitable for direct querying or analysis. One can view the problem of data warehousing as the problem of maintaining, in the warehouse, a materialized view or views of the relevant data stored in the original information sources. Conventional view maintenance techniques cannot always be used, however: The "views" stored in data warehouses tend to be more complicated than conventional views, not necessarily expressible in a standard view definition language (such as SQL), and often involve historical information that may not remain in the original sources. Furthermore, the information sources updating the "base data" often are independent of the warehouse in which the "view" is stored, and base data may be transformed ("scrubbed") before it is integrated into the warehouse. There are a number of rich problems to be solved in adapting view maintenance techniques to the data warehousing environment.

Data warehousing is especially important in industry today because of a need for enterprises to gather all of their information into a single place for in-depth analysis, and the desire to decouple such analysis from their on-line transaction processing systems. Analytical processing that involves very complex queries (often with aggregates) and few or no updates—usually termed *decision support*—is one of the primary uses of data warehouses, hence the terms data warehousing and decision support often are found together, sometimes interchanged. Other relevant terms include *data mining*, *on-line analytical processing* (*OLAP*), and *multidimensional analysis*, which are (in my view) refinements or subclasses of decision support. Since decision support often is the goal of data warehousing, clearly warehouses may be tuned for decision support, and perhaps vice-versa. Nevertheless, decision support being a very broad area, we have focused this special issue on the problem of data warehousing.

The first paper in the issue, by Gupta and Mumick, provides a comprehensive survey of work in materialized view maintenance, suggests remaining open problems, and describes a number of relevant applications. The second paper, by Roussopoulos et al., is an overview of the *ADMS* project at the University of Maryland. A variety of important contributions in the database field have arisen from the ADMS project over the last several years. ADMS can in fact be viewed as a data warehousing system, and as such has been well ahead of its time. The third paper, by Zhou et al., describes the *H2O* project underway at the University of Colorado. H2O elegantly combines the data warehousing ("in-advance") approach to information integration with the traditional "on-demand" approach (where information is not collected and integrated until queries are posed). The last paper, by Hammer et al., describes the *WHIPS* data warehousing project at Stanford University, providing an overview of the project architecture along with discussion of technical issues such as information source monitoring and consistency of warehouse data.

I regret that no commercially-oriented papers appear in this issue. I attempted to solicit appropriate papers, but apparently at this point the market is too hot and the relevant companies too small. Nevertheless, I certainly hope you enjoy this special issue.

Jennifer Widom
Stanford University
widom@cs.stanford.edu

Maintenance of Materialized Views: Problems, Techniques, and Applications

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 $\texttt{part}(p1, 2000, c12)$ be deleted. Clearly part $p1$ must be in the materialization, but we cannot delete $p1$ from the view because some other tuple, like $\texttt{part}(p1, 3000, c13)$, 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 modelled 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 $\texttt{supp}(\texttt{supp\_no}, \texttt{part\_no}, \texttt{price})$ and part ($\bowtie_{\texttt{part\_no}}$ represents an equijoin on attribute part_no):

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

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 $\texttt{part}(p1, 5000, c15)$. 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).

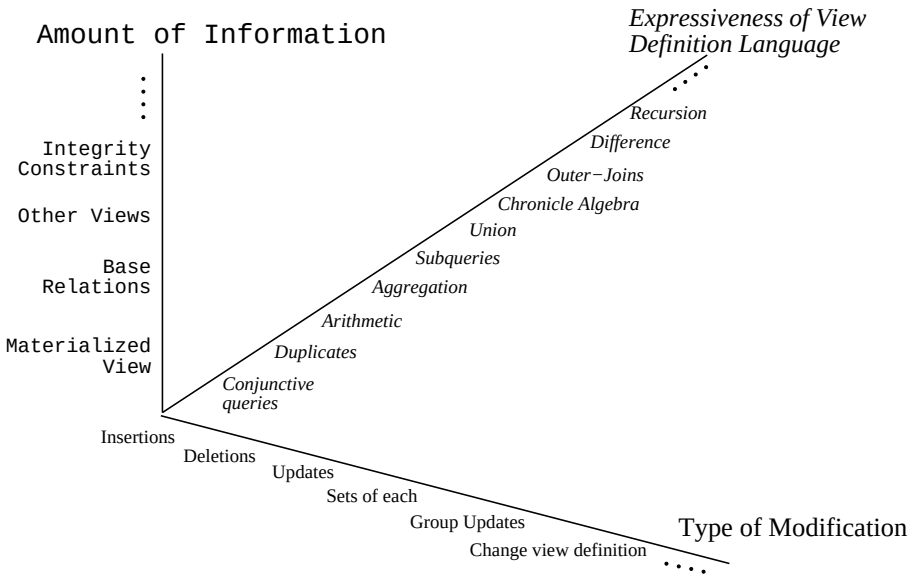


Figure 1: The problem space

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 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:

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

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:

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

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 $\texttt{supp}(s1, p1, 100)$ be deleted when the view contains tuple $p1$. The tuple $p1$ cannot be deleted from the view because `supp` may also contain a tuple $\texttt{supp}(s2, p1, 200)$ 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 $\texttt{part}(\texttt{part\_no}, \texttt{part\_cost}, \texttt{contract})$ 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 inexpensivelym.

[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.

The ADMS Project: Views "Я" Us †

Nick Roussopoulos *Chungmin M. Chen* *Stephen Kelley*
Department of Computer Science
University of Maryland
College Park, MD 20742
{nick,min}@cs.umd.edu, skelley@umiacs.umd.edu

Alex Delis
Department of Information Systems
Queensland University of Technology
Brisbane, Australia
ad@icis.qut.edu.au

Yannis Papakonstantinou
Stanford University
Department of Computer Science
Stanford, CA 94305
yannis@DB.Stanford.EDU

Abstract

The goal of the ADMS project is to create a framework for caching materialized views, access paths, and experience obtained during query execution. The rationale behind this project is to amortize database access cost over an extended time period and adapt execution strategies based on experience. ADMS demonstrates the versatility of the views and their role in performance, data warehousing, management and control of data distribution and replication.

1 Introduction and Motivation for ADMS

The main goal of the *Adaptive Database Management System (ADMS)* project is to capitalize on the reuse of retrieved data and data paths in order reduce execution time of follow-up queries. From its inception, ADMS is trying to satisfy a need that has been neglected, that is, learning to perform better with experience obtained by query execution. Although database management systems use sophisticated query optimization techniques and access methods, they neither gather nor retain any experience from query execution and/or obtained results. For example, the cost of an execution plan generated by the optimizer is not compared to the actual cost incurred during the execution in order to adapt strategies during follow-up plans. Similarly, data retrieved during query execution has a very short life span in the buffer area of the database systems and is not retained on disk for future use. ADMS's goals include capturing knowledge that affects query optimization, such as attribute value distributions, selectivities, and page faults, and creating a framework for caching materialized views, access paths, and amortizing their maintenance cost.

Query execution cost can be reduced by reusing intermediate and/or final query results cached in *Materialized View Fragments (MVFs)*. These MVFs can then be accessed efficiently at only a fraction

†This research was partially sponsored by the National Science Foundation under grants IRI-9057573 and GDR-85-00108, by NASA/USRA under contracts 5555-09 and NAG 5-2926, by ARPA under contract 003195, by the Institute of Systems Research, and by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

of the cost of their initial generation. Access paths can also be cached in the form of *ViewCache* pointers [Rou91] which are trail markers captured during the execution of queries. They point to base relation tuples and/or to lower level ViewCache entries satisfying a query and may be used by the system during subsequent queries to walk through the same data paths without search.

Subsumption is a common technique for discovering useful MVFs and ViewCaches which contain (subsume) the results of a given query. Such a facility is necessary in any large data warehouse in which the catalog cannot be manually browsed. Subsumption goes hand-in-hand with validation techniques such as cache coherence [FCL92] and relevant update [BLT86]. These are techniques for deciding whether or not cached data is affected by updates. However, since these techniques were developed for short-lived caches, they are of rather limited value for the long term disk caching of a warehouse. To remedy this, ADMS uses *incremental access methods* [RES93] for propagating updates to the MVFs and ViewCaches and, thus, prolongs their useful life span. The ADMS optimizer [CR94b] uses a subsumption algorithm for discovering applicable MVFs and ViewCaches, incremental access methods for updating MVFs and ViewCaches [Rou91], and an amortized cost model in evaluating alternative execution plans.

The adaptive functionality of ADMS captures not only access paths, intermediate and final results, but also the experience obtained during their use in query execution. The ADMS buffer manager observes page faults during execution of queries and builds a "page fault characteristic curve" which predicts page faults under different buffer availability situations [CR93]. These predictions are then used by the system for selecting global buffer allocation strategies. Similarly, ADMS exploits knowledge captured inside MVFs and ViewCaches. The selectivity of operators and cardinality of predicates are harvested during the creation, use, and update of ViewCaches and MVFs, and are fed back to an "adaptive curve-fitting" module which obtains accurate attribute value distributions with minimal overhead [CR94a].

This philosophy of caching query results has been extended in the $ADMS\pm$ Enhanced Client-Server database architecture [RK86b, RK86a, RD91, RES93, DR92, DR93]. MVFs are dynamically downloaded from multiple heterogeneous (commercial) server DBMSs to clients running ADMS–, a single user client version of ADMS. ADMS– creates its own data warehouse by caching downloaded results in replicas, incorporating them in locally processed queries, and making incremental update requests from the servers holding their primary copies. ADMS– allows the user to create composite views from multiple heterogeneous DBMSs and enables him to integrate them with local and proprietary data. An extension of the ADMS subsumption algorithm is to find a "best fit" set of MVFs residing on multiple clients for answering a given query. In this technique, the cost is affected by the number of fragments used and the negation predicates which preclude duplicates from the final result [Pap94].

In this paper, we outline the motivation, rationale, concepts, techniques, and the implementation of the University of Maryland ADMS prototype. Section 2 of this paper describes the view engine of ADMS. Section 3 outlines the ADMS optimizer. Section 4 describes the distributed $ADMS\pm$ Client-Server architecture and the management of replicas. Section 5 contains concluding remarks, a brief historical account of ADMS, and current developments.

2 The ADMS View Engine

ADMS uses traditional relational storage organization and standard access methods including sequential scan, B-trees, R-trees and hashing. The ADMS catalogs store names, locations, cardinalities, selectivities, etc. for all databases connected to the warehouse, relations in them, attributes, and indexes. They are the core resource for ADMS query validation and optimization, and are maintained in base relations so they can be queried through ADMS's standard SQL interface. The two novel

extensions of the core ADMS system are its ViewCache storage organization and its adaptive buffer manager.

2.1 The ViewCache Storage and the Incremental Access Method

The ViewCache [Rou91] is the most innovative feature and has the most far reaching consequences of any caching technique in ADMS. It is a persistent storage structure consisting of pointers to data objects which are either base relation tuples or pointers in lower level ViewCaches.

The ViewCache storage structure and its incremental access methods are built around a single relational operator we call *SelJoin*. *SelJoin* corresponds to a join with concurrent selections on its arguments. The outer relation can be empty in which case a *SelJoin* reduces to a simple selection. Three join methods have been implemented for *SelJoin*, nested loop, index, and hash join. Each ViewCache corresponds directly to a single application of *SelJoin* and is, therefore, a 1- or 2-dimensional array of pointers (TIDs) depending on the number of arguments of *SelJoin*. The TIDs point to records of the underlying relations or other views necessary to make up the result of the *SelJoin*. A multiway join view is then expressed as a tree of *SelJoin* ViewCaches with the base relations at the leaves. Every ViewCache in the hierarchy is maintained for the life of the view and the expression which defines each subtree is inserted into the catalogs so that ViewCache fragments can be reused, possibly in other views. Since ViewCaches contain only TIDs, they are relatively small in size and can be constructed, quickly materialized (dereferenced), and actively maintained with little system overhead.

All other relational operators such as projection, duplicate elimination, aggregate, ordering, and set theoretic operators are performed on the fly during output, using masking, hashing, and main memory pointer manipulation routines. The advantages of having a single underlying operator are manyfold: Firstly, the optimizer does not have to consider pushing selections or projections ahead of joins or vice versa. Secondly, subsumption on *SelJoin* ViewCaches is more general when no attributes have been projected out. Lastly, the incremental algorithms for *SelJoin* ViewCaches are simple and require no heavy bookkeeping as opposed to incremental update algorithms for projection views or views containing other aggregate operators which have significant complexity and require sophisticated bookkeeping and expensive logging.

ViewCaches are maintained in ADMS by its *Incremental Access Method (IAM)* which amortizes their creation and update costs over a long period of time (indefinitely). IAM maintains update logs which permit either eager or deferred (periodic, on-demand, event-driven) update strategies. The on-demand strategy allows a ViewCache to remain outdated until a query needs to selectively or exhaustively materialize the underlying view. The IAM is designed to take advantage of the ViewCache storage organization, a variation of packed R-trees [RL85]. This organization attempts to reduce the number of intermediate node groupings in the R-tree. This number is the most significant parameter in determining the cost of materializing the ViewCaches. Both ViewCache incremental update and tuple materialization (dereferencing) from it (ViewCache) are interleaved using one–pass algorithms. The interleaved mode avoids the duplication of retrieving the modified records to be updated and then materialized again for the remaining of the query processing.

Compared to the query modification technique for supporting views that requires re-execution of the definition of the view, and to the incremental algorithms for MVFs [TB88], IAM on ViewCaches offers significant performance advantages, in some cases up to an order of magnitude. The decision about whether or not an IAM on a ViewCache is cost-effective, i.e. less expensive than re-execution, depends on the size of the differentials of the update logs between consecutive accesses of the ViewCaches. For frequently accessed views and for base relations which are not intensively updated, IAM by far outperforms query modification [RES93]. Performance gains are higher for multilevel ViewCaches because all the I/O and CPU for handling intermediate results is replaced with efficient pointer manipulation.

2.2 The ADMS Adaptive Buffer Manager

ADMS uses an adaptive allocation scheme to allocate buffers from the global buffer pool to concurrent queries. Page reference behavior of ViewCache materialization and recurring queries involving MVFs are quantified using page fault statistics obtained during executions [CR93]. This page fault information is fed back to the buffer manager and gets associated with each MVF and/or ViewCache. An "adaptive curve-fitting" module is used to capture the *Marginal Gain Ratio (MGR)* on page faults, i.e. the faults reduction per additional buffer allocated to a query using a ViewCache. ADMS's buffer manager basically identifies two important characteristics, the "critical size", that is the number of buffers beyond which the reduction of faults starts diminishing, and the "saturation size", the number of buffers beyond which no reduction is attained. As queries utilizing MVFs and ViewCaches recur, the buffer manager observes, adapts, and saves their characteristics to continuously capture the effects on page faulting as the database changes in time.

ADMS allocates buffers to these queries according to their page fault characteristics and the global buffer availability. Buffers are allocated to individual queries/relations in proportion to their average Marginal Gain Ratios (MGR) subject to the following constraints: (a) never allocate more than the saturated size (avoid waste), and (b) when the demand for buffers is high, never exceed the critical size of each reference string.

Experimental results in ADMS validated the advantage of MGR over traditional methods such as global LRU and DBMIN [CD85]. Through a comprehensive set of ADMS experiments, we demonstrated that MGR offers significant performance improvement over a pattern prediction-based algorithm [NFS91] and a load control-based algorithm [FNS91]. In all cases of query mixing and under various degree of data sharing, on the average, MGR outperforms the second best strategy by $15\% - 30\%$ in query throughput.

The merit of the MGR allocation scheme can be attributed to the feedback of faulting characteristics, which provides more insightful buffer utilization information than the probabilistic analysis-based methods which rely on the infamous crude assumption of uniformity.

3 The ADMS Query Optimizer

The ADMS query optimizer invokes a subsumption matching algorithm to identify relevant ViewCaches and generates alternative query plans utilizing those matched ViewCaches. It then selects between incremental or re-execution update strategies depending on their corresponding projected costs. Another key feature of the ADMS query optimizer is its adaptive selectivity estimation mechanism. This mechanism provides cost-effective and accurate selectivity estimation using *query feedback*. In essence, this feedback is information contained in the MVFs and ViewCaches constructed or updated during prior query processing. Minimal (CPU only) overhead is incurred to compute and adaptively maintain selectivity statistics. With this approach, ADMS completely avoids the overhead of the traditional off-line access of the database for gathering value distribution statistics. The first subsection describes the ADMS query optimizer; the second subsection describes the technique of adaptive selectivity estimation.

3.1 Subsumption Driven Optimization

The query optimizer uses a dynamic programming graph search algorithm for obtaining an efficient query execution plan. A query graph can be reduced to another one generating the same final result either by a *SelJoin-reduction* or by a *match-reduction*. A *SelJoin-reduction* corresponds to the execution of a *SelJoin* for evaluating a fragment of the query graph; a *match-reduction* corresponds to answering

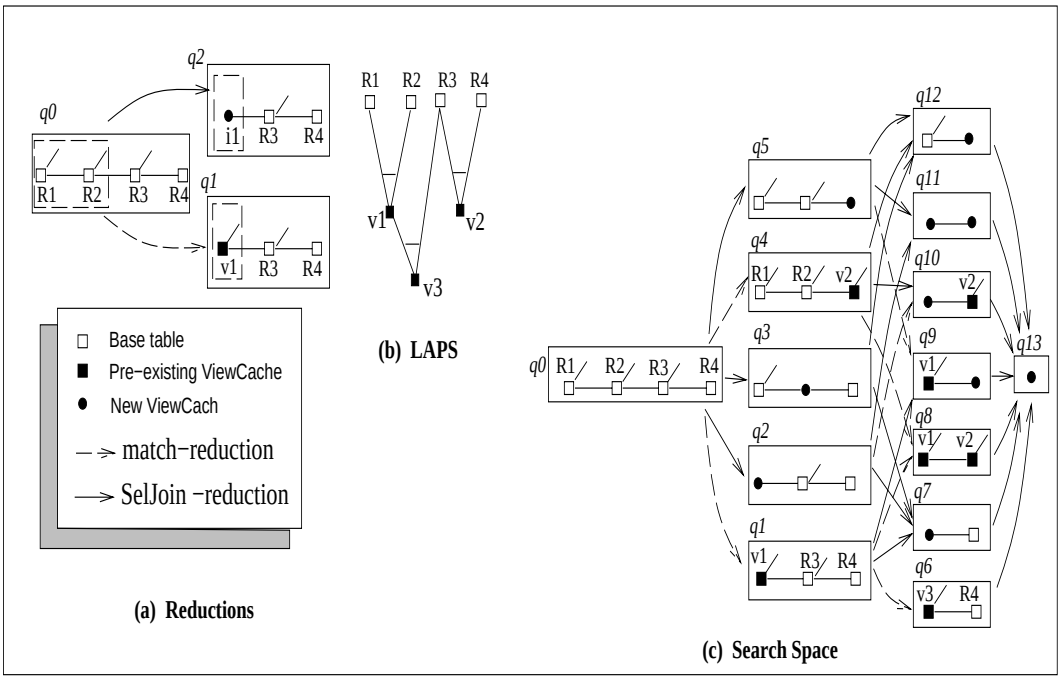


Figure 1: LAPS, Query Graph Reductions, Subsumption Reductions Rules and Search Space

the query graph via a matched, pre-existing ViewCache that subsumes the reduced fragment. Figure 1.a illustrates the two different kinds of reductions that can be applied to a query graph. The Logical Access Path Schema (LAPS) [Rou82] is used to organize the ViewCaches in a directed acyclic graph data structure. This is built from the database catalogs containing the definition of ViewCaches along with their derivation paths. Figure 1.b shows a LAPS with four base relations and three ViewCaches. The search of the query optimizer traverses top-down the LAPS structure and applies *match-reductions* in a breadth-first manner.

Subsumption is used during the *match-reduction* for deciding whether or not a more general ViewCache V "logically implies" a query fragment q. This is in general an undecidable problem, but in the context of database queries it is an NP-hard problem [RH80]. Although several sophisticated algorithms have been proposed [LY85, S$^+$89], we adopted a rather simple subsumption algorithm which extends the "direct elimination" technique proposed in [Fin82] and has worst case complexity of $O(mn)$, where m and n are the number of predicates in V and q respectively. The subsumption algorithm is *sound*, in the sense that it answers positively only when the implication statement is valid, but *not complete*, in the sense that it may not discover all possible subsumable clauses.

For each reduction, the cost of performing the $SelJoin$ from scratch or accessing the matched ViewCache is estimated and accumulated in the reduced query graph. If the ViewCache is outdated, the cost includes both alternatives, the projected cost of an incremental update and the projected cost of a re-execution of the ViewCache. Thus, starting from the initial query graph, the search algorithm generates successive query graphs until a single node graph is obtained which represents the query's result. The path with the lowest cost is then selected for execution. Figure 1.c shows the search space for the query and LAPS given in figure 1.a and b.

The performance of the ADMS query optimizer was evaluated by running a comprehensive set of experiments using a benchmark database and synthetic queries. By turning off the subsumption algorithm and the incremental access methods, the ADMS optimizer reduces to the System R [SAC$^+$79], thus allowing us to make comparisons. The experiments showed that ViewCaches with subsumption

and dynamic selection between incremental update and re-execution of MVFs and ViewCaches save substantial query execution time, and thus, increase the overall query throughput under a variety of query and update loads [CR94b]. The improvement ranged between 30% - 60% in query throughput under moderate update loads while it suffered no loss under heavy update loads. Although query optimization cost is increased by up to one tenth of a second per query, this overhead is insignificant when compared to query execution reduction of seconds or tens of seconds.

3.2 Adaptive Selectivity Estimation

The most significant factor in evaluating the cost of each query execution plan is *selectivity* – the number of tuples in the result of a relational selection or join operator. Various methods based on maintaining attribute value distributions [Chr83, PSC84, MD88, SLRD93, Ioa93] and query sampling [HOT88, LN90, HS92] have been proposed to facilitate selectivity estimation.

ADMS uses and adaptively maintains approximating functions for value distributions of attributes used in query predicates. We implemented both polynomials and splines[1] and an "adaptive curve-fitting" module which feeds back accurate selectivity information after queries and updates [CR94a]. The CPU overhead of adapting the coefficients of the approximating functions is hardly noticeable but the estimation accuracy of this approach is comparable to traditional methods based on periodic off-line statistics gathering or sampling. However, unlike these methods, the query feedback of ADMS incurs no I/O overhead and, since it continuously adapts, it accurately reflects changes of the value distributions caused by updates in the database.

In all our experiments, the adaptive selectivity estimating functions converge very closely to the actual distribution after 5 to 10 query feedbacks of random selection ranges on the attribute. For a rather staggered actual distribution, it takes almost the same time to converge to a stable curve. In such a case, the resultant curve may not fit seamlessly to the actual distribution due to the smoothness nature of the polynomials, but it represents the optimal approximation to the actual distribution in the sense of least square errors.

4 ADMS±: An Enhanced Client-Server Database Architecture

Commercial Client-Server DBMS architectures, which exemplify primary copy distributed database management, have significant performance and scalability limitations. Firstly, record-at-a-time navigation through their interfaces is way too slow to be functional. Secondly, a dynamic SQL interface is rather restrictive, simply because no optimized plans can be submitted; the user is forced to use the server's optimization services as they are. Finally, these architectures do not scale up because all database accesses (I/O) and processing are done on the server.

The *ADMS*± architecture, [RK86b, RK86a], preceded all known database client-server architectures and introduced the concept of a client DBMS that cooperates with the servers not only for query processing, but also for data accessing from replicated MVFs dynamically downloaded onto their disks. Figure 2 shows the *ADMS*± architecture with two commercial database servers and three clients. The ADMS– client has all the capabilities of ADMS, but is usually run in single user mode (indicated by the – sign). The ADMS+ component on the figure is the gateway layer that runs as an application on top of the underlying DBMS.

A user on an ADMS– client can connect to a number of commercial DBMS servers and make queries against their databases. Query results are downloaded and incrementally maintained as MVFs on the

[1]Splines are piecewise polynomials and give the database administrator the flexibility of choosing parameters which best fit the application, such as the degree and the number of polynomial pieces.

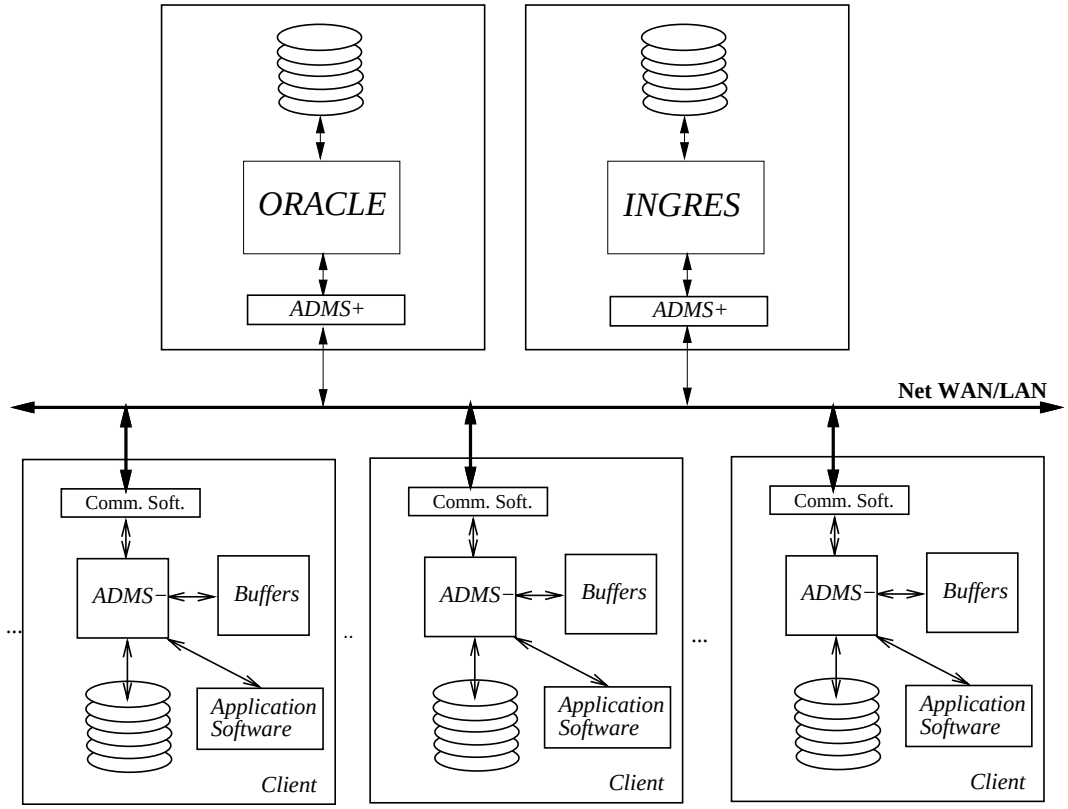


Figure 2: The $ADMS\pm$ Client-Server Architecture

client ADMS–. Again subsumption of views and incremental access methods provide the foundations for efficient and controlled management of the data replication. Since ADMS– views can be glued from multiple global heterogeneous sources, i.e. server-server joins, combined with possibly proprietary data on the client, i.e. server-client joins, $ADMS\pm$ became the first conceived *Data Warehousing Architecture* and is operational since 1989 [Eco89, RES93].

Updates from a client are sent to the primary copy on the server(s) and find their way to the downloaded MVFs through incremental update algorithms by transmitting the log differences [RES93]. $ADMS\pm$ supports eager and a number of deferred update propagation strategies including on-demand, periodic, and event driven. The deferred strategies allow $ADMS\pm$ to operate under "weak connectivity" mode in which clients can be disconnected and reconnected at later time. Again, MVFs and their incremental update algorithms provide the foundations of this architecture.

The value of the $ADMS\pm$ architecture is three-fold. First, it distributes asynchronous and parallel I/O to the clients, alleviating the I/O bottleneck on the servers [DR92]. This is a significant performance booster because of the exhibited scalability of the architecture. Second, it provides a controlled mechanism for managing replication and update propagation [DR94]. Third, by caching MVFs on the clients it permits client mobility and database access in disconnected mode, whereby a client uses its local disk when some of the database server(s) are unaccessible.

4.1 Performance of $ADMS\pm$

We extensively studied the $ADMS\pm$ Enhanced Client–Server architecture, [RD91, DR92, DR93] and showed that it (a) outperforms all other client-server architectures including those with equal number

of disks replicating their data, (b) scales up very nicely and reaches linear scalability on read- and append-mostly databases. The studies showed that the distribution of the I/O to the client disks and the parallelism obtained this way are the main contributors to the performance and scalability.

In [DR94], we proposed a number of update propagation techniques for the $ADMS\pm$ architecture. We then studied performance under various workloads and scalability as the number of participating clients increases. We showed that, under high server resource utilization, a simple broadcasting strategy for server updates gives better performance than any other update propagation policy. However, when none of the server resources reach full utilization, on-demand update propagation strategy furnishes better results.

4.2 Utilization of MVF's on the Clients

In the distributed environment of $ADMS\pm$, query requests from a client may be answered by MVFs that reside on other clients instead of retrieving from the server(s). This is very important for performance reasons but even more so for fault tolerance when the connections between some clients and the servers are unavailable.

We have extended the ADMS subsumption algorithm to discover *unions* of MVFs that subsume a client's query posed on the network as a *range* query [Pap94]. A query may be subsumed by the union of a number of MVFs residing on different clients even when none of the fragments does. Duplicates resulting from the union are filtered out by constructing additional range predicates.

Union subsumption can be used in several ways. First, we can improve performance of a query by replacing expensive join operators with simpler selections. Second, given that the MVFs may be located at client workstations we can reduce the contention for the server resources. Third, we can use union subsumption to parallelize query processing with fragments residing on different workstations. Note, even if we are not able to subsume the client query, we may be able to subsume some part of the data that are necessary for the computation of the client query; i.e., we may be able to subsume some nodes of the query graph by unions of MVFs.

Assuming that the MVFs are not indexed, the optimizer attempts to minimize the total size of the MVFs – equivalently, the total retrieval time – and also attempts to distribute the work evenly to all clients. However, the decision of an optimal set that subsumes Q is an NP-hard problem. Thus, we use a greedy polynomial best-fit optimization algorithm that selects at every step the "most promising" MVF. In addition, the selected set must not have more than a few hundred MVFs, because otherwise, the cost of applying the filters becomes greater than the cost of retrieving the MVF from the disk.

5 Conclusions

This article presented the motivation, concepts, ideas, and techniques of the ADMS Project. Many of the ideas pioneered in this project are finding their way into the commercial world. For example, Oracle is now offering limited incremental refresh of select only MVFs. Also, both Sybase and Ingres use incremental techniques for maintaining replicated data. Similarly, the recent flurry of research activity on views and their management is gratifying. We are content that our long-term conviction and persistence on view maintenance and replication have paid off, and that our ideas are finally receiving appropriate attention.

The ADMS design document was written in 1984, and implementation began on a SUN 3 workstation at that time. It has gone though several major revisions, such as when the ViewCache storage organization was evolving 1985, 1986, 1987, and when the SQL parser and cost-based optimizer were added in 1991. It has now migrated and been ported to SUN SPARC, DEC MIPS, HP SNAKE, and

IBM POWER architectures running their various flavors of UNIX. The unified source tree consists of approximately 120,000 lines of "C" code.

The *ADMS±* client-server architecture was designed during the fall of 1985 but the first implementation of the prototype began only in late 1987. *ADMS±* is now in its third incarnation which includes new client-server and server-server join strategies, enhanced server catalogs (for selectivity estimation), and a robust TCP/IP based communication layer. Last year, we ran our first trans-atlantic joins between an Oracle database at the University of Maryland and an Ingres database at the National Technical University in Athens, Greece.

And the ADMS saga goes on. We are targeting our energy towards adaptive and intelligent techniques capable of learning from running queries against the database and fine tune their processing. We have developed a query optimizer for the ADMS– client. It has an adaptive cost estimator which exhibits excellent learning capability over foreign commercial DBMSs. An experiment is being conducted as of this writing and we will report the results in the near future.

References

[BLT86] J.A. Blakeley, P.A. Larson, and F.W. Tompa. Efficiently Updating Materialized Views. In *Proc. of the 1986 ACM SIGMOD Intern. Conference*, pages 61–71, August 1986.

[CD85] H. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Procs. of the 11th Intl. Conf. on VLDB*, pages 127–141, 1985.

[Chr83] S. Christodoulakis. Estimating Record Selectivities. *Inf. Syst.*, 8(2):105–115, 1983.

[CR93] C.M. Chen and N. Roussopoulos. Adaptive Database Buffer Allocation Using Query Feedback. In *Procs. of the 19th Intl. Conf. on Very Large Data Bases*, 1993.

[CR94a] C.M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[CR94b] C.M. Chen and N. Roussopoulos. The implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Procs. of the 4th Intl. Conf. on Extending Database Technology*, 1994.

[DR92] A. Delis and N. Roussopoulos. Performance and Scalability of Client–Server Database Architectures. In *Proc. of the 19th Int. Conference on Very Large Databases*, Vancouver, BC, Canada, August 1992.

[DR93] A. Delis and N. Roussopoulos. Performance Comparison of Three Modern DBMS Architectures. *IEEE–Transactions on Software Engineering*, 19(2):120–138, February 1993.

[DR94] A. Delis and N. Roussopoulos. Management of Updates in the Enhanced Client–Server DBMS. In *Proccedings of the 14th IEEE Int. Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.

[Eco89] N. Economou. Multisite Database Access in *ADMS±*. Master's thesis, University of Maryland, College Park, MD, 1989. Department of Computer Science.

[FCL92] M. Franklin, M. Carey, and M. Livny. Local Global Memory Management in Client–Server DBMS Architectures. In *Proc. of the 18th Int. Conference on Very Large Data Bases*, Vancouver, Canada, August 1992.

[Fin82] S. Finkelstein. Common Expression Analysis in Database Applications. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 235–245, 1982.

[FNS91] C. Faloutsos, R. T. Ng, and T. Sellis. Predictive Load Control for Flexible Buffer Allocation. In *Procs. of the 17th Intl. Conf. on VLDB*, pages 265–274, 1991.

[HOT88] W. Hou, G. Ozsoyoglu, and B. K. Taneja. Statistical Estimators for Relational Algebra Expressions. In *Procs. of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 276–287, 1988.

[HS92] P. Haas and A. Swami. Sequential Sampling Procedures for Query Size Estimation. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 341–350, San Diego, CA, 1992.

[Ioa93] Y.E. Ioannidis. Universality of Serial Histograms. In *Procs. of the 19th Intl. Conf. on VLDB*, Dublin, Ireland, 1993.

[LN90] R. J. Lipton and J. F. Naughton. Practical Selectivity Estimation through Adaptive Sampling. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 1–11, Atlantic City, NJ, 1990.

[LY85] P.-Å. Larson and H. Z. Yang. Computing Queries from Derived Relations. In *Procs. of the 11th Intl. Conf. on VLDB*, pages 259–269, 1985.

[MD88] M. Muralikrishma and D. DeWitt. Equi-depth Histograms for Estimating Selectivity Factors for Multi-dimensional Queries. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 28–36, Chicago, Illinois, 1988.

[NFS91] R. T. Ng, C. Faloutsos, and T. Sellis. Flexible Buffer Allocation Based on Marginal Gains. In *Procs. of 1991 ACM SIGMOD Intl. Conf. on Management of Data*, pages 387–396, 1991.

[Pap94] Y. Papakonstantinou. Computing a Query as a Union of Disjoint Horizontal Fragments. Technical report, Department of Computer Science, University of Maryland, College Park, MD, 1994. Working Paper.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 256–275, Boston, MA, 1984.

[RD91] N. Roussopoulos and A. Delis. Modern Client–Server DBMS Architectures. *ACM–SIGMOD Record*, 20(3):52–61, September 1991.

[RES93] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A Testbed for Incremental Access Methods. *IEEE Trans. on Knowledge and Data Engineering*, 5(5):762–774, 1993.

[RH80] D.J. Rosenkrantz and H.B. Hunt. Processing Conjunctive Predicates and Queries. In *Procs. of the 6th Intl Conf. on VLDB*, 1980.

[RK86a] N. Roussopoulos and H. Kang. Principles and Techniques in the Design of *ADMS±*. *Computer*, December 1986.

[RK86b] N. Roussopoulos and Y. Kang. Preliminary Design of ADMS±: A Workstation–Mainf rame Integrated Architecture. In *Proc. of the 12th Int. Conference on Very Large Databases*, August 1986.

[RL85] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-trees. In *Procs. of 1985 ACM SIGMOD Intl. Conf. on Management of Data*, Austin, 1985.

[Rou82] N. Roussopoulos. The Logical Access Path Schema of a Database. *IEEE Trans. on Software Engineering*, SE-8(6):563–573, 1982.

[Rou91] N. Roussopoulos. The Incremental Access Method of View Cache: Concept, Algorithms, and Cost Analysis. *ACM–Transactions on Database Systems*, 16(3):535–563, September 1991.

[S+89] X. Sun et al. Solving Implication Problems in Database Applications. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 185–192, 1989.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Data Base System. In *SIGMOD–Conference on the Management of Data*, pages 22–34. ACM, June 1979.

[SLRD93] W. Sun, Y. Ling, N. Rishe, and Y. Deng. An Instant and Accurate Size Estimation Method for Joins and Selection in a Retrieval-Intensive Environment. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 79–88, Washington, DC, 1993.

[TB88] F. Tompa and J. Blakeley. Maintaining Materialized Views Without Accessing Base Data. *Information Systems*, 13(4):393–406, 1988.

Data Integration and Warehousing Using H2O [†]

Gang Zhou[‡] Richard Hull Roger King Jean-Claude Franchitti

Computer Science Department, University of Colorado
Boulder, CO 80309-0430
email: {gzhou, hull, king, franchit}@cs.colorado.edu

Abstract

This paper presents a broad framework for data integration, that supports both data materialization and virtual view capabilities, and that can be used with legacy as well as modern database systems. The framework is based on "integration mediators", these are software components that use techniques generalized from active databases, such as triggering and rulebases. This permits the logic, especially for incremental maintenance of materialized data, of an integration mediator to be specified in a relatively declarative and modular fashion.

One specific focus of this paper is the development of a taxonomy of the solution space for supporting and maintaining integrated views. A second focus concerns providing support for intricate object matching criteria that specify when object representations (e.g., OIDs) in different databases correspond to the same object-in-the-world (or interchangeable ones).

1 Introduction

One of the most important computer science problems today is to develop flexible mechanisms for effectively integrating information from heterogeneous and geographically distributed information sources, one or more of which may be legacy systems. Among a wide range of techniques addressing this problem, *data warehousing*, i.e., materializing integrated information in a persistent store, is gaining increasing importance [WHW90, KAAK93, IK93, ZGHW95]. This paper describes how we are applying research being developed in the H2O project at the University of Colorado, Boulder, to support data integration. We describe here a broad framework for data integration, that supports both data materialization and virtual view capabilities, and that can be used with legacy as well as modern database systems.

The main focus of this paper is on (i) a taxonomy of the solution space for the problem of data integration, (ii) the concept of "active modules", these are software components that support the specification of behavior using rules as in active databases, and (iii) the Squirrel prototype for constructing "integration mediators", in the sense of [Wie92]. These integration mediators are active modules that support data integration using a hybrid of virtual and materialized data approaches.

Modern integration applications involve a number of issues, including different kinds of data and data repositories, available resources at the site of the mediator (e.g., storage capacity), requirements on the integrated view (e.g., query response time and up-to-dateness). As a result, no single approach to supporting data integration can be universally applied. To better understand the impact of those

[†]This research was supported in part by NSF grant IRI-931832, and ARPA grants BAA-92-1092 and 33825-RT-AAS.
[‡]A student at the University of Southern California

```
interface Student {
    string      studName;
    integer[7]  studID;     // key
    string      major;
    string      local_address;
    string      permanent_address; };
```

Subschema of *StudentDB* database

```
interface Employee {
    string     empName;
    integer[9] SSN;     // key
    string     divName;
    string     address; };
```

Subschema of *EmployeeDB* database

Figure 1: Subschemas of `StudentDB` and `EmployeeDB` in ODL syntax

issues on data integration, we have developed a taxonomy of the solution space. We consider several spectra in the solution space, including for example a spectrum about the degree of materialization, which ranges from fully materialized to fully virtual.

A fundamental aspect of data warehousing is propagation of incremental updates at the source databases to the warehouse. Activeness, as found in active databases [WC95], is emerging as the paradigm of choice for supporting such propagation. Part of the H2O project is focused on the notion of "active module" [Dal95, BDD+95]. These are software modules that include a rule base, an execution model for rule application, and optionally a local persistent store. While an active module might be a full-fledged active database, it might also be a lightweight process supporting a focused family of functionalities. Intuitively speaking, the concept of active module allows the separation of activeness from active databases. An active module permits separate specification of the logic (in the rule base) and the control (in the execution model) of an application. A substantial benefit of the active module concept is that it allows the development of integrated mediators in a modular fashion, thus facilitating convenience, maintainability, and reusability.

Finally, this paper presents a generator for integration mediators, called *Squirrel*. The framework for generating integration mediators is currently focused primarily on four issues:

- managing fully materialized, fully virtual, or hybrid integrated information in the mediator;
- supporting a variety of incremental update mechanisms for materialized/cached data, which can work with legacy as well as state-of-the-art (active) DBMSs;
- developing a high level integration specification language (ISL), so that integration mediators can be automatically generated based on ISL specifications of data integration applications; and
- providing support for intricate object matching criteria that specify when object representations in different databases refer to the same object-in-the-world (or interchangeable ones). The traditional virtual approach to data integration typically uses universal keys to perform object matching, and cannot efficiently support more intricate object matching criteria.

This paper is a survey of some aspects of the on-going research in the H2O project at the University of Colorado, Boulder, and the presentation here is somewhat abbreviated. More details can be found in [BDD+95, ZHKF95, DHDD95, DHR95]. The rest of the paper is organized as follows: Section 2 gives a motivating example that illustrates our approach. Section 3 presents the taxonomy of the space of approaches to data integration. The notion of active modules is described in Section 4. Section 5 gives a high level description of the Squirrel framework, including a description of ISL and how integration mediators are generated from ISL specifications. The conclusion of the paper is given in Section 6.

2 An Example Problem and Its Solution

This section gives an informal overview, based on a simple example, of our framework for data integration using an integration mediator. An important component of the mediator is a data warehouse that

```
DEFINE VIEW Student_Employee
SELECT s.studName, s.major, e.divName
FROM s IN StudentDB:Student, e IN EmployeeDB:Employee
WHERE match(s, e);
```

Figure 2: Definition of view `Student_Employee` expressed in extended OQL

holds a materialized portion of the integrated view and also other data from the source databases that is needed for incrementally maintaining the integrated view. More details concerning this development may be found in [ZHKF95].

In the example we assume that there are two databases, `StudentDB` and `EmployeeDB`, that hold information about students at a university and employees in a large nearby corporation, respectively. The relevant subschemas of the two databases are shown in Figure 1. An integration mediator, called here `S_E_Mediator`, will maintain an integrated view about persons who are both students and employees, providing their names, majors, and names of the divisions where they work. The definition of the view is given in Figure 2, where `match` is a predicate derived from a user-given matching criteria that is used to determine whether two objects match (i.e., they refer to the same person in the real world).

A host of issues are raised when attempting to perform this kind of data integration in practical contexts. The primary emphasis of our current research is on three fundamental issues:

(a) mechanisms to maintain replicated (possibly derived) data and how to automatically generate rules for a given data integration application,

(b) mechanisms for the construction of integrated views, perhaps involving both materialized and virtual data, and

(c) mechanisms to support rich object matching criteria.

In this section we consider each of these issues with respect to the example problem in reverse order.

With regards to issue (c), we assume in the example that a student object s *matches* an employee object e if (1) either s.`local_address` = e.`address` or s.`permanent_address` = e.`address`, and (2) their names are "close" to each other according to some metric, for instance, where different conventions about middle names and nick names might be permitted. The "closeness" of names is determined by a user-defined function, called here `close_names()`, that takes two names as arguments and returns a boolean value. It should be noted that no universal key is assumed for the class of student employees. As a result, under the traditional approach based on a virtual integrated view and query shipping, the potentially expensive step of identifying matching pairs of objects may be incurred with each query against the view. However, if the update-to-query ratio is sufficiently small, then it will be much more efficient to materialize the match information and update it incrementally.

To support the object matching criteria between students and employees, we propose that the data warehouse of `S_E_Mediator` holds three classes: a class called `Stud_match_Emp` and two *auxiliary* classes that are used to facilitate incremental updates of the objects in `Stud_match_Emp` class. Speaking intuitively, `Stud_match_Emp` will hold pairs of matching `Student` and `Employee` objects. For this example, the two auxiliary classes are `Stud_minus_Emp` and `Emp_minus_Stud`. `Stud_minus_Emp` will hold one object for each student in `Student` who is not an employee; and analogously for `Emp_minus_Stud`.

Figure 3 shows the interfaces of the three classes in more detail. Here the `Stud_minus_Emp` and `Emp_minus_Stud` classes include the attributes needed to perform matches. The `Stud_match_Emp` class holds all of the attributes from both `Stud_minus_Emp` and `Emp_minus_Stud`. Attributes corresponding to `s.major` and `e.divName` in the export class `Student_Employee` are not present in `Stud_match_Emp`; in this example they are supported as virtual attributes.

```
interface Stud_minus_Emp {
    string      studName;
    integer[7]  studID;
    string      local_address;
    string      permanent_address; };

interface Emp_minus_Stud {
    string      empName;
    integer[9]  SSN;
    string      address; };
```

```
interface Stud_match_Emp {
    string      studName;
    integer[7]  studID;
    string      local_address;
    string      permanent_address;
    string      empName;
    integer[9]  SSN;
    string      address; };
```

Figure 3: Class interfaces of S_E_Mediator

```
R1: on message m from StudentDB
    if create Student( x: sn, sid, maj, ladd, padd)
    then [create Stud_minus_Emp(new: x.sn,x.sid,x.ladd,x.padd); pop message m];

R2: on create Stud_minus_Emp( x: sn, sid, ladd, padd)
    if (exists Emp_minus_Stud(y: en, ssn, addr) and close_names(x.sn, y.en)
        and (x.ladd = y.addr or x.padd = y.addr))
    then [delete Stud_minus_Emp(x); delete Emp_minus_Stud(y);
          create Stud_match_Emp(new: x.sn,x.sid,x.ladd,x.padd,y.en,y.ssn,y.addr)];
```

Figure 4: Sample rules for maintaining local store of the integration mediator

We now illustrate (b) the issue of constructing integrated views. As just noted, in the example solution `s.studName` is materialized in `S_E_Mediator`, while `s.major` and `e.divName` are virtual. Queries against the view would be broken into three pieces, one each for `StudentDB`, for `EmployeeDB`, and for the `Stud_match_Emp` class. Suppose now that the cost of query shipping to `EmployeeDB` is considered to be very high. Then we can adopt a solution involving more materialization as follows: an attribute `divName` (division name) is included in both the `Stud_match_Emp` and `Emp_minus_Stud` classes, and is maintained by `S_E_Mediator` in a materialized fashion. In this case, the integration mediator does not need to ship subqueries to `EmployeeDB`. More generally, given a rich integrated view, some portions can be supported using materialization and others using the virtual approach. Further, the choice of what is materialized or virtual can be changed dynamically.

Finally, we turn to issue (a), that of incrementally maintaining materialized data in the integration mediator. Two basic issues arise: (i) importing information from the source databases and (ii) correctly maintaining the materialized data to reflect changes to the source databases. In connection with the example, with regards to (i) we assume that both source databases can actively send messages containing the net effects of updates (i.e., insertions, deletions, and modifications) to `S_E_Mediator`. (Other possibilities are considered in Section 3.) A rule base can be developed to perform (ii). Two representative rules responding to the creation of new `Student` objects in the source database `StudentDB`, written in a pidgin H2O [BDD+95, DHR95] rule language, are shown in Figure 4. Intuitively, the two rules state:

Rule `R1`: If an object of class `Student` is created, create a new object of class `Stud_minus_Emp`.

Rule `R2`: Upon the creation of a `Stud_minus_Emp` object `x`, if there is a corresponding object `y` of class `Emp_minus_Stud` that matches `x`, then delete `x` and `y`, and create a `Stud_match_Emp` object that represents the matching pair.

The complete rule base would include rules dealing with creation, deletion, and modification of objects in both source databases, and are generated automatically (see Section 5).

No.	Spectra	Range
1	Materialization	fully materialized $\longleftrightarrow$ hybrid $\longleftrightarrow$ fully virtual
2	Activeness of Source DB	sufficient activeness $\longleftrightarrow$ restricted activeness $\longleftrightarrow$ no activeness
3	Maintenance Strategies	incremental update $\longleftrightarrow$ refresh
4	Maintenance Timing	trans. commit, net change, network reconnect, periodic, ...

Table 1: Solution space for data integration problems

3 A Taxonomy of the Solution Space for Data Integration

The solution presented in the previous section for the Student/Employee example represents just one point in the space of possible approaches to solving data integration problems using integration mediators and data warehousing. This section identifies the major spectra of this solution space. More details concerning this taxonomy may be found in [ZHKF95].

Our taxonomy is based on four spectra (Table 1). The first spectrum is relevant to all solutions for data integration, and the latter three are relevant to solutions that involve data warehousing. We feel that these spectra cover the most important design choices that must be addressed when solving a data integration problem. (Some other dimensions include kinds of matching criteria used, languages used for view definition, and support for security.) Within each spectra we have identified what we believe to be the most important points, relative to the kinds of data integration problems and environments that arise in practice. While the spectra are not completely orthogonal, each is focused on a distinct aspect of the problem.

The taxonomy is useful in creating modular implementations of integration mediators. For example, the taxonomy suggests that the implementation of incremental update can be independent from the choice and implementation of maintenance timing. Such modularity facilitates the reusability and maintainability of the rulebases in integration mediators.

Before discussing the spectra individually, we indicate where the solution of the previous section fits in the taxonomy. The solution assumed partially materialized and partially virtual (hybrid) approach (Spectrum 1) and that the source databases were sufficiently active (Spectrum 2). The Maintenance Strategy (Spectrum 3) used was incremental update, and Maintenance Timing (Spectrum 4) was event triggering by the source databases.

3.1 Materialization

This spectrum concerns the approach taken by an integration mediator for physically storing the data held in its integrated view. The choices include

fully materialized approach, as presented in references [WHW89, KAAK93], which materializes all relevant information in the data warehouse of the mediator;

hybrid approach, as illustrated in the Student/Employee example of Section 2, that materializes only part of the relevant information; and

fully virtual approach, as presented in [DH84, ACHK93, FRV95], that uses query pre-processing and query shipping to answer queries that are made against the integrated view.

The fully virtual approach saves storage space and offers acceptable response time for queries if the computation of the view definition is not expensive and network delay is minimal, e.g. the source databases and the mediator are located on a local area network. In many other cases, the fully materialized approach is much more effective.

A compromise between the fully virtual and fully materialized approaches is the hybrid approach, which materializes only the data that is most critical to the response time and leaves other data virtual to conserve storage space of the mediator, as illustrated in Section 2.

It is well-known that data from multiple information sources cannot be accessed at precisely the same time. As a result, the answer to a query against an integrated view may not correspond to the state of the world at any given time, and inconsistent data may be retrieved. This problem is exacerbated if some of the data is replicated, as is the case in our approach. The Eager Compensation Algorithm developed in [ZGHW95] provides a promising direction for solving at least some of these inconsistency problems. We plan to address this issue in our future research.

3.2 Activeness of Source Databases

This spectrum concerns the active capabilities of source databases, and is relevant only if some materialization occurs. This spectrum allows for both new and legacy databases. The three most important points along this spectrum represent three levels of activeness.

Sufficient activeness: A source database has this property if it is able to periodically send deltas corresponding to the net effect of all updates since the previous transmission, with triggering based either on physical events or state changes. This provides two major advantages: First, it could significantly reduce the network traffic by transferring deltas rather than full snapshots of the membership of a class. Second, most algorithms [BLT86, GMS93] for maintaining materialized views compute the incremental updates on derived data based on the net effects of the updates of the source data.

Restricted activeness: A source database has this property if it cannot send deltas, but it has triggering based on some physical events (e.g., method executions or transaction commits), and the ability to send (possibly very simple) messages to the integration mediator. Perhaps the most useful possibility here is the case that on a physical event the source database can execute a query and send the results to the integration mediator. Even if the source database can send only more limited messages, such as method calls (with their parameters) that were executed, then the mediator may be still able to interpret this information (assuming that encapsulation can be violated).

No activeness: This is the case where a source database has no triggering capabilities. In this case the mediator can periodically poll the source databases and perform partial or complete refreshes of the replicated information.

3.3 Maintenance Strategies

Maintenance strategies are meaningful only if some materialization occurs in the mediator. We consider two alternative maintenance strategies:

incremental update of the out-of-date objects, and

refresh of the out-of-date classes in the mediator by re-generating all their objects.

In general, refreshing is more straightforward in terms of implementation and can be applied under all circumstances. In contrast, incremental updating is generally more efficient, especially when the updates in the source databases affect only a small portion of the objects, as is true in most cases.

3.4 Maintenance Timing

Maintenance timing concerns when the maintenance process is initiated. Many different kinds of events can be used to trigger the maintenance. Some typical kinds of events include: (i) a transaction commits in a source database, (ii) a query is posed against out-of-date objects in the mediator, (iii)

the net change to a source database exceeds a certain threshold, for instance, 5% of the source data, (iv) the mediator explicitly requests update propagation, (v) the computer holding the mediator is reconnected via a network to the source databases, and (vi) a fixed period of time has passed.

With the periodic approach, the user can balance the tradeoff between out-of-date data and maintenance costs, by setting the appropriate length of maintenance cycles.

4 Heraclitus and Active Modules

This section briefly describes the concept of active module and the underlying Heraclitus paradigm, which forms one of the roots of the H2O project.

In its broadest sense, an *active module* [Dal95, BDD+95] is a software module that incorporates:

- a rule base, that specifies the bulk of the behavior of the module in a relatively declarative fashion;
- an execution model for applying the rules (in the spirit of active databases);
- (optionally) a local persistent store[1]

An active module can be viewed as capturing some of the spirit and functionality of active databases, without necessarily being tied to a DBMS. In particular, the separation of rules (logic/policy) from execution model (implementation/mechanism) allows a more declarative style of program specification, and facilitates maintenance of the active module as the underlying environment evolves. Reference [Dal95] describes an implemented prototype system that uses several active modules with different execution models to support complex interoperation of software and database systems.

A key enabling technology in the development of active modules has been the Heraclitus paradigm [HJ91, GHJ+93, GHJ94]. As detailed in those citations, the Heraclitus paradigm permits the flexible specification of a wide range of execution models based on deferred rule firing, immediate rule firing, and hybrids of these, and also supports rich expressive power in rule conditions. More concretely, it elevates deltas, i.e., collections of updates proposed to the current database state, to be first-class citizens in a database programming language. Deltas can be used to easily represent different virtual states that are created during the course of rule application, and make these accessible to rule conditions.

The Heraclitus[Alg,C] DBPL [GHJ+93, GHJ94] implements the Heraclitus paradigm for the relational data model; a central component of the H2O project is the development of the H2O DBPL [DHDD95, DHR95], an extension and generalization of Heraclitus[Alg,C] for object-oriented databases.[2] The current experimentation with the framework described in this paper is based on Heraclitus[Alg,C], and we expect the port to the H2O DBPL to be relatively straightforward.

As noted in the Introduction, our approach to data integration and warehousing is based on integration mediators, which are a special kind of active module. This allows us to approach the data integration problem using a framework that supports the declarative specification of behavior, thus facilitating maintainability and reusability.

5 The Squirrel Prototype

Section 3 presented a broad taxonomy of the solution space for supporting data integration using mediators. We are currently developing a general tool, called Squirrel, for generating integration mediators that can accommodate many of the points in that solution space. This section describes some of the key components of this tool.

[1] Actually, part or all of the store could be remote, but for the present paper we focus on the case where it is local.

[2] Indeed, 'H2O' is an abbreviation for Heraclitus[OO].

```
Source-DB: StudentDB
    interface Student {
        string     studName;
        ... ...
    };
    key: studName;

Source-DB: EmployeeDB
    interface Employee {
        string     empName;
        ... ...
    };
    key: empName;
```

```
Correspondence #1:
    Match criteria:
        close_names(studName, empName)
        AND (address = local_address
             OR address = permanent_address)
    Match object files:
        $home/demo/close_names.o

Export classes:
    DEFINE VIEW Student_Employee
    SELECT s.studName, s.major, e.divName
    FROM s IN StudentDB:Student,
         e IN EmployeeDB:Employee,
    WHERE match(s, e);
```

Figure 5: An example of ISL specification

We are currently focused on supporting hybrid materialized/virtual integrated views, but otherwise on the left-most positions in Table 1, i.e., on sufficiently active source databases, incremental update, and maintenance timing based on event triggering from the source databases. In this presentation we focus on the case where information from two databases is to be integrated; the generalization to more than two databases is a subject of future research.

Squirrel will be used to construct integration mediators for specific integration applications. Users can invoke Squirrel by specifying an integration problem using a high level Integration Specification Language (ISL) (Subsection 5.1). Based on this, Squirrel generates a corresponding integration mediator (Subsection 5.2). The presentation here is rather abbreviated, more details are presented in [ZHKF95].

5.1 Integration Specification Language (ISL)

The Integration Specification Language (ISL) allows users to specify their data integration applications in a largely declarative fashion. The primary focus of ISL to be discussed here is on the specification of integrated views and matching criteria. (Issues such as composing heterogeneous applications, as handled by, e.g., Amalgame Specification Language (ASL) [FK93], are not addressed here.) In the current version of ISL, users can specify (1) (relevant portions of) source database schemas, (2) the criteria to be used when matching objects from corresponding pairs of classes in the source databases, and (3) derived classes to be exported from the integration mediator. Item (2) may include conditions based on, among other things, boolean relations or user-defined functions (that may in turn refer to "look-up tables" or intricate heuristics). (This part of ISL is optional; it is not needed for data integration applications that do not involve object matching.) The export classes of item (3) are specified using (extended) OQL queries, and may refer to both the source databases and the match classes of the correspondence specifications. If an attribute name is unique within the two classes, it can be used without the class name attached to it, otherwise it takes the form of `class_name.attr_name`. Although not illustrated here, the keyword `virtual` can be used in ISL specifications of export classes, to indicate that selected attributes or full classes are to be supported in a virtual manner. The ISL specification of the Student/Employee example is shown in Figure 5.

5.2 Generating Integration Mediators

In the Squirrel framework, integration mediators share the same basic architecture, which includes communication facilities with queues for incoming messages, a query processor, a local store, a rule

base, and an execution model for rule application. The communication facilities and query processor are also common to all Squirrel mediators. We currently use a fixed execution model which uses deferred (transaction boundary) rule-firing and accumulates the effect of rule actions using a natural composition operator; we are also experimenting with other execution models.

When given an ISL specification, the Squirrel prototype generates an integration mediator based on the common architecture and features just mentioned. The following components may vary with each application: (a) the schema definition for the local store, (b) the rulebase, and (c) initialization for the source databases. In what follows we focus primarily on the generation of components (a) and (b), i.e., the schema for the local store and the rule base. Assuming that the source databases are sufficiently active, (c) has the form of rules for the source databases that propagate relevant updates to the integration mediator.

With regards to the schema for the local store (a), the most novel aspect of Squirrel concerns the maintenance of the materialized information for object matching. We assume here full materialization of the match classes. In order to support the match classes and the materialized attributes of export classes we use three kinds of attributes (these sets may overlap):

identification attributes: These are used to identify objects from the source databases. They might be keys or immutable OIDs (cf. [EK91]) from the source databases.

match attributes: These are the attributes referred to in the match criteria. For example, the match attributes of the class `Employee` are `empName` and `address`.

export attributes: These are attributes that are used in the export classes.

As illustrated in Section 2, we use three classes in the local store to maintain match information for a pair `R` and `S` of corresponding classes from the two databases:

Auxiliary class I: `R_minus_S` with attribute set equal to the union of at least the identifying and matching attributes of class `R`. The (remaining) export attributes may or may not be included, depending on if they should be materialized. Class `R_minus_S` holds one object for each object in `R` which does not correspond to any object of `S`.

Auxiliary class II: `S_minus_R`, analogous to auxiliary class I.

Match class: `R_match_S` with attribute set the union of those of the two auxiliary classes. Class `R_match_S` contains an object `m` for each pair (`r`, `s`) of objects from `R` and `S`, respectively, that correspond according to the matching conditions.

Note that this framework can also be used to access the sets corresponding intuitively to $\mathtt{R} - \mathtt{S}$ and $\mathtt{S} - \mathtt{R}$, and also to the the "union" or "outer-join" of `R` and `S`.

Export classes are defined as views constructed from the source databases and from the match classes. The current prototype can support export classes that are defined using what amounts to conjunctive queries [AHV95]; these correspond to relational algebra queries that can be expressed using selection, projection, and join. Given a view defined by such a query and information about what parts of it should be materialized, it is relatively straightforward to determine what (projections of) source database classes should be materialized. The export class itself might also be materialized, or might be left as virtual. Generalizing this to export classes defined using richer queries is a topic of current research.

The basic approach to generating rules for integration mediators follows the general spirit of [CW91, Cha94], which describe how view definitions can be translated into rules for performing incremental maintenance. The primary mechanism used by Squirrel is a family of *rule templates*. These are used to generate the specific rules of an integration mediator, based on the object matching and materialization needed for export classes. To illustrate, we consider how object matching is supported. Templates are inluded for handling creates, deletes and modifies arising from either source database, and for propagating these through the auxiliary and match classes. A representative rule template is:

```
on create R_minus_S(x: r_a_1,...,r_a_v)
if (exists S_minus_R(y:s_a_1,...,s_a_w) && x match y)
then [delete R_minus_S(x); delete S_minus_R(y); create R_match_S(new: m_1,...,m_u)];
```

This template has the following effect: on the event that a new `R_minus_S` object `x` is created, if an object `y` of class `S_minus_R` matches `x`, then delete `x` and `y` and create a `R_match_S` object.

Translation of the templates into actual rules uses information about the source database classes, the auxiliary and match classes of the integration mediator, and possibly user-defined functions. In the running example, the above rule template would generate rule `R2`. Although the rules generated from the templates refer to individual objects, the execution model we currently use applies the rules in a set-at-a-time fashion.

6 Conclusions and Current Status

This paper presents a broad framework for using integration mediators based on active modules to integrate and warehouse information from heterogeneous data sources. It makes five specific contributions towards database interoperation. To provide context for the research presented here, we (a) present a taxonomy of the solution space for supporting and maintaining integrated views, with an emphasis on situations where part or all of the integrated view is materialized. At a more concrete level, our solution (b) uses integration mediators as a special class of active modules to provide declarative and modular specification of integration and incremental update propagation. Furthermore, (c) our framework can work with legacy as well as state-of-the-art DBMSs. We also (d) develop a preliminary version of a high-level Integration Specification Language (ISL), along with a description of how to translate ISL specifications into integration mediators. Finally, (e) our framework provides additional support for intricate object matching criteria.

We are developing the Squirrel prototype for generating integration mediators. These are implemented in the Heraclitus[Alg,C] DBPL, but as the H2O DBPL becomes available we shall port our prototypes to H2O. For communication between source databases and the integration mediators we are using Knowledge Query and Manipulation Language (KQML) [FWW+93].

In the near future, we plan to extend this research primarily in the direction of the hybrid materialized/virtual approach. In one experiment we plan to use the SIMS [ACHK93] query processing engine to execute queries where matching information is materialized but all other export data is virtual. We also plan to incorporate mechanisms for integrating data that involves related but "non-congruent" classes, in the spirit of [Cha94, CH95].

Acknowledgements

We are grateful to Omar Boucelma, Ti-Pin Chang, Jim Dalrymple, and Mike Doherty for many interesting discussions on topics related to this research. We also thank Jennifer Widom for her careful review of this paper and valuable comments.

References

[AHV95] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, MA, 1995.

[ACHK93] Y. Arens, C.Y. Chee, C.N. Hsu, C.A. Knoblock. Retrieving and integrating data from multiple information sources. *Intl. Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[BDD+95] O. Boulcema, J. Dalrymple, M. Doherty, J-C. Franchitti, R. Hull, R. King, and G. Zhou. Incorporating active and multi-database-state services into an OSA-compliant interoperability toolkit. *The Collected Arcadia Papers, Second Edition*. University of California at Irvine, 1995.

[BLT86] J.A. Blakeley, P.-A. Larson, F.W. Tompa. Efficiently updating materialized views. *Proc. ACM SIGMOD Symp. on the Management of Data*, 61–71, 1986.

[Cha94] T.-P. Chang. *On Incremental Update Propagation Between Object-Based Databases*. PhD thesis, University of Southern California, Los Angeles, CA, 1994.

[CH95] T.-P. Chang and R. Hull. On Witnesses and Witness Generators for Object-Based Databases. *Proc. of the ACM Symp. on Principles of Database Systems*, 196–207, 1995.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. *Proc. of Intl. Conf. on Very Large Data Bases*, 577–589, 1991.

[Dal95] J. Dalrymple. *Extending Rule Mechanisms for the Construction of Interoperable Systems*. PhD thesis, University of Colorado, Boulder, 1995.

[DH84] U. Dayal and H.Y. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Trans. on Software Engineering*, SE-10(6):628–644, 1984.

[DHDD95] M. Doherty, R. Hull, M. Derr, J. Durand. On detecting conflict between proposed updates. To appear, *Proc. Intl. Workshop on Database Programming Languages*, Italy, September, 1995.

[DHR95] M. Doherty, R. Hull, M. Rupawalla. The Heraclitus[OO] database programming language, 1995. Technical Report in preparation.

[EK91] F. Eliassen and R. Karlsen. Interoperability and Object Identity. *SIGMOD Record* 10(4):25–29, 1991.

[FK93] J. C. Franchitti and R. King. A Language for Composing Heterogeneous, Persistent Applications. *Proc. of the Workshop on Interoperability of Database Systems and Database Applications*, Fribourg, Switzerland, October 13-14 1993. Springer-Verlag, LNCS.

[FRV95] D. Florescu, L. Raschid, P. Valduriez. Using heterogeneous equivalences for query rewriting in multidatabase systems. *Proc. of Third Intl. Conf. on Cooperative Information Systems (CoopIS-95)*, Vienna, Austria, May 1995.

[FWW+93] T. Finin, J. Weber, G. Wiederhold, et al. DRAFT Specification of the KQML Agent-Communication Language. June 15, 1993.

[GHJ+93] S. Ghandeharizadeh, R. Hull, D. Jacobs, et. al. On implementing a language for specifying active database execution models. *Proc. of Intl. Conf. on Very Large Data Bases*, 441–454, 1993.

[GHJ94] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus[Alg,C]: Elevating deltas to be first-class citizens in a database programming language. Technical Report USC-CS-94-581, Computer Science Department, Univ. of Southern California, 1994.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. *Proc. ACM SIGMOD Symp. on the Management of Data*, 157–166, 1993.

[HJ91] R. Hull and D. Jacobs. Language constructs for programming active databases. *Proc. of Intl. Conf. on Very Large Data Bases*, 455–468, 1991.

[IK93] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse*. QED Publishing Group, Boston, Massachussetts, 1993.

[KAAK93] W. Kent, R. Ahmed, J. Albert, and M. Ketabchi. Object identification in multidatabase systems. D. Hsiao, E. Neuhold, and R. Sacks-Davis, editors, *Interoperable Database Systems (DS-5) (A-25)*. Elsevier Science Publishers B. V. (North-Holland), 1993.

[WC95] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, Inc., San Francisco, California, 1995.

[WHW89] S. Widjojo, R. Hull, and D. Wile. Distributed Information Sharing using WorldBase. *IEEE Office Knowledge Engineering*, 3(2):17–26, August 1989.

[WHW90] S. Widjojo, R. Hull, and D. S. Wile. A specificational approach to merging persistent object bases. Al Dearle, Gail Shaw, and Stanley Zdonik, editors, *Implementing Persistent Object Bases*. Morgan Kaufmann, December 1990.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 38–49, March 1992.

[ZGHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom. View maintenance in a warehousing environment. *Proc. ACM SIGMOD Symp. on the Management of Data*, San Jose, California, May 1995.

[ZHKF95] G. Zhou, R. Hull, R. King, J-C. Franchitti. Using object matching and materialization to integrate heterogeneous databases. *Proc. of Third Intl. Conf. on Cooperative Information Systems (CoopIS-95)*, Vienna, Austria, May 1995.

The Stanford Data Warehousing Project

Joachim Hammer, Hector Garcia-Molina, Jennifer Widom, Wilburt Labio, and Yue Zhuge
Computer Science Department
Stanford University
Stanford, CA 94305
E-mail: joachim@cs.stanford.edu

Abstract

The goal of the data warehousing project at Stanford (the WHIPS project) is to develop algorithms and tools for the efficient collection and integration of information from heterogeneous and autonomous sources, including legacy sources. In this paper we give a brief overview of the WHIPS project, and we describe some of the research problems being addressed in the initial phase of the project.

1 Introduction

A *data warehouse* is a repository of integrated information, available for querying and analysis [13, 14]. As relevant information becomes available or is modified, the information is extracted from its source, translated into a common model (e.g., the relational model), and integrated with existing data at the warehouse. At the warehouse, queries can be answered and data analysis can be performed quickly and efficiently since the information is directly available, with model and semantic differences already resolved. Furthermore, warehouse data can be accessed without tying up the information sources (e.g., holding locks or slowing down processing), accessing data at the warehouse does not incur costs that may be associated with accessing data at the information sources, and warehouse data is available even when the original information source(s) are inaccessible.

The key idea behind the data warehousing approach is to extract, filter, and integrate relevant information in *advance* of queries. When a user query arrives, the query does not have to be translated and shipped to the original sources for execution (as would be done in, e.g., a *mediated* approach to information integration [19].) Not only can such translation and shipping be a complex operation, but it also can be time consuming, especially if the sources are many and remote. Thus, warehousing may be considered an "active" or "eager" approach to information integration, as compared to the more traditional "passive" approaches where processing and integration starts when a query arrives. Warehousing also simplifies metadata management, and it provides a trusted and long-term repository for critical data that is under the control of the end-user.

One potential drawback of the warehousing approach is that data is physically copied from the original sources, thus consuming extra storage space. However, given dropping storage prices and the fact that data can be filtered or summarized before it is warehoused, we do not believe this is a serious problem. A more significant problem is that copying data introduces potential inconsistencies with the sources—warehouse data may become out of date. Another potential drawback is that the "warehouse administrator" must specify in advance what sites data should be extracted from, and

which data should be copied and integrated. For these reasons, the data warehousing approach may not be appropriate when absolutely current data is required, or when clients have unpredictable needs.

In reality, we believe that data warehousing should be seen as a complement, not a replacement, to passive query processing schemes, or to ad-hoc exploration and discovery mechanisms [2, 11]. For example, ad-hoc discovery mechanisms can identify information of interest, which can can be collected at the warehouse, improving and simplifying access to the information.

The following examples suggest some of the application areas or characteristics for which the warehousing approach to information integration seems well suited.

1. *Collecting scientific data.* In these applications, large amounts of heterogeneous data are created so rapidly that real-time query processing becomes impossible. Furthermore, the sources may be sporadic and unreliable, so that warehousing the data in a safe and convenient location for later processing is appropriate.
2. *Maintaining historical enterprise data.* Processing and mining enterprise data (e.g., computing the sales history of all stores of a large supermarket chain over a certain period of time) is a resource-intensive job that is better performed off-line so as to not affect the day-to-day operations of the business. Thus it is desirable to move historic enterprise data away from the mainstream transaction processing systems where the data is created into an enterprise-owned data warehouse.
3. *Caching frequently requested information.* By storing in the data warehouse previously fetched and integrated answers to frequently asked queries, the inherent disadvantages of federated database systems (e.g., inefficiency, delay in query processing, etc.) can be overcome, resulting in improved performance and efficiency.

In this paper we present a brief overview of the data warehousing project we are undertaking at Stanford, called WHIPS (for "WareHouse Information Project at Stanford"). The goal of the WHIPS project is to develop algorithms and tools for the efficient collection and integration of information from heterogeneous and autonomous sources. Because the project is quite new—it began officially in January 1995—our focus in this paper is on the overall architecture of the system (Section 3), and on some of the specific research issues that we have investigated to date (Section 4).

2 Related Research

An introduction to data and knowledge warehousing is presented in reference [15]. In this book, the advantages of the warehousing approach are put forth and a general architecture and suggested functionality are presented. There has been a significant amount of research in the database and knowledge-base community related to the problem of integrating heterogeneous database and knowledge bases; representative literature is [4, 6, 10, 16, 17, 18, 19, 20]. Much of this work uses the "passive" approach discussed in the introduction. Some of the approaches rely on modifying the individual databases to conform to a "global schema." Despite these differences with our project, there are still a number of similarities, e.g., translating heterogeneous data into a common model, merging data from multiple sources, propagating data from source to target databases, etc. Hence, we are adapting methods from the heterogeneous database literature to the mediation and integration aspects of our warehousing approach.

If one considers the data residing in the warehouse as a *materialized view* over the data in the individual information sources, then it may be possible to adapt previously devised algorithms for *view maintenance* to the problem of change propagation and warehouse updating in the data warehousing environment. Two important differences with the traditional view maintenance problem are (1) the heterogeneity of the sources, and (2) the autonomy of the sources. The second problem implies that

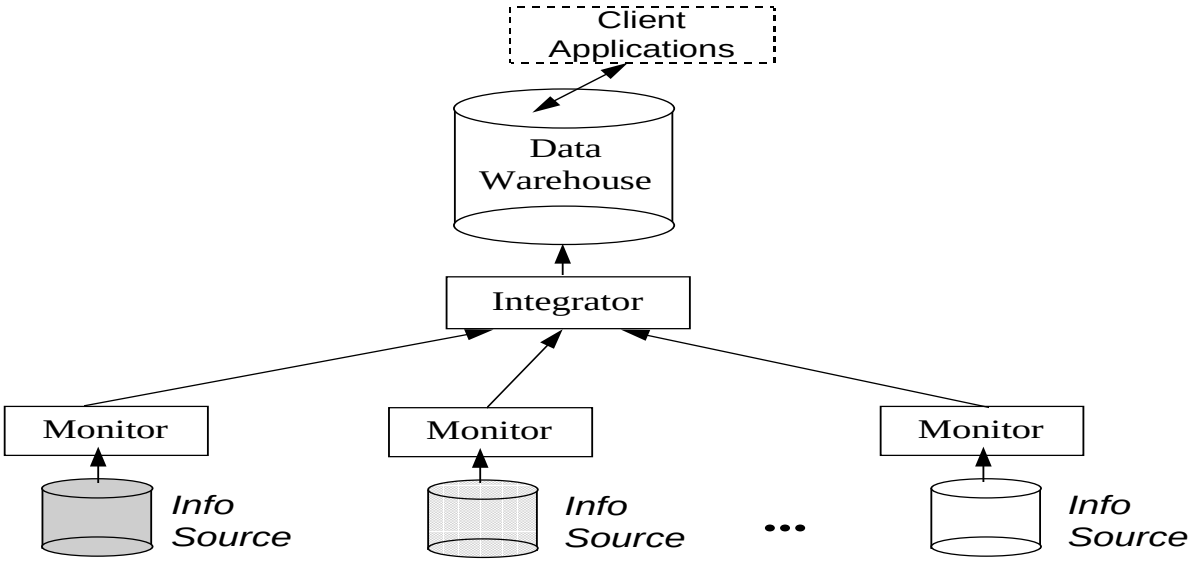


Figure 1: Architecture

sources may not fully cooperate in view management, which leads to problems discussed in Section 4.3 below. Previously devised algorithms for view maintenance can be found in [1, 3, 5], and we have built upon these in our solutions.

The problem of monitoring individual databases to detect relevant changes is central to the area of *active databases*, as represented in, e.g., [7, 9, 12]. We are exploiting appropriate parts of this technology in the WHIPS project.

3 Architecture

Figure 1 illustrates the basic architecture of our system. The bottom of the figure depicts the multiple *information sources* of interest. These sources could include imagery, video, and text data, along with more traditional data sources such as relational tables, object-oriented structures, or files. Data that is of interest to the client(s) is copied and integrated into the *data warehouse*, depicted near the top of the figure.[1] Between the sources and the warehouse lie the (*source*) *monitors* and the *integrator*. The monitors are responsible for automatically detecting changes in the source data. Whenever the source content changes, the new or updated information that is relevant to the warehouse is propagated to the integrator. More details on how monitors operate and how they can be implemented are provided in Section 4.1.

The integrator is responsible for bringing source data into the warehouse. Integrating heterogeneous data into an existing schema is a difficult problem requiring several steps. First, the data must be made to conform to the conceptual schema used by the warehouse. Then the data must be merged with existing data already present, in the process resolving inconsistencies that might exist between the source and warehouse data. In essence, the integrator is where the differences across information sources are specified, where relationships among data at multiple sources are defined, where duplicates and inconsistencies are detected, and where it is determined how information will be integrated into the warehouse. More details on our approach to integration are provided in Section 4.2.

[1] If feasible, the client(s) may initially choose to replicate *all* of the data in order to get an overview of what is available before making a choice.

At the top of the figure are the *client applications* that let users interact with the warehouse. Currently we plan to provide only the basic query capabilities offered by any "off-the-shelf" database system, but additional capabilities (e.g., decision support, data mining, etc.) can be added at a later point if necessary.

A number of features of our planned architecture are not depicted explicitly in Figure 1. For example, although in the figure it appears that information flow is exclusively "upwards" (from information sources through monitors to the integrator and finally to the warehouse), in reality it often becomes necessary for the integrator to submit queries to one or more information sources in order to correctly integrate new or updated information (see Sections 4.2 and 4.3). Also note that the figure depicts *run-time* aspects of the system only. It is our ultimate goal to provide, in addition, a powerful *compile-time* component. Using this component, the desired contents of the warehouse are specified by the warehouse administrator. From this specification, appropriate monitors and an appropriate integrator are generated in an automatic or semi-automatic fashion, leading to easy and fast system configuration and reconfiguration for a variety of client needs.

4 Currently Under Investigation

There are numerous challenges associated with realizing the architecture in Figure 1. In this section we briefly summarize the main issues we have been addressing in the initial phase of the project.

4.1 Monitors

Monitors detect changes to an information source that are of interest to the warehouse and propagate the changes in a generic format to the integrator. If the source is a full-functionality database system, it may be possible for the monitor to simply define an appropriate set of triggers and wait to be notified of changes. Another option may be for the monitor to examine the update log file and extract the changes of interest. In other cases, such as legacy systems, the monitoring task may be much harder. Typically, legacy systems do not have trigger or logging capabilities (or, if such capabilities are present, they are not available to an external component like the monitor). In these cases, there are two options:

- Every application that changes the source data is modified so it emits appropriate notification messages to the monitor. For example, the application program that creates a new patient record for a hospital database will send a message to the monitor giving the new patient's name, status, and so on. Clearly, application level notifications are not particularly desirable, but they may need to be used when the underlying storage system will not perform notifications itself.
- A utility program is written that periodically dumps the source data into a file, and the monitor compares successive versions of the file. Many legacy systems already have dump utilities, typically used for backup or copying of data. Although the dump file format is usually specific to the legacy system, it often is possible to describe its "schema" in a generic way. We refer to the problem of detecting modifications by comparing successive dumps as the *snapshot differential* problem. Again, this solution is not particularly desirable, but it may often need to be used in practice.

We are currently focusing on solutions to the snapshot differential problem. One simple version of this problem is stated as follows: Each snapshot is a semistructured file F of the form $\{R_1, R_2, \ldots, R_n\}$, where R_i denotes a row or record in the file. Each row is (logically) of the form $\langle K, B \rangle$, where K is a key value and B is an arbitrary value representing all non-key data in the record. Given two files F_1 (the old file) and F_2 (the new file), produce a stream of *change notifications*. Each change notification is of one of the following forms:

1. *updated*(K_i,B_1,B_2), indicating that the row with key K_i has value B_1 in file F_1 and value B_2 in file F_2.

2. *deleted*(K_i,B_1), indicating that there exists a row with key K_i and value B_1 in file F_1, but no row with key K_i in file F_2.

3. *inserted*(K_i,B_2), indicating that there exists a row with key K_i and value B_2 in file F_2, but no row with key K_i in file F_1.

Note that rows with the same key may appear in different places in files F_1 and F_2. The snapshot differential problem is similar to the problem of computing *joins* in relational databases (*outerjoins*, specifically), because we are trying to match rows with the same key from different files. However, there are some important differences, among them:

- For snapshots, we may be able to tolerate some rows that are not "properly" matched. For example, suppose files F_1 and F_2 are identical except F_1 contains the row with key K_1 at the beginning, while F_2 contains K_1 towards the end. A simple "sliding window" snapshot algorithm might see K_1 in F_1 and try to find it in the first N (say) rows of F_2. Not seeing it there, it will report a deleted row with key K_1. Later, the algorithm will find K_1 in the second file and will report it as an inserted row. The superfluous delete and insert may create unnecessary work at the warehouse, but will not cause an inconsistency and therefore may be acceptable. In exchange, we may be able to construct a much more efficient snapshot differential algorithm, using an approach that cannot be used for relational (outer-)join.
- For the snapshot problem, we may be able to tolerate efficient probabilistic algorithms. For instance, suppose the B fields are very large. Instead of comparing the full B fields to discover differences, we may hash them into relatively small *signatures*. Consequently, there will be a small probability that we will declare rows $\langle K_i, B_1 \rangle$ and $\langle K_i, B_2 \rangle$ to be identical because the signatures matched, even though $B_1 \neq B_2$. However, a few hundred bits for the signature makes this probability insignificant, and this approach lets us dramatically reduce the size of the structures used in matching and differentiating rows.
- The snapshot problem is a "continuous" problem, where we have a sequence of files, each to be compared with its predecessor. Thus, in comparing F_2 to F_1, we can construct auxiliary structures for F_2 that will be useful when we later compare F_3 to F_2, and so on.

We are currently experimenting with several solutions to the snapshot differential problem. Some of the algorithms are similar to traditional join algorithms such as *sort-merge join* and *hash join*. Others are similar to UNIX *diff* and may not produce the minimal set of change notifications. The performance of these algorithms depends largely on the characteristics of the snapshots, e.g., whether the files are ordered or almost ordered, whether the keys are unique, whether the modifications are restricted to inserts only, etc. We have implemented an initial, simple differential monitor. It takes as input a schema specification for the files to be compared. It then analyzes the snapshot files using a sort-merge algorithm, sending change notifications to the integrator. Based on this framework, we plan to implement and evaluate several of the other schemes discussed above.

4.2 Integrator

The integrator component receives change notifications from the monitors and must integrate the changes into the warehouse. We are developing an approach where the integrator is implemented as a rule-based engine. Each rule is responsible for handling one kind of change notification, and is implemented as an object-oriented method. The method is called (or "triggered") whenever a monitor

generates a change notification of the appropriate type. The method body then performs the necessary processing to integrate the change into the warehouse. During this processing, the method may need to obtain additional data from the warehouse, or from the same or other sources. For example, suppose the warehouse keeps the average salary of employees at a company. When an update to an employee salary is reported, the update rule will have to obtain the current average from the warehouse in order to compute the new value. Scenarios where the integrator must obtain additional data from the sources are discussed in Section 4.3.

As discussed in Section 3, our ultimate goal is to provide a non-procedural, high-level specification language for describing how the relevant source data is integrated into the warehouse. Specifications in this language are compiled into the appropriate event processing code for the rule-based integration engine. This approach makes the integration process highly flexible and configurable, and allows the system to adapt easily to metadata changes at the sources or the warehouse.

For the initial implementation of our prototype, we are using the (CORBA compliant) Xerox PARC ILU distributed object system [8]. Using ILU allows the information sources, the monitors, the integrator, and the warehouse to run on different (distributed) machines and platforms while hiding low-level communication protocols, and it allows different components to be programmed using different languages.

4.3 Warehouse Update Anomalies

As mentioned in Section 2, in developing algorithms for integration one can think of a data warehouse as defining and storing a materialized view (or views) over the information sources. Numerous methods have been developed for maintaining materialized views in conventional database systems. Unfortunately, these methods cannot be applied directly in our warehousing environment. The key difference is that existing approaches assume that the system in which changes occur is the same system in which the view resides (or at least is a tightly-coupled related system). Consequently, when a change occurs, the system knows and has available any additional data needed for modifying the view.

In our warehousing environment, sources may be legacy or unsophisticated systems that do not understand views, and they are decoupled from the system where the view is stored. Sources may inform the integrator (through the monitor) when a change occurs, but they may not be able to determine or obtain additional data needed for incorporating the change into the warehouse. Hence, when a change notification arrives at the integrator, the integrator may discover that additional source data (from the same or different sources) is necessary to modify the view. When the integrator issues queries back to the sources, the queries are evaluated later than the corresponding changes, so the source states may have changed. This decoupling between the base data on the one hand (at the sources), and the view definition and view maintenance machinery on the other (at the integrator), can lead the to incorrect views at the warehouse. We refer to this problem as the *warehouse update anomaly* problem [21].

There are a number of mechanisms for avoiding warehousing update anomalies. As argued above, we are interested only in mechanisms where the source, which may be a legacy or unsophisticated system, does not perform any "view management." The source will only notify the integrator of relevant changes, and answer queries asked by the integrator. We also are not interested in, for example, solutions where the source must lock data while the warehouse view is modified, or in solutions where the source must maintain timestamps for its data. In the following potential solutions, view maintenance is autonomous from source changes.

Recompute the view. The integrator can either recompute the view whenever a change occurs at a source, or it can recompute the view periodically. Recomputing views is usually time and resource

consuming, particularly in a distributed environment where a large amount of data might need to be transferred from the source to the warehouse.

Store at the warehouse copies of all data involved in views. By keeping up-to-date copies of all relevant source data at the warehouse, queries can be evaluated locally at the warehouse and no anomalies arise. The drawbacks are wasted storage and overhead for keeping the copies current.

The Eager Compensating Algorithm. The solution we advocate avoids the overhead of recomputing the view or of storing copies of source data. The basic idea is to add to queries sent by the integrator to the sources *compensating queries* to offset the effect of concurrent updates. For details and examples of the Eager Compensating Algorithm, variations on the algorithm, and discussion of performance issues, refer to [21].

We plan to implement and experiment with the Eager Compensating Algorithm, along with the alternative approaches, within the rule-driven integrator framework described in Section 4.2.

5 Summary and Plans

Data warehousing is a valuable alternative to traditional "passive" approaches for integrating and accessing data from autonomous, heterogeneous information sources. The warehousing approach is particularly useful when high query performance is desired, or when information sources are expensive or transitory.

The initial WHIPS project testbed installed in our laboratory uses for its data warehouse a Sybase relational DBMS. We are currently using data from our University accounting system, which is supplied to us as large legacy snapshots. As described earlier, we are experimenting with simple monitors that compute changes between consecutive snapshots. We also are in the process of implementing the integrator component, using ILU objects as the communication infrastructure between the monitors and the integrator, and between the integrator and the warehouse. Since we have not yet developed our high-level integration description language, event processing code is currently "hard wired" into the integrator.

In addition to the work in progress described in Section 4, plans for the near (and not so near) future include:

- Migrate to a larger and more heterogeneous testbed application, most likely financial data from a variety of sources including subscription services, monthly transaction histories, World-Wide-Web sites, news postings, etc. The heterogeneity of the sources in this application will provide impetus for experimenting with a wide variety of monitor types and will provide additional challenges for the integrator.
- Extend our work on snapshot differential algorithms to handle dump files with nested-object structures in addition to flat record structures.
- Develop and implement algorithms that optimize the change propagation and integration process in our warehousing architecture. In particular, we would like to perform filtering of changes at the monitor level that is as sophisticated as possible, so that we can avoid overloading the integrator with change notifications that are discovered to be irrelevant to the warehouse. Conversely, we may find it useful to store certain additional data at the warehouse, so that we can eliminate the need to query the sources when a change notification occurs.
- Develop an appropriate warehouse specification language and techniques for compiling specifications into integration rules and appropriate monitoring procedures (recall Sections 3 and 4.2).

References

[1] S. Abiteboul and A. Bonner. Objects and views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 238–247, Denver, Colorado, May 1991.

[2] T. Addyman. WAIS: Strengths, weaknesses, and opportunities. In *Proceedings of Information Networking '93*, London, UK, May 1993.

[3] E. Bertino. A view mechanism for object-oriented databases. In *Advances in Database Technology—EDBT '92, Lecture Notes in Computer Science 580*, pages 136–151. Springer-Verlag, Berlin, March 1992.

[4] N. Cercone, M. Morgenstern, A. Sheth, and W. Litwin. Resolving semantic heterogeneity. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, California, February 1990.

[5] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991.

[6] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[7] S. Chakravarthy and D. Lomet, editors. *Special Issue on Active Databases,* IEEE Data Engineering Bullet in 15(4), December 1992.

[8] A. Courtney, W. Janssen, D. Severson, M. Spreitzer, and F. Wymore. Inter-language unification, release 1.5. Technical Report ISTL-CSA-94-01-01 (Xerox accession number P94-00058), Xerox PARC, Palo Alto, CA, May 1994.

[9] U. Dayal. Active database management systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pages 150–169, Jerusalem, Israel, June 1988.

[10] U. Dayal and H.-Y. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Transactions on Software Engineering*, 10(6):628–645, November 1984.

[11] J. Hammer, D. McLeod, and A. Si. Object discovery and unification in federated database systems. In *Proceedings of the Workshop on Interoperability of Database Systems and Database Applications*, pages 3–18, Swiss Information Society, Fribourg, Switzerland, October 1993.

[12] E.N. Hanson and J. Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(2):121–143, June 1993.

[13] W.H. Inmon. Building the data bridge: the ten critical success factors of building a data warehouse. *Database Programming & Design*, 1992.

[14] W.H. Inmon. EIS and the data warehouse: a simple approach to building an effective foundation for EIS. *Database Programming & Design*, 5(11):70–73, November 1992.

[15] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, Boston, Massachusetts, 1993.

[16] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

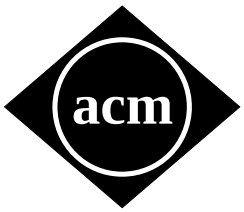
[17] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, Taipei, Taiwan, March 1995.

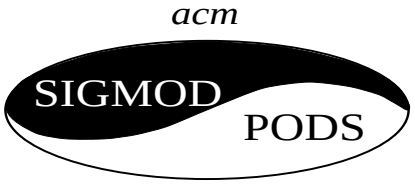
[18] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[19] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[20] G. Wiederhold. Intelligent integration of information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 434–437, Washington, DC, May 1993.

[21] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 1995.





Call for Papers
ACM SIGMOD/PODS 96 Joint Conference

Le Centre Sheraton, Montreal, Canada

June 3–6, 1996

The ACM SIGMOD/PODS '96 Joint Conference will, once again, bring together the SIGMOD and PODS conferences under one umbrella. The two events will overlap for two days to form a joint conference, but each will have a separate third day without overlap. There will be only one registration process for the joint, four-day conference. SIGMOD and PODS will have separate program committees and will have separate proceedings.

Authors must submit their papers to the conference that they feel is most appropriate for their work. We recommend that applied papers be submitted to SIGMOD and theoretical ones to PODS. The same paper or different versions of the same paper should not be submitted to both conferences simultaneously. Attendees will receive both proceedings and be encouraged to attend sessions in both conferences. Some of the technical events and the lunches will be joint events.

Submission Guidelines

All submissions are due on October 23, 1995. All submissions should be directed to the appropriate program chair:

H.V. Jagadish (Re: SIGMOD '96)
2T204, AT&T Bell Labs.
600 Mountain Avenue
Murray Hill, NJ 07974
USA
E-mail: `sigmod96@research.att.com`
Fax: (908) 582-5809

Richard Hull (Re: PODS '96)
Computer Science Department, ECOT 7-7
University of Colorado
Boulder, CO 80309-0430
USA
E-mail: `hull@cs.colorado.edu`
Fax: (303) 492-2844

The address, telephone number, FAX number, and electronic address of the contact author should be given on the title page of the submission. All authors of accepted papers will be expected to sign copyright release forms, and one author of each accepted paper will be expected to present the paper at the conference. Proceedings will be distributed at the conference, and will be subsequently available for purchase through the ACM.

For SIGMOD submissions: Please submit six copies of an 8000 word original manuscript to the program chair. In addition, submit by electronic mail, in clear text (no postscript, no Latex), an abstract of no more than 250 words, along with the title, authors, and the track (research or experience) for which the paper is being submitted. **The FIRM deadline for the submission of electronic abstracts is October 23, 1995.** The full paper should still be sent as hard copy. If you do not have access to electronic mail, please contact the program chair to make alternative arrangements for submission of the abstract.

For PODS submissions: Please submit twelve copies of a detailed abstract (not a complete paper) to the program chair by October 23, 1995. No papers received after October 30, 1995 will be considered. **This is a FIRM deadline.** A limit of 10 typed pages with roughly 35 lines/page (about 5000 words or 10K bytes in total) is placed on submissions. Font size should be at least 10 points.

For up-to-date information about the Joint Conference, consult the SIGMOD/PODS Web page that is set up: `URL=http://web.cs.ualberta.ca/~database/sp96/info.html`.

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903