Pergamon

Information Systems Vol. , No. , pp. –, 2001

ADAPTING MATERIALIZED VIEWS AFTER REDEFINITIONS: TECHNIQUES AND A PERFORMANCE STUDY

Ashish Gupta[1]*, Inderpal S. Mumick[2]†, Jun Rao[3]‡ and Kennth A. Ross[4]§

[1]Tavant Technology, ashishgupta@stanfordalumni.org
[2]Kirusa, mumick@mumick.com
[3]IBM Almaden Research Center, junrao@almaden.ibm.com
[4]Columbia University, kar@cs.columbia.edu

We consider a variant of the view maintenance problem: How does one keep a materialized view up-to-date when the view definition itself changes? Can one do better than recomputing the view from the base relations? Traditional view maintenance tries to maintain the materialized view in response to modifications to the base relations; we try to "adapt" the view in response to changes in the view definition.

Such techniques are needed for applications where the user can change queries dynamically and wants to see the changes in the results fast. Data archaeology, data visualization, and dynamic queries are examples of such applications. Views defined over the Internet tend to evolve and our technique can be useful for adapting such views.

We consider all possible redefinitions of SQL `SELECT-FROM-WHERE-GROUPBY-HAVING`, `UNION`, and `EXCEPT` views, and show how these views can be adapted using the old materialization for the cases where it is possible to do so. We identify extra information that can be kept with a materialization to facilitate redefinition. Multiple simultaneous changes to a view can be handled without necessarily materializing intermediate results. We identify guidelines for users and database administrators that can be used to facilitate efficient view adaptation.

We perform a systematic experimental evaluation of our proposed techniques. Our evaluation indicates that adaptation is much more efficient than rematerialization in most cases. In-place adaptation methods are better than the non-in-place methods when the change is small. We also point out some important factors that can impact the efficiency of adaptation.

1. INTRODUCTION

Materialized views are important for providing viable solutions to problems in applications related to decision support, data warehousing and data integration. A materialized view is like a data cache; the main reason for defining and storing a materialized view is to increase query performance. An important issue is how to maintain a materialized view when there are changes at the data sources and view redefinitions. Most previous view maintenance work deals with changes of the source relations. In this article, we consider the problem of keeping a materialized view up-to-date in response to changes made to the view definition, that is, in response to redefinition of the view. We call this problem the "view adaptation problem."

Many applications try to visualize data stored in a database. A view is defined, and a graphical display program may present the data in the view visually. If the user changes the view definition, the system must be able to recompute the view fast in order to keep the application interactive. An interface for such queries in a real estate system is reported in [WS93], where they are called *dynamic queries* [AWS93]. After posing a query, the user will analyze the result. He/she can modify the query (say by changing the price range of houses of interest) if too many or too few answers are presented.

Data archaeology [BST$^+$92, BST$^+$93] is another application where an archaeologist tries to discover rules about data by formulating queries, looking at the results (which can be cached) of the query, and then changing the query iteratively as the archaeologist's understanding improves.

*A preliminary version of this article appeared as [GMR95].

†Part of the research was done while the author was working at AT&T Laboratories.

‡The work of Jun Rao was performed while at Columbia University.

§Research supported by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, by NSF grants IRI-9209029, CDA-90-24735, and by an NSF Young Investigator award.

In an environment such as the Internet, information sources are autonomous and are free to update their schemas [Bel00]. Besides, the number of sources can also vary over time. As a result, the definition of a view over such independent sources is also likely to change.

In this article, we assume that the underlying data don't change during view adaptation. This is a reasonable assumption in data warehouse environments where data updates are often batched.

1.1. Motivating Example

Example 1 Suppose a customer wants to buy a car and defines the following view to check the price and finance from two dealers on the Web:

```
CREATE VIEW V AS
SELECT D1.CarID, D1.Price, D1.Finance, D2.Price, D2.Finance
FROM   Dealer1 D1, Dealer2 D2
WHERE  D1.CarID = D2.CarID AND
       D1.Price < 20000 AND D2.Finance < 0.06
```

Say the customer redefines the view by lowering the price limit from *Dealer*1 to 15000. The answer to this new view can be computed easily from the answer in the original view. We just need to remove all those cars that have *Price* less than 20000 but not less than 15000 from the first dealer. This incremental computation is much more efficient than recomputing the view from scratch.

Not all changes to the view definition are so easily computable. For instance, if *Price* is changed to a higher threshold of interest to *D1.Price* < 25000, then the above computation is not possible. However, we can still infer that (a) the old tuples still need to be kept in the view and (b) some more tuples need to be added, namely, those cars that have *Price* less than 25000 but not less than 20000 from *Dealer*1 satisfying the finance restriction from *Dealer*2. Thus, even though the new query is not entirely computable using the answer to the old query, it is possible to substantially reduce the amount of recomputation.

Now, say the customer wants to add a third dealer for comparison and changes the view definition to the following:

```
CREATE VIEW V AS
SELECT D1.CarID, D1.Price, D1.Finance, D2.Price, D2.Finance, D3.Price, D3.Finance
FROM   Dealer1 D1, Dealer2 D2, Dealer3 D3
WHERE  D1.CarID = D2.CarID AND D1.CarID = D3.CarID AND
       D1.Price < 20000 AND D2.Finance < 0.06
```

To compute this new view, we need only look up the *Price* and *Finance* attribute from *Dealer*3 using the value of *CarID* for each tuple in the current answer set. This is likely to be much cheaper than computing a three-way join from scratch. □

1.2. Related Work

Previous work on using views to answer queries includes [LY85, YL87, SJGP90, TSI94, CR94, CKPS95, LMSS95, RSU95]. Subsequently, [GHQ95, SDJY96] discuss how to answer aggregate queries using materialized aggregate views. [TSI94, CKPS95] also describe how to incorporate their techniques in a traditional cost-based query optimizer. The problem of view adaptation is related to the problem of query answering using views. However, a fundamental difference is that view adaptation doesn't have to compute the full answer set for the new view. For example, if the new view has one fewer tuple (say t) than the old view, view adaptation can simply delete t from the old view. The query answering approach would have to compute an answer set including all the tuples but t from the old view. We refer to such adaptation methods as in-place methods as the view is modified directly. Query answering is similar to our non-in-place methods which require computing the full set to the new view. The in-place methods can be much better than the non-in-place ones as discussed in the experimental section (Section 7). Also, our adaptation techniques involving aggregates (techniques 24 to 27 described in Section 4.5) can adapt the original view when a group is partially modified. To the best of our knowledge, no previous work has similar methods since they all require that the underlying set for each group be the same.

View adaptation differs from query answering also in that adaptation assumes the new view definition is "close" to the old view definition, in the sense that the view changes via a small set of local changes. There is no such assumption in the query answering problem, which means that a query compiler/optimizer may have to spend more time determining how to use the existing views to *correctly* answer a given query.

Traditional view maintenance tries to maintain a materialized view in response to modifications to the base relations [BLT86, CW91, GL95, GMS93]. Related issues include when and how to bring a materialized view up-to-date [CKLMR97], and consistency of queries to views in the phase of update [KLMQR97]. View adaptation differs from view maintenance in that only the view definition, not the underlying data is changing.

Query modification [Han87] considers how to translate a query using views to a query on base tables directly. Since adaptation considers the query to be the new view itself, query modification is essentially rematerialization.

[YL95] considers whether a group-by should be performed before a join by checking the availability of functional dependencies. Such information is also used in adaptation techniques involving aggregates.

"Classic" [BBMR89] is a system developed at AT&T Bell Laboratories that allows users to define new concepts and optimizes the evaluation of their extents by classifying the concepts in a concept hierarchy, and then computing them starting with the parent concepts. This corresponds to evaluating a new Classic query (the new concept), using information in several materialized views (the old concepts). Classic has been used for data archaeology.

[TSI94] focuses on the broader issue of enhancing physical data independence using "gmaps." They use a logical schema and then specify the underlying physical storage structures as results of "gmap" queries on the logical schema. User queries on the logical schema are rewritten using one or more gmap queries that access the physical structures. The gmap and user queries are SPJ expressions. Query translation is similar to using only existing views (gmaps) to compute new views (user queries).

1.3. Results

We define the process of redefining a view as a sequence of local changes in the view definition. The adaptation is expressed as an additional query or update upon the old view and the base relations that needs to be executed to adapt the view in response to the redefinition. We identify a basic set of local changes so that a sequence of local changes can be maintained by composing the maintenance process for each local change. In almost all cases, this composition can be performed without materializing the intermediate results, yielding a single adaptation method for arbitrary changes to a view definition.

We present a *comprehensive* study of different types of local changes that can be made to a view, and present algorithms to maintain the views in response to these changes. Many techniques have both in-place and non-in-place versions. Our techniques can be integrated into a conventional cost-based query optimizer. The optimizer considers the additional plans provided by our adaptation algorithms and uses one of them if its cost is lower than the cost of rematerializing the view.

We show that the maintenance in response to a redefinition is facilitated by keeping a small amount of extra information (beyond the view definition's attributes themselves). We only consider information that can be maintained efficiently, and show how the adaptation process can be made possible and far more efficient with this information.

We then present a thorough experimental evaluation of the adaptation techniques. Our work shows that (a) adaptation is often significantly better than recomputing the view, and (b) there are tradeoffs between the in-place and the non-in-place adaptation methods; small changes favor the in-place versions, (c) if one knows in advance that a view might change in certain ways, one can include appropriate kinds of additional information in the view.

In this article, we focus on changing a single materialized view, and on recomputing the new materialization using the old materialization and the base relations. We do not consider how multiple materialized views may be used to further assist the adaptation process.

The rest of this article is organized as follows: Section 2 describes the system model. Section 3, Section 4 and Section 5 introduce our adaptation techniques for `SPJ` views, views with aggregation and views with set operations respectively. We consider complex view changes in Section 6. We present our experimental results in Section 7 and conclude in Section 8.

2. THE SYSTEM MODEL

2.1. Notation

We consider simple SQL `SELECT-FROM-WHERE` views, in addition to views definable using `UNION`, difference (`EXCEPT`) and aggregation (`GROUPBY`). We use a syntactic shorthand "&" to avoid having to write down all the equality conditions in a natural join.

$$\begin{array}{ll} \texttt{SELECT} & A_1, \ldots, A_n \\ \texttt{FROM} & R_1 \,\&\, \cdots \,\&\, R_m \\ \texttt{WHERE} & C_1 \texttt{ AND } \cdots \texttt{ AND } C_k. \end{array}$$

When the relations in the `FROM` clause are separated by ampersands rather than commas, we mean that the relations $R_1, \ldots, R_n$ are combined by a natural join over all attributes that are mentioned in more than one relation. If we want an equijoin that is not a natural join, we shall specify the equijoin condition in the `FROM` clause rather than in the `WHERE` clause, inside square brackets. Join conditions that are not equijoins or natural joins will be specified in the `WHERE` clause. The conditions $C_1, \ldots, C_k$ are basic, i.e., non-conjunctive conditions. The order in which we write the conditions or the relations is not important.

When we perform schema changes, we use standard `SQL3` "`ALTER TABLE`" and "`UPDATE`" statements.

Relations will be of two types – base relations and view relations. Base relations are physically stored by the system, and are updated directly. The view relations are defined as views (i.e., queries) over base relations and other view relations. A *materialized* view relation has its extension physically stored by the system. Materialized views are not updated directly; updates on the base relations and other view relations are translated by a view maintenance algorithm into updates to the materialized view.

Definition 1 Key Attributes A *key* of a relation R is a minimal subset of the attributes of R that uniquely identifies tuples in R.

Definition 2 Redefinable Views A *redefinable view* is a view whose definition can be changed.

A relation may have several keys, and any one of these could be used in any of the results we derive. Key information will be used in the analysis for view changes.

Adaptation and Recomputation

When view V is redefined, let the new extent of V be called E. When E is obtained utilizing the previously materialized extent of view V, the process will be called *adapting* view V. When E is obtained by evaluating the view definition, without utilizing the previously materialized extent of view V, the process will be called *recomputing* (or *rematerializing*) view V. We can look upon a recomputation as a special case of adaptation where the previously materialized extent of view V is not used profitably.

2.2. View Adaptation Issues

We make the minimalistic assumption that the redefinition is expressed as a sequence of primitive local changes. Each local change is a small change to the view definition. For example, dropping or changing a selection predicate, adding an attribute to the result, changing the grouping list, and adding a join relation are all examples of local changes. We shall consider sequences of local changes (without necessarily materializing intermediate results) in Section 6.

Given a redefinable view, the system and/or the database administrator has to first determine (a) whether the view should be augmented with some extra information to help with later adaptation, (b) how the materialized view should be stored (maybe keep some free space for each tuple to grow; maybe physically order the view by a particular attribute), and (c) whether the materialized view should be indexed. We will give recommendations for augmentation in Sections 3.5, 4.6 and 5.3. We discuss the choice of clustering and indexing in Section 7.

We can incorporate view adaptation techniques into a conventional query optimizer. A cost-based optimizer can be extended by pre-generating the best plan for each possible adaptation method. Normally, only one in-place and one non-in-place adaptation method will be applicable to a specific change. The optimizer can then choose the overall best plan among adaptation plans and rematerialization on a cost basis. A similar approach has been used in [TSI94]. Alternatively, we can implement each adaptation technique (after local changes) as an additional rule in the query rewriting phase. The observations we summarized in the experimental section (Section 7) can be used to guide the design of those heuristic rules.

We aim to use the adaptation technique in systems that are queried more frequently than updated. As argued in [Fre95, Fre97], an OLAP system separated from the traditional OLTP system can provide much better query performance. In such systems, updates are batched and data are changed only periodically. As a result, we assume that there are no data changes during view adaptation in this article. When batch updates arrive, we can lock all the corresponding tables and bring all the views up-to-date using conventional view maintenance methods.

2.3. Primitive changes

We support the following changes as primitive local changes to a view definition.

- Addition or deletion of an attribute in the `SELECT` clause.
- Addition, deletion, or modification of a predicate in the `WHERE` clause (with and without aggregation).
- Addition or deletion of a join operand (in the `FROM` clause), with associated equijoin predicates and attributes in the `SELECT` clause.
- Addition or deletion of an attribute from the `GROUPBY` list.
- Addition or deletion of an aggregation function to a `GROUPBY` view.
- Addition, deletion, or modification of a predicate in the `HAVING` clause. Addition of the first predicate or deletion of the last predicate corresponds to addition and deletion of the `HAVING` clause itself.
- Addition or deletion of an operand to the `UNION` and `EXCEPT` operators.
- Addition or deletion of the `DISTINCT` operator.

We will discuss each of these primitive changes, and outline an algorithm to adapt the view upon redefinition with the primitive change. As we consider each primitive change, we will build a table of alternative techniques to do the adaptation. Our discussion will be based on the following relations E (employees), W (works), and P (projects). The key of each relation is underlined.

$E(\underline{Emp\#}, Name, Address, Age, Salary)$.
$W(\underline{Emp\#}, \underline{Proj\#}, Hours)$.
$P(\underline{Proj\#}, Projname, Leader\#, Location, Budget)$.

3. `SELECT-FROM-WHERE` VIEWS

In this section we consider views defined by a basic `SELECT-FROM-WHERE` query and redefinitions that may change the `SELECT`, the `FROM`, and/or the `WHERE` clauses. For each type of possible redefinition, we show: (a) How to obtain the materialization of the redefinition, and (b) What extra information may be kept to facilitate adaptation.

A generic materialized view V may be defined as

$$\begin{array}{ll} \texttt{CREATE} & \texttt{VIEW}\ V\ \texttt{AS} \\ \texttt{SELECT} & A_1, \ldots, A_n \\ \texttt{FROM} & R_1\ \&\ \cdots\ \&\ R_m \\ \texttt{WHERE} & C_1\ \texttt{AND} \cdots \texttt{AND}\ C_k \end{array}$$

As discussed in Section 2.1, an equijoin is written in the `FROM` clause of a query. Thus, changes to the equijoin predicates are considered in the subsection on the `FROM` clause, while changes to other predicates are considered in the subsection on the `WHERE` clause.

3.1. Changing the `SELECT` *Clause*

Reducing the set of attributes that define a view V is straightforward: we can project out the unneeded attributes to get the new view. Alternatively, one could simply keep the old view V, and do the projection when V is being accessed.

Adding attributes to a view is more difficult. One solution, is to keep more attributes than those needed for V in an augmented relation W, and to perform the projection only when references to V occur. In that case, we can add attributes to the view easily if they are attributes of W.

The solution mentioned above may be appropriate for a small number of attributes. However, when there are several base relations and many attributes, keeping a copy of all of the attributes may not be feasible. In such cases, we shall prefer where possible to keep *foreign keys* in the view.

Example 2 Define a view V as

```
CREATE VIEW V AS
SELECT Name,Projname
FROM   E & W & P
WHERE  Salary ≥ 100,000
```

Keeping all of the attributes in an augmented relation would require maintaining eleven additional attributes. Alternatively, we could just keep *Emp#* and *Proj#* in addition to *Name* and *Projname* in an augmented relation, say G.

Suppose we wished to add the *Address* attribute to the view. We could do this addition incrementally by scanning once through relation G, and doing an indexed lookup on the E relation based on *Emp#*. This can be expressed as:

```
ALTER TABLE G ADD Address
UPDATE  G SET  Address = (SELECT Address
                          FROM   E
                          WHERE  E.Emp# = G.Emp#).
```

The update could be done in place, or it could be done by copying the result into a new version of G. A query optimizer could also rewrite the update statement into a join between E and G and modify the tuples of G as they participate in the join. In either case, the cost of updating G is easily estimated using standard cost-based optimization techniques, and is likely to be far less than recomputing the entire three-way join. $\square$

Often the original view itself keeps the key columns for one of the base relations. Thus, if view V includes the key for a base relation R, or the key of R is equated to a constant in the view definition, and a redefinition requires additional columns of R, then the view can be adapted by using the keys present in the old materialization of the view to pick the appropriate tuples from relation R.

Sometimes, adaptation can be done even in the absence of a key for R in the view. This requires more complicated checking similar to tests in [RSU95, LMSS95] to see if a query can be answered using views.

Changing the `DISTINCT` Qualifier.

Suppose that a user adds a `DISTINCT` qualifier to the definition of a view that did not previously have one. Thus we have to delete duplicate entries from the old view to obtain the new view. This adaptation is fairly simply expressed as a `SELECT DISTINCT` over the old view to obtain the new view. Deleting a `DISTINCT` qualifier is more difficult, since it is not clear how many duplicates of each tuple should be in the new view. A more detailed discussion appears in Section 3.4.

3.2. Changes in the `WHERE` *Clause (no aggregation)*

In this section we discuss changes to a condition in the `WHERE` clause. We do not distinguish between conditions on a single relation and conditions on multiple relations (i.e., "join conditions") in the `WHERE` clause.

Let C_1' be a new condition. (Without loss of generality, we assume we are changing C_1 to C_1' in our generic view.) We want to efficiently materialize V', which could be defined as

$$\begin{array}{ll}\texttt{CREATE} & \texttt{VIEW}\ V'\ \texttt{AS}\\ \texttt{SELECT} & A_1,\ldots,A_n\\ \texttt{FROM} & R_1\ \&\ \cdots\ \&\ R_m\\ \texttt{WHERE} & C_1'\ \texttt{AND}\cdots\texttt{AND}\ C_k\end{array}$$

by taking advantage of the fact that V has already been materialized.

Algebraically, $V' = V \cup V^+ - V^-$ where

$$V^+ = \begin{array}{ll}\texttt{SELECT} & A_1,\ldots,A_n\\ \texttt{FROM} & R_1\ \&\ \cdots\ \&\ R_m\\ \texttt{WHERE} & C_1'\ \texttt{AND NOT}\ C_1\ \texttt{AND}\ C_2\ \texttt{AND}\cdots\texttt{AND}\ C_k\end{array}$$

$$V^- = \begin{array}{ll}\texttt{SELECT} & A_1,\ldots,A_n\\ \texttt{FROM} & R_1\ \&\ \cdots\ \&\ R_m\\ \texttt{WHERE} & \texttt{NOT}\ C_1'\ \texttt{AND}\ C_1\texttt{AND}\ C_2\ \texttt{AND}\cdots\texttt{AND}\ C_k\end{array}$$

If the attributes mentioned by C_1' are a subset of $\{A_1,\ldots,A_n\}$, then

$$V^- = \ \texttt{SELECT}\ A_1,\ldots,A_n\ \texttt{FROM}\ V\ \texttt{WHERE NOT}\ C_1'$$

or

$$V - V^- = \ \texttt{SELECT}\ A_1,\ldots,A_n\ \texttt{FROM}\ V\ \texttt{WHERE}\ C_1'$$

V can thus be adapted as follows:

$$\texttt{DELETE FROM}\ V\ \texttt{WHERE NOT}\ C_1'$$

$$\begin{array}{ll}\multicolumn{2}{l}{\texttt{INSERT INTO}\ V}\\ \texttt{(SELECT} & A_1,\ldots,A_n\\ \texttt{FROM} & R_1\ \&\ \cdots\ \&\ R_m\\ \texttt{WHERE} & C_1'\ \texttt{AND NOT}\ C_1\ \texttt{AND}\ C_2\ \texttt{AND}\cdots\texttt{AND}\ C_k)\end{array}$$

Alternatively, if the attributes of C_1' are not available in the original view, the view adaptation algorithm for the SELECT clause could have materialized some extra attributes in an augmented relation W, as discussed in Section 3.1. In this case, even if C_1' mentioned an attribute not in $\{A_1,\ldots,A_n\}$, we could write V^- as above as long as all the attributes mentioned by C_1' were obtainable from the augmented view.

Thus we can see that the cost of adapting V in either of the cases above is (at most) one selection on V (or on the augmentation G) to adapt V into $V - V^-$, plus the cost of computing V^+ for insertion into V. As we shall see, in many examples the cost of computing V^+ will be small compared with the cost of recomputing V.

Example 3 Consider a view V defined by

$$\begin{array}{l}\texttt{CREATE VIEW}\ V\ \texttt{AS}\\ \texttt{SELECT * FROM}\ E\ \&\ W\ \texttt{WHERE}\ \mathit{Salary} > 50000\end{array}$$

Suppose that we wish to adapt V to

$$\texttt{SELECT * FROM}\ E\ \&\ W\ \texttt{WHERE}\ \mathit{Salary} > 60000$$

Let us refer to the new expression as V'. Using the terminology above, we see that C_1 is "$\mathit{Salary} > 50000$" and C_1' is "$\mathit{Salary} > 60000$." Hence V^- and V^+ can be defined as

$$\begin{array}{ll}V^- = & \texttt{SELECT * FROM}\ V\ \texttt{WHERE}\\ & \mathit{Salary} \le 60000\ \texttt{AND}\ \mathit{Salary} > 50000\\ V^+ = & \texttt{SELECT * FROM}\ E\ \&\ W\ \texttt{WHERE}\\ & \mathit{Salary} > 60000\ \texttt{AND}\ \mathit{Salary} \le 50000\end{array}$$

V^+ is empty, since its conditions in the WHERE clause are inconsistent with each other. Hence, the cost of recomputing the view is (at most) one pass over V. Now suppose that V' is defined by

$$\texttt{SELECT * FROM}\ E\ \&\ W\ \texttt{WHERE}\ \mathit{Salary} > 49000.$$

Then V^- is empty, and V^+ is given by

$$\texttt{SELECT * FROM}\ E\ \&\ W\ \texttt{WHERE}\ \mathit{Salary} > 49000\ \texttt{AND}\ \mathit{Salary} \le 50000.$$

If there is an index on salary in E, then (with a reasonable distribution of salary values) $V \cup V^+$ might be computed much more efficiently than recomputing V' from scratch. The query optimizer would have enough information to decide which is the better strategy. □

The same analysis holds even for join predicates. For example:

Example 4 Consider the view defined as

CREATE VIEW V AS
SELECT *Emp#*, *Salary*, *Proj#*, *Budget*
FROM E & W & P
WHERE $Salary > 0.2 * Budget$.

The join condition "$Salary > 0.2 * Budget$" could be changed to either "$Salary > 0.3 * Budget$" or "$Salary > 0.1 * Budget$" in a fashion similar to that of Example 3. □

Most queries that involve multiple relations use either equijoins or use single table selection conditions. For example, in one of our application environments, making efficient visual tools for browsing data, users are known to refine queries by changing the selection conditions on a relation interactively. Thus, it is likely that both the old condition C_1 and the new condition C_1' are single table selection conditions on the same attributes. Thus, the condition NOT C_1 AND C_1' can be pushed down to a single base relation, making the computation of V^+ more efficient.

Adding or Deleting a Condition

We can express the addition of a condition C' in the WHERE clause as a change of condition by adding some tautologically true selection to the old view definition V, then changing it to C'. The analysis above then means that V^+ is empty, and the new view can be computed as $V - V^-$, i.e., as a filter on the extension of V.

Similarly, the deletion of a condition is equivalent to replacing that condition by a tautologically true condition. In this case, V^- is empty, and the optimizer needs to compare the cost of computing V^+ with the cost of computing the view from scratch.

Here, we have an option of keeping an augmented view, containing attributes being augmented and not applying selection conditions until query time. The benefit is that no adaptation is needed if any condition changes in the future. But the augmentation is not always practical for two reasons: (a) the materialized view becomes significantly large, and (b) querying may be slowed down because all the conditions have to be reevaluated on the larger underlying data set. Clustering won't solve the second problem completely if there are more than one changing attribute.

3.3. Changing the FROM *Clause*

If we change an equijoin condition, then it is not clear that V^+ is efficiently evaluable. This corresponds to our intuition, which states that if an equijoin condition changes then there will be a dramatic change in the result of the join, and so the old view definition will not be much help in computing the new join result. We note that it is unlikely that the users will change the equijoin predicates.

Nevertheless, there are situations where we can make use of the old view to efficiently compute a new view in which we have either added or deleted relations from the FROM clause.

Adding a join relation

Suppose that we add a new relation R_{m+1} to the FROM clause, with an equijoin condition equating some attribute A of R_{m+1} to another attribute B in R_i for some $1 \leq i \leq m$. Suppose also that we want to add some attributes $D_1, \ldots, D_j$ from R_{m+1} to the view.

If B is part of the view, then the new view can be computed as

$$\texttt{SELECT } A_1, \ldots, A_n, D_1, \ldots, D_j \texttt{ FROM } V, R_{m+1} \texttt{ WHERE } A = B.$$

If the joining attribute A is a key for relation R_{m+1}, or we can otherwise guarantee that A values are all distinct, then we can express the adaptation as an update (we generalize SQL syntax to assign values to a list of attributes from the result of a subquery that returns exactly one tuple). For each of the updates below, we first apply the command "ALTER TABLE V ADD $D_1, \ldots, D_j$."

$$\texttt{UPDATE}\ V\ \texttt{SET}\ D_1,\ldots,D_j = (\texttt{SELECT}\ R_{m+1}.D_1,\ldots,R_{m+1}.D_j\ \ \texttt{FROM}\ R_{m+1}\ \ \texttt{WHERE}\ R_{m+1}.A = V.B).$$

If B is not part of the view, then it still may be possible to obtain B by joining V with R_i (assuming that V contains a key K for R_i) and hence compute the new view either as

$$\texttt{UPDATE}\ V\ \texttt{SET}\ D_1,\ldots,D_j = (\texttt{SELECT}\ R_{m+1}.D_1,\ldots,R_{m+1}.D_j\ \ \texttt{FROM}\ R_{m+1}, R_i\ \ \texttt{WHERE}\ R_{m+1}.A = R_i.B\ \texttt{AND}\ V.K = R_i.K).$$

if A is a key in R_{m+1}, or as

$$\texttt{SELECT}\ A_1,\ldots,A_n,D_1,\ldots,D_j\ \texttt{FROM}\ V,R_i,R_{m+1}\ \texttt{WHERE}\ A = B\ \texttt{AND}\ V.K = R_i.K.$$

if A is not guaranteed to be distinct in R_{m+1}.

Example 5 Suppose we have a materialized view of customers with their customer data, including their zip-codes. If we want to also know their cities, we can take the old materialized view and join it with our zip-code/city relation to get the city information as an extra attribute. □

Deleting a join relation

When deleting a join operand, one has to make sure that the number of duplicates is maintained correctly, and also allow for dangling tuples. For $R \bowtie S \bowtie T$, when the join with T is dropped, the system (1) needs to go back and find $R \bowtie S$ tuples that did not join with T, and (2) figure out the exact multiplicity of tuples in the new view. The former can be avoided if the join with T is on a key of T and if the system enforces referential integrity. The latter can be avoided if the view does not care about duplicates (`SELECT DISTINCT`), or if T is being joined on its key attributes.

3.4. Adapting `DISTINCT` `SELECT-FROM-WHERE` *views*

Removing the `DISTINCT` qualifier

It is usually difficult to adapt the view in response to this change. We discuss how adaptation may be done in some cases. If the old view contains a key for some of the base relations $R_1,\ldots,R_j$, but no keys from $R_{j+1},\ldots,R_m$, then the tuple multiplicity can be correctly determined by joining the old view with $R_{j+1},\ldots,R_m$ according to the original join conditions on $R_{j+1},\ldots,R_m$. If these original join conditions mention a nonkey attribute from $R_1,\ldots,R_j$ then the relations containing those attributes will also have to participate in the join.

An alternative is to augment the view so as to always keep a count of the number of derivations for each tuple in the view. In this case, changes to the `DISTINCT` qualifier can be handled easily by either presenting the count to the user, or by hiding the count.

Changing the `SELECT` clause

These changes are handled exactly as when the `SELECT-FROM-WHERE` view did not use a `DISTINCT` qualifier.

Changing a condition in the `WHERE` clause

Recall that adapting a view in response to changes to the `WHERE` clause involved computing a set V^+ and a set V^-. Incorporating the set V^-, even if it is computable, is the difficult part of adapting V if it uses the `DISTINCT` qualifier. The reason is that if duplicates are eliminated from a view then deletions become difficult in the absence of counts. Thus, the difference in handling `SELECT-FROM-WHERE` with `DISTINCT` as compared to views without `DISTINCT`, arises in the way V^- is handled. Insertions are handled as before (albeit with a duplicate elimination step that also correctly updates counts).

If the attributes of C'_1 are not present in the `SELECT` clause then counts are retained with the original view in order to correctly incorporate the value of V^- computed as described in Section 3.2. If the attributes in C'_1 are all present in the `SELECT` clause then the following query correctly updates V:

$$\texttt{DELETE FROM}\ V\ \texttt{WHERE NOT}\ C'_1$$

Adding/deleting a condition in the `WHERE` clause

If a condition is deleted then tuples are only added to the view and thus the discussion of a non-`DISTINCT` view applies. However, if a condition is added, then tuples are deleted from a view thus requiring counts to be maintained in the original view.

Changing the `FROM` clause

Changes to the `FROM` clause are handled as in the case when the view did not use the `DISTINCT` qualifier.

3.5. Summary: `SELECT-FROM-WHERE` *Views*

As described earlier, the cost of the adaptation technique can be significantly less than the cost of recomputing from scratch. Also, since the adaptation techniques are `SQL` style query/update statements, their cost can be estimated by the optimizer. Table 4 and Table 5 in Appendix A summarizes our adaptation techniques for `SELECT-FROM-WHERE` queries. We assume that the initial view definition is as stated at the beginning of Section 3. For each possible redefinition, we give the possible adaptations along with the assumptions needed for the adaptation to work. The assumptions are listed separately in Table 6 in Appendix A.

Table 4 can be used in three ways. Firstly, the query optimizer would use this table to find the adaptation technique (and compute its cost estimate) given the properties of the current schema vis-a-vis the assumptions stated in the table. Secondly, a database administrator or user would use this table to see what assumptions need to hold in order to make incremental view adaptation possible at the most efficient level. Given this information, the views can be defined with enough extra information so that view changes can be computed most efficiently. Note that different collections of assumptions make different types of incremental computation possible, so that different "menus" of extra information stored should be considered. Thirdly, the database administrator could interact with the query optimizer to see which access methods and indexes should be built, on the base relations and on the materialized views, in order to facilitate efficient adaptation.

Recommendations for Augmentation.

To make adaptations more applicable, we need to keep the keys of referenced relations from which attributes may be added and attributes referenced by the selection conditions in the view definition. We may also need to keep the count of the number of derivations for each tuple.

Augmentation can also be used to make adaptations more efficient. In systems that store tables row by row, we may store the view with padding in each tuple for future in-place horizontal expansion. Another option is to materialize a larger data set (postponing selection). But that's trading space and querying response time for the ease of adaptation. We can choose to keep more data than necessary if the data set is within a threshold.

4. AGGREGATION VIEWS

In this section, we show how to adapt views when grouping columns and/or the aggregate functions change in a materialized SQL aggregation view.

Example 6 We could express the total salaries charged to a project with the following materialized view: We assume that an employee is nominally employed for 40 hours per week, and that if an employee works more or less, a proportional salary is paid. Thus the charge to a project for an employee is obtained by multiplying the salary by the fraction of the 40 hour week the employee works on the project.

$$\begin{array}{ll}
\texttt{CREATE} & \texttt{VIEW}\ V(\textit{Proj\#, Location, Proj_Sal})\ \texttt{AS} \\
\texttt{SELECT} & \textit{Proj\#, Location, SUM}((\textit{Salary} \times \textit{Hours})/40) \\
\texttt{FROM} & E\ \&\ W\ \&\ P \\
\texttt{GROUPBY} & \textit{Proj\#, Location}
\end{array}$$

Suppose we want to modify V so that it gives a location-by-location sum of charged salaries. This modification corresponds to removing the *Proj#* attribute from the list of grouping variables and output variables, to give the following view definition:

```
CREATE   VIEW V'(Location, Proj_Sal) AS
SELECT   Location, SUM((Salary × Hours)/40)
FROM     E & W & P
GROUPBY  Location
```

Using the commutativity properties of *SUM*, the query optimizer can observe that V' can be materialized as

```
SELECT   Location, SUM(Proj-Sal)
FROM     V
GROUPBY  Location
```

In this way we can use the original view to redefine the materialized view more efficiently.

Next, suppose we want to modify V to compute the sum of charged salaries for each *Proj#*. We can adapt V simply by dropping the *Location* attribute because *Proj#* is the key for relation P and functionally determines *Location*. The redefined groups are the same as before. □

4.1. *Dropping* GROUPBY *Columns*

Given an aggregation view, the set of tuples in the grouped relation that have the same values for all the grouping attributes is called a *group*. Thus, for the original view in Example 6, there is one group of tuples for each pair of (*Proj#*, *Location*) values. For the redefined view, there is one group of tuples for each (*Location*) value.

When a grouping attribute is dropped, each redefined group can be obtained by combining one or more original groups, so we can try to get the aggregation function over the redefined groups by combining the aggregation values from the combined groups. For instance, in Example 6, after dropping the *Proj#* attribute, the sum for the group for a particular (*Location*) value was obtained from the sum *Proj-Sal* of all the groups with this *Location*. When we dropped the *Location* attribute, we inferred that each redefined group was obtained from a single original group. So no new aggregation was needed.

A materialized view can be adapted when grouping columns are dropped if:

- The dropped column is functionally determined by the remaining grouping columns, or
- The aggregate functions in the redefined view are expressible as a computation over one or more of the original aggregation functions and grouping attributes. Table 1 lists a few aggregation functions that can be computed in such a manner.

Redefined Aggregation	**Adaptation using Original View**
$MIN(X)$	$MIN(M)$ where $M = MIN(X)$ was an original aggregation column.
$MAX(X)$	$MAX(M)$ where $M = MAX(X)$ was an original aggregation column.
$MIN(X)$	$MIN(X)$, where X was an original grouping column.
$MAX(X)$	$MAX(X)$, where X was an original grouping column.
$SUM(X)$	$SUM(S)$ where $S = SUM(X)$ was an original aggregation column.
$SUM(X)$	$SUM(X \times C)$, where $C = COUNT(*)$ was an original aggregation column, and X was an original grouping column.
$COUNT(*)$	$SUM(C)$ where $C = COUNT(*)$ was an original aggregation column.
$AVG(X)$	$SUM(A \times C)/SUM(C)$ where $C = COUNT(*)$ and $A = AVG(X)$ were original aggregation columns.
$AVG(X)$	$SUM(X \times C)/SUM(C)$ where $C = COUNT(*)$ was an original aggregation columns, and X was an original grouping column.

Table 1: Aggregate functions for a group defined as functions of subgroup aggregates.

Table 1 is meant to be illustrative, and not exhaustive. Several other aggregation functions may be decomposed in this manner.

4.2. Adding GROUPBY *Columns*

In general, when adding a groupby column, we would need to go back to the base relations since we are looking to aggregate data at a finer level of granularity. However, in case the added attribute is functionally determined by the original grouping attributes, we can add it just like we add a new projection column (Section 3.1).

Example 7 Consider the aggregation view defined first in Example 6, and suppose we want to add the leader of each project to the grouping column. The redefined view now is

CREATE VIEW $V'(Proj\#, Location, Leader\#, Proj\_Sal)$ AS
SELECT $Proj\#$, $Location$, $Leader\#$, $SUM((Salary \times Hours)/40)$
FROM E & W & P
GROUPBY $Proj\#$, $Location$, $Leader\#$

Since $Leader\#$ is functionally determined by $Proj\#$, we can adapt the original view to V' by:

ALTER TABLE V ADD $Leader\#$
UPDATE V SET $Leader\# =$ (SELECT $P.Leader\#$
FROM P
WHERE $P.Proj\# = V.Proj\#$)

□

Another situation where we can add GROUPBY columns is when there was no grouping or aggregation before. In that case, the new view is formed simply by applying the grouping and aggregation over the old view, assuming that the attributes needed for the grouping and aggregation are present in the old view. Even if the needed attributes are not present, they can be added in many cases, as discussed previously.

4.3. The HAVING *Clause*

The HAVING clause behaves in a similar fashion to the WHERE clause in many ways, from the point of view of adaptation. Adding, deleting, or changing a conjunct in the HAVING clause can be handled using the techniques of Section 3.2. If the HAVING clause refers to an aggregate that is not in the view definition, then one possible augmentation would be to keep that aggregate in the view. That way, one could adapt the view efficiently if the condition in the HAVING clause was modified.

The cost of adapting the HAVING clause may be higher than making a similar adaptation to the WHERE clause. Consider the following example.

Example 8 Consider the following view based on the view of Example 6. The WHERE clause restricts the view to projects with a budget of more than \$1000, while the HAVING clause restricts the view further to projects having more than 5 employees.

CREATE VIEW $V(Proj\#, Location, Proj\_Sal)$ AS
SELECT $Proj\#$, $Location$, $SUM((Sal \times Hours)/40)$
FROM E & W & P
WHERE $Budget > 1000$
GROUPBY $Proj\#$, $Location$
HAVING $COUNT(*) > 5$

Suppose that the view is augmented with the *Budget* attribute and the $COUNT(*)$ aggregate for adaptation purposes. Changing $Budget > 1000$ or $COUNT(*) > 5$ to a *stronger* condition is straightforward, and can be expressed as a selection on the old view.

Changing $Budget > 1000$ to $Budget > 900$, say, can be handled in an efficient manner if an index is available on the *Budget* attribute in the P relation. However, it is unlikely that there is any access method that would aid adaptation if $COUNT(*) > 5$ was changed to $COUNT(*) > 3$, for example. Without such an access method, one may have to recompute the aggregate on *all* groups that were not previously in the view. □

Example 8 suggests that it may be particularly important for views with aggregates to keep additional tuples beyond those satisfying the view. In the example above, we might materialize a larger view W that does not restrict the $COUNT$ aggregate. V can then be expressed as a selection and a projection on W. In this way we can more efficiently adapt to changes in the `HAVING` clause.

4.4. Dropping/Adding Aggregation Functions

Adapting a view to drop an aggregation function is straightforward, similar to the case where a column is projected out (Section 3.1). However, it is not possible to adapt to most additions of aggregation functions, unless the new function can be expressed in terms of existing functions, or unless the aggregation view is significantly augmented.

One type of augmentation requires storing the key values (or tuples of key values) of all tuples in each group in the view. For normalization reasons, one would want to keep such keys in a separate relation, and so this kind of augmentation is more general than the kind of augmentation considered elsewhere in this article. Due to the size of the augmented view, this particular kind of augmentation is beneficial for very limited kinds of adaptation. Hence, we do not pursue it further here.

4.5. Changes in the `WHERE` *Clause (in the presence of aggregates)*

Familiar as it looks, we are actually facing a different problem from the one in Section 3.2 where we considered changing the `WHERE` clause in a view without aggregates. In Section 3.2, the old view contains the individual tuples themselves. Here, the old view contains only the aggregate of the qualifying tuples. In order to adapt the old view, we need to find the difference set of the underlying tuples between the new view and the old view, compute the aggregate functions on the set and then adjust the result in the old view. We consider adding and deleting a condition separately.

Adding a Condition

Suppose we have the old view V and the new view V' defined as

`CREATE`	`VIEW` $V(A_1,\ldots,A_n,M_1,\ldots,M_j)$ `AS`	`CREATE`	`VIEW` $V'(A_1,\ldots,A_n,M_1,\ldots,M_j)$ `AS`
`SELECT`	$A_1,\ldots,A_n,\ F_1(B_1),\ldots,F_j(B_j)$	`SELECT`	$A_1,\ldots,A_n,\ F_1(B_1),\ldots,F_j(B_j)$
`FROM`	$R_1\ \&\ \cdots\ \&\ R_m$	`FROM`	$R_1\ \&\ \cdots\ \&\ R_m$
`WHERE`	C_1 `AND` $\cdots$ `AND` C_k	`WHERE`	C_0 `AND` C_1 `AND` $\cdots$ `AND` C_k
`GROUPBY`	$A_1,\ldots,A_n$	`GROUPBY`	$A_1,\ldots,A_n$

Since the new view is more restrictive than the old one, we can be assured that there will no more groups in the new view. For each group in the old view, we select those tuples in the group that doesn't satisfy the new condition C_0 and calculate the aggregates on the set selected. We use $\Delta F_i(B_i)$ to denote the aggregate result for a particular group (in V) and it can be calculated using the following SQL statement:

$$\Delta F_i(B_i) = \begin{array}{ll} \texttt{SELECT} & F_i(B_i) \\ \texttt{FROM} & R_1\ \&\ \cdots\ \&\ R_m \\ \texttt{WHERE} & \texttt{NOT } C_0 \texttt{ AND } C_1 \texttt{ AND } \cdots \texttt{ AND } C_k \texttt{ AND} \\ & A_1 = V.A_1 \texttt{ AND } \cdots\ A_n = V.A_n \end{array} \qquad (1)$$

We can then use $\Delta F_i(B_i)$ to compensate the aggregate values (M_i) in the old view. Here is the formula:

$$M_i = H_i(M_i,\ \Delta F_i(B_i)),\text{ for each }(A_1,\ldots,A_n)\text{ in the old view }V.$$

H_i is a function that depends on the specific aggregate F_i and its choices are listed in Table 2.

However, we need to be careful when we update the old view. If no tuple satisfies the conditions in the `WHERE` clause in Equation (1) for a particular group, $\Delta F_i(B_i)$ will return a NULL value (i.e., the underlying tuple set for that group doesn't change after redefinition). Since H_i will become NULL if any of its parameters is NULL, this may cause M_i to be updated as a NULL. In this case, what we actually want is to leave the old aggregate value for that particular group unchanged. So we need to restrict the `UPDATE` using a `WHERE` clause so that the update is performed only for those groups whose underlying tuple sets have changed under the new condition C_0.

F_i	**Corresponding** H_i
$COUNT$	$M_i - \Delta COUNT(*)$
SUM	$M_i - \Delta SUM(B_i)$
AVG	$M_i = (M_i * M_j - \Delta SUM(B_i))/(M_j - \Delta COUNT(*))$, where $M_j = COUNT(*)$ was an original aggregation column
MIN	not available
MAX	not available

Table 2: H_i functions for Adding a Condition.

If all the underlying tuples for a group have been deleted, the group itself should also be removed. But we need to keep a $COUNT(*)$ for each group in the original view to be able to do this. So the original view needs to be augmented if necessary.

Here is an example that illustrates the whole process.

Example 9 Consider a view V defined by

CREATE VIEW V(Location, Num) AS
SELECT $Location$, $COUNT(*)$
FROM P
WHERE $Budget > 10,000$
GROUPBY $Location$

Suppose that we define the new view by changing the WHERE clause to

WHERE $Budget > 20,000$

Here is the adaptation:

```
UPDATE  V
SET     Num =   Num -
                (SELECT COUNT(*) FROM P
                WHERE NOT (Budget > 20,000) AND Budget > 10,000 AND
                P.Location = V.Location)
WHERE   EXISTS  (SELECT * FROM P
                WHERE NOT (Budget > 20,000) AND Budget > 10,000 AND
                P.Location = V.Location)
DELETE  V
WHERE   Num = 0
```

The EXISTS predicate makes sure only appropriate groups will be updated to avoid the side effect of a NULL change as we have explained. □

Notice that the attribute of C_0 ($Budget$ in the above example) doesn't have to exist in the original view since we are querying the base tables directly. This is different from the requirement in Section 3.2 where there is no aggregate in the view.

Deleting a Condition

If the new view V' is defined as

```
CREATE   VIEW V'(A_1, ..., A_n, M_1, ..., M_j) AS
SELECT   A_1, ..., A_n, F_1(B_1), ..., F_j(B_j)
FROM     R_1 & ··· & R_m
WHERE    C_2 AND ··· AND C_k
GROUPBY  A_1, ..., A_n
```

Now, more underlying tuples will satisfy the selection condition in the new view. These tuples belong to either the groups already in the old view or some new groups. We can divide the tuples into two sets based on whether they belong to the groups in the old view or not. We then adapt the old view in two steps. First, we update the aggregate values in the old view using tuples in the first set. This is similar to the update we

F_i	**Corresponding H_i'**
COUNT	$M_i + \Delta COUNT(*)$
SUM	$M_i + \Delta SUM(B_i)$
AVG	$(M_i * M_j + \Delta SUM(B_i))/(M_j + \Delta COUNT(*))$, where $M_j = COUNT(*)$ was an original aggregation column
MIN	$M_i = MIN(M_i, \Delta MIN(B_i))$
MAX	$M_i = MAX(M_i, \Delta MAX(B_i))$

Table 3: H_i' functions for Deleting a Condition.

used when a condition is added. Next, we calculate the aggregates for the new groups using tuples in the second set and insert them into the original view.

The following is the formula for the update in the first step. Here a different function H_i' is used. We summarize its choices in Table 3.

$$
\begin{array}{ll}
M_i = H_i'(M_i, \Delta F_i(B_i)), & \\
\text{where } \Delta F_i(B_i) = & \texttt{SELECT} \quad F_i(B_i) \\
& \texttt{FROM} \quad R_1 \ \& \ \cdots \ \& \ R_m \\
& \texttt{WHERE} \quad \texttt{NOT } C_1 \texttt{ AND } C_2 \texttt{ AND} \cdots \texttt{AND } C_k \texttt{ AND} \\
& \qquad\qquad A_1 = V.A_1 \texttt{ AND} \cdots A_n = V.A_n \\
\text{for each } (A_1, \ldots, A_n) \text{ in the old view } V. &
\end{array}
$$

The second step is listed in Table 8 in Appendix A.

We can adapt a view with only scalar aggregates (i.e., no groupby clause) in a similar way since it's just a special case of views with groupbys. The details of the corresponding techniques can be found in Table 8 in Appendix A.

As we have seen in the motivating example, it's impractical to try to augment the view in a way similar to Section 3.2 and apply selections upon querying. This requires us to keep the information for all the combinations of possible adaptation parameters for all the groups. A more practical augmentation is to omit the `WHERE` clause and `GROUPBY` clause in the view and store the complete underlying data set. Although adaptation becomes trivial, we have to store information at a finer level, which takes more space, and the querying will be slowed down even more because of the need to recompute the aggregates.

4.6. Summary: `GROUPBY` *Views*

We assume that the initial view definition is

$$
\begin{array}{ll}
\texttt{CREATE} & \texttt{VIEW } V \texttt{ AS} \\
\texttt{SELECT} & A_1, \ldots, A_n, F_1(B_1), \ldots, F_j(B_j) \\
\texttt{FROM} & R_1 \ \& \ \cdots \ \& \ R_m \\
\texttt{WHERE} & C_1 \texttt{ AND} \cdots \texttt{AND } C_k \\
\texttt{GROUPBY} & A_1, \ldots, A_p
\end{array}
$$

where $p \geq n$.

The full list of adaptation techniques for aggregate views is given in Appendix A in Tables 7 and 8. The assumptions used are listed in Table 9. Tables 7 and 8 can be used in the same ways as Table 4.

Recommendations for Augmentation.

Table 1 illustrates that redefinition can be helped tremendously if the views are augmented with a $COUNT(*)$ aggregate. If the `HAVING` clause mentions an aggregate not in the view, then augment the view with this aggregate.

5. UNION AND DIFFERENCE VIEWS

5.1. `UNION`

A view V may be defined as the union of subqueries, say V_1 and V_2 (for now, we assume the `UNION ALL` semantics where duplicates are not eliminated). If the definition of V changes by a local change in either

V_1 or V_2 but not both, then it would be advantageous to apply the techniques developed in the previous sections to incrementally update either the materialization of V_1 or V_2 while leaving the other unchanged.

In order to do this, we need to know which tuples in V came from V_1 and which from V_2. With this knowledge, we can simply keep the tuples from the unchanged part of the view, and update the changed part of the view. Thus it would be beneficial to store with each tuple an indication of whether it came from V_1 or V_2. Alternatively, one could store V_1 and V_2 separately, and form the union only when the whole view V is accessed.

Example 10 Suppose we want the names of employees who either work on a project located in New York, or who manage a project located in New York. We can write this view V as V_1 `UNION` V_2 where V_1 and V_2 are as follows.

V_1 = `SELECT` *Name*,*SubQ*=“V_1”
`FROM` E & W & P
`WHERE` *Location*=*New-York*
V_2 = `SELECT` *Name*,*SubQ*=“V_2”
`FROM` E, P [*E.Emp#* = *P.Leader#*]
`WHERE` *Location*=*New-York*

(We would probably choose not to display the *SubQ* field to the user, but to keep it as an attribute of a larger augmented relation.) If we wanted to change V_1 so that we get only employees working more than 20 hours per week, then we could do so using techniques developed in the previous sections for tuples in V with *SubQ*=“V_1”, and leave the other tuples unchanged. □

It is easy to delete a `UNION` operand if we keep track of which tuples came from which subqueries. We simply remove from V all tuples with the *SubQ* attribute matching that of the subquery being deleted.

Adding a union operand is also straightforward: The old union is unchanged, and the new operand is evaluated to generate the new tuples.

Note that our approach can also support the `UNION` eliminating duplicates. However, there will be an additional cost of sorting or hashing the view result.

5.2. `EXCEPT`

Example 11 Suppose we want the names of employees who work on a project located in New York, but who are not managers. We can write this view as V_1 `EXCEPT` V_2 where V_1 and V_2 are defined as follows.

`CREATE VIEW` V_1 `AS`
`SELECT` *Name* `FROM` E & W & P
`WHERE` *Location*=*New-York*

`CREATE VIEW` V_2 `AS`
`SELECT` *Name* `FROM` E, P [*E.Emp#* = *P.Leader#*]
`WHERE` *Location*=*New-York*

□

Unlike the case for unions, the extension of V could conceivably be much smaller than the extensions of either V_1 or V_2. Thus, we cannot argue that in general we should keep all of the V_1 and V_2 tuples with an identification of whether they came from V_1 or V_2.

However, in two cases we can still use information in the old view to compute the new view more efficiently.

1. If V_2 is replaced by a view V_2' that is strictly weaker (i.e., contains more tuples) than V_2, then we can observe that V_2^- is empty, and $V' = V$ `EXCEPT` V_2^+.

2. If V_1 is replaced by a view V_1' that is strictly stronger (i.e., contains fewer tuples) than V_1, then we can observe that V_1^+ is empty, and $V' = V$ `EXCEPT` V_1^-.

If we want to subtract a new subquery V_2 from an existing materialized view V, then we can do so efficiently using the first observation above. In that case, the new view V' is V EXCEPT V_2 and we can make use of the old extension of V.

In the general case, there is another possibility that the optimizer can consider for computing V'. Suppose that V_2 changes with both V_2^+ and V_2^- nonempty. The new answer is V EXCEPT V_2^+ UNION U where U is $V_1 \cap V_2^-$. While we probably have not materialized V_1, we can still evaluate U by considering each tuple in V_2^- and *checking* that it satisfies the conditions defining V_1. If V_2^+ and V_2^- are small, then this strategy will still be better than recomputing V' from scratch. A symmetric case holds if V_1 changes rather than V_2. In order for this strategy to be effective, the query optimizer needs to estimate the sizes of V_2^+ and V_2^-. For simple views V_2 this may be achieved using selectivity information and information about the domains of the attributes. For complicated queries, it may be hard to estimate these sizes.

5.3. Summary: Views with Union and Difference

We assume that the initial view definition is either

CREATE VIEW V AS
V_1 UNION V_2

or

CREATE VIEW V AS
V_1 EXCEPT V_2

The full list of adaptation techniques for union and difference views is given in Appendix A in Table 10. The assumptions used are listed in Table 11. Table 10 can be used in the same ways as Table 4.

Recommendations for Augmentation.

Keep an attribute identifying which subquery in a union each tuple came from.

6. COMPLEX CHANGES TO A VIEW DEFINITION

It is conceivable that a user might want to make several simultaneous changes to a view definition. One may easily concatenate several of the basic techniques to obtain the new view. However, that strategy would materialize all of the intermediate results, which may not be necessary.

For example, if more than one condition in the WHERE clause is simultaneously changed, then the analysis of Section 3.2 still applies, but thinking of C_1 and C_1' as *conjunctions* of conditions. Similarly, one can add or delete multiple attributes from a view simultaneously using the techniques of Section 3.1 without materializing intermediate results. Adding several relations to the FROM clause follows the same pattern: the techniques of Section 3.3 can be applied for multiple added relations without materializing the intermediate results.

It is less clear, however, how to combine several changes of different types without unnecessarily materializing intermediate results. For example, is it possible to simultaneously change the SELECT clause, the FROM clause and the WHERE clause without storing intermediate relations?

If the updates are done in-place, then there is little choice but to perform the individual adaptations sequentially. However, if the adaptations are done by creating a new version of the materialized view then we have more flexibility. Note that each of the in-place updates has an alternative expression as the creating of a new version. For example,

DELETE FROM V WHERE NOT C

can be expressed as inserting into a new version of the view the result of

SELECT * FROM V WHERE C.

The critical observation is that, at the physical level, it is always possible to avoid storing an intermediate result if the intermediate result can be fully used as it is generated.

Example 12 Let us define a materialized view V by

CREATE VIEW V AS SELECT A, B, C FROM R_1 & R_2 WHERE $A > 10$.

Suppose that V is materialized. Suppose that we change the view definition by simultaneously (a) changing $A > 10$ to $A > 20$ in the WHERE clause, (b) adding a new relation R_3 to the FROM clause, with a natural join between C in the view and C as an attribute of R_3, and (c) adding a new attribute D from R_3 to the SELECT clause. The new view V' is then defined by

$$\texttt{CREATE VIEW } V' \texttt{ AS SELECT } A, B, C, D \texttt{ FROM } R_1 \ \& \ R_2 \ \& \ R_3 \texttt{ WHERE } A > 20.$$

The first change of $A > 10$ to $A > 20$ would give the result

$$\texttt{SELECT * FROM } V \texttt{ WHERE } A > 20.$$

Using the expression above, one could then express the full adaptation as

$$\texttt{SELECT } A, B, C, D \texttt{ FROM (SELECT * FROM } V \texttt{ WHERE } A > 20) \ \& \ R_3.$$

The important characteristic of this expression is that the subquery (SELECT * FROM V WHERE $A > 20$) does not have to be stored on secondary storage as an intermediate relation. The tuples satisfying the subquery could be directly pipelined into a join algorithm for joining with R_3. The join algorithm must need to make only one pass over the pipelined relation. For example, the pipelined relation could be used as the outer loop relation in a nested-loop join, but not as the inner-loop relation.

A different way of achieving the same result would be for the system to observe that

$$\texttt{SELECT } A, B, C, D \texttt{ FROM (SELECT * FROM } V \texttt{ WHERE } A > 20) \ \& \ R_3$$

can be rewritten as

$$\texttt{SELECT } A, B, C, D \texttt{ FROM } V \ \& \ R_3 \texttt{ WHERE } A > 20.$$

which it can then execute in a cost-optimal fashion. □

Given the discussion above, the question to ask of each basic technique is whether it can be applied with a single pass over the previously materialized view. If this were true of some collection of techniques, then we could cascade basic view changes by applying pipelining.

When one looks at the techniques developed earlier it turns out that, with one exception, all use of previously materialized views can be done in a single pass. The exception is the use of a previously materialized view V within an aggregation that is grouped on an attribute that is not the (physical) ordering attribute of V. Thus, for changes other than this one exception, it is possible in principle to cascade changes without materializing intermediate results.

We thus have three choices for adaptation among which the optimizer can choose: (a) applying successive in-place updates, (b) cascading the adaptations as above, or (c) recomputing the view from base relations. Even though the in-place adaptations materialize the intermediate relations, choice (a) may still be the best, since the cost of the in-place adaptation is sometimes less than the cost of scanning the whole of the old view.

7. EXPERIMENTAL VALIDATION AND EXPLANATIONS

In this section, we experimentally test some of our adaptation techniques. We consider in-place adaptation, non-in-place adaptation, and rematerialization. Note that non-in-place adaptation is essentially the query answering approach.

Our goal is to (a) validate the efficiency of the adaptation techniques, and (b) identify certain physical designs that can improve adaptation. We tested most of the adaptation techniques introduced in this article, including techniques for changes in the `SELECT` clause, `FROM` clause, `WHERE` clause (with and without aggregates), `GROUPBY` clause and `HAVING` clause. Our experiments are based on a centralized database system. However, our analysis of each test can help understand the tradeoff in other environments as well.

The rest of the section is organized as follows: In Section 7.1, we describe our experimental system. tables and Section 7.2 introduces the test sets. From Section 7.3 to Section 7.8, we present the result for each individual technique. We summarize our experiments in Section 7.9.

7.1. Description of the Environment

We implemented a system as described in Figure 1 to do all the experiments. First, the view definitions are sent to an SQL parser and transformed into some internal representations. Next, the SQL code for adaptation and for rematerialization is generated, passed to a commercial database system and executed there. The time we measured is wall clock time. (Response time, as measured by the database server, was

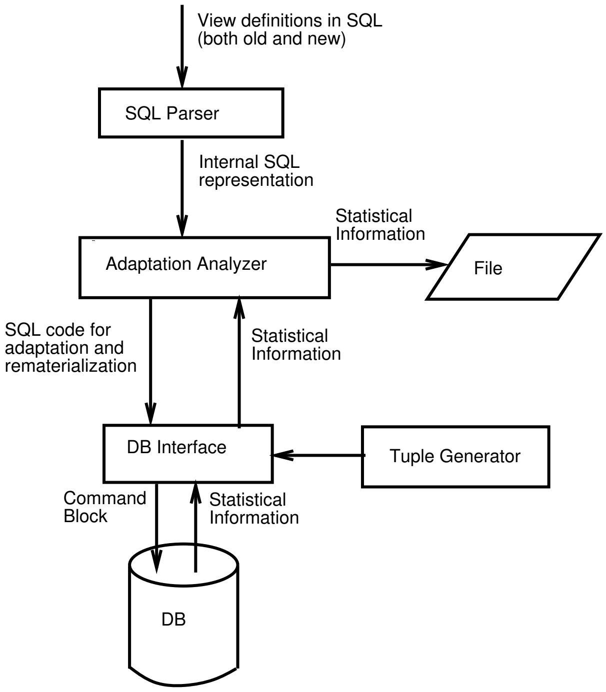


Fig. 1: Architecture of Our Experimental System

always within 1 second of the wall clock time.) To ensure the results were accurate, we flushed the memory buffer in the database before doing any adaptation or rematerialization and also force-wrote all the dirty pages.

We used two separate disks, each of size 2GB, one for data and the other for indices and log files. The database system was running on a Sun 4m 630-M140 machine with 128M of memory. The experiments were run at night when the load on the machine from other users was small. In order to get rid of transient fluctuations, we repeated the experiments two or three times and took the minimal time.

All the tables we used are from the TPC-D benchmark [TPC95]. Most of the view definitions come directly from TPC-D queries. Considering the fact that TPC-D queries are very complex, we also designed some simpler test cases.

Rematerialization is done by using the "select into" clause that doesn't perform logging. Some adaptation techniques (the in-place versions) do not require logging, but there was no easy way to turn off logging in our database. Although logging doesn't take much time when the size of an update is small, our setup actually favors rematerialization a little bit.

7.2. Design of Test Sets

We designed five test sets, each testing a group of related adaptation techniques. In the figures (Figures 2, 3, 6, 7, 9) showing those test sets, the SQL code in the box corresponds to a view definition. The label on the arrow represents the number of the adaptation technique (as listed in Appendix A) that can be used to adapt from an old view to a new view following the direction of the arrow. Enclosed in brackets are those substitution parameters that we'll change to specific values within the range. We use bold font to highlight the difference between view definitions. All the substitution parameters are randomly chosen as specified in TPC-D.

Here are the ranges of the attributes related to our test sets:

- SUPPLIER.s-acctbal [-1,000.00 ... 10,000.00]
- LINEITEM.l-shipdate [1992-01-01 ... 1998-11-31]
- LINEITEM.l-quantity [0 ... 50]
- LINEITEM.l-discount [0.00 ... 0.10]
- PARTSUPP.ps-partkey [1 ... 20,000]

- ORDER.o-orderdate [1992-01-01 ... 1998-07-31]
- ORDER.o-orderpriority [1, 2, 3, 4, 5]

All the tables are populated at scale factor 0.1 (size of the qualification database specified in TPC-D, 100M of data in total). But for the simple test sets in Figures 2 and 6, the tables we use will be too small at scale factor 0.1 and therefore are populated at scale factor 1. If not specified otherwise, all the base tables are physically ordered by their primary keys. Clustering is achieved by building a clustering index on the appropriate columns.

7.3. Changes in the `SELECT` *Clause*

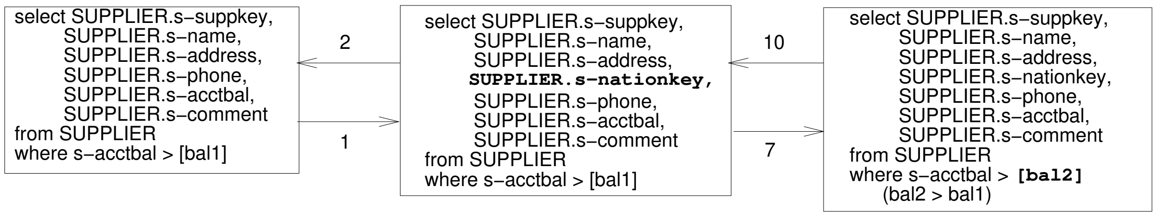


Fig. 2: Test Set 1

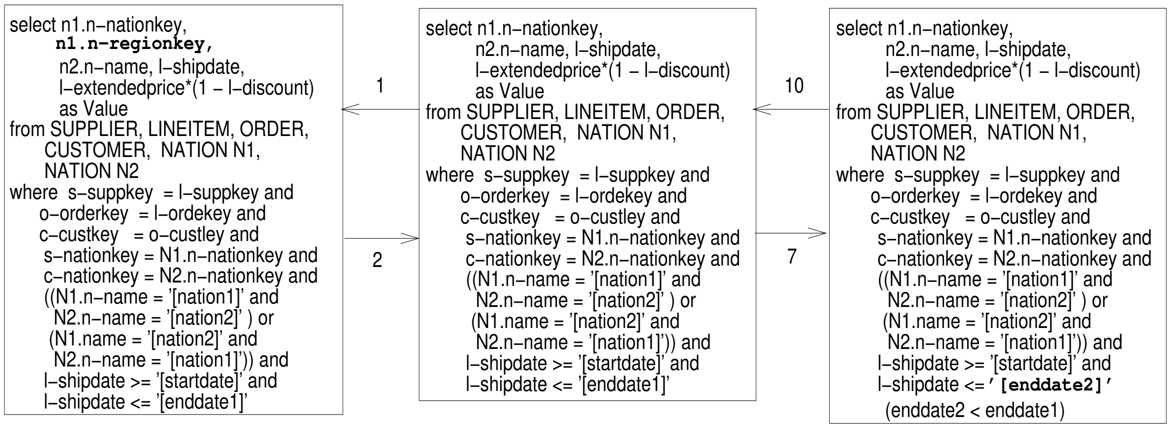


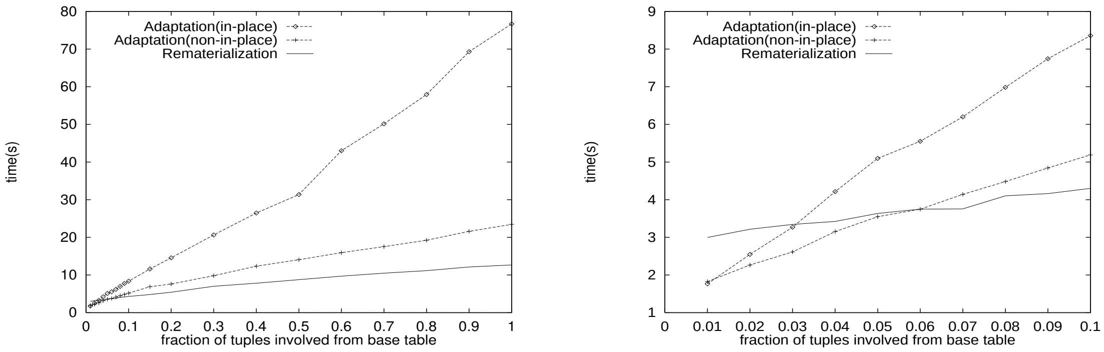
Fig. 3: Test Set 2

We use Test Set 1 and 2 (Figures 2 and 3) to measure the performance of adaptation technique 1 and 2 for handling changes in the `SELECT` clause. Graphs in Figures 4(a)-4(d) summarize our results. Figures 4(a) and 4(c) are based on Test Set 1 (labeled 1 and 2). The graphs are derived by varying the selection parameter `[bal1]` within its range. The x-axis is measured as the fraction of tuples from `SUPPLIER` that'll be included in the view. Figures 4(b) and 4(d) are based on Test Set 2 (labeled 1 and 2). `[nation1]` and `[nation2]` are chosen as 'CHINA' and 'JAPAN', respectively. `[startdate]` is chosen as '1992-01-01'.† The graphs are derived by varying `[enddate1]` from '1992-01-01' to '1998-10-01'. The x-axis is measured as the fraction of tuples from `LINEITEM` participating the join.

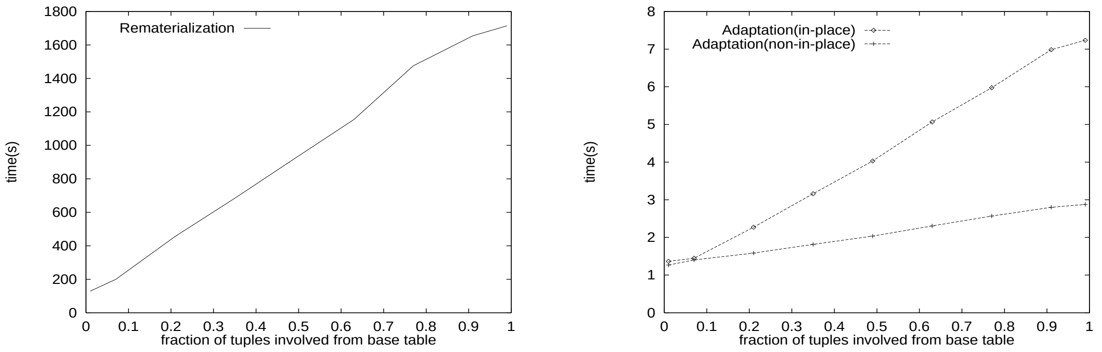
Adding a column

Figure 4(a) shows that for select-project views (no join), the adaptation technique only wins within a small range (i.e., around 3%) of tuple involvement. This is because adaptation does a look-up for each tuple in the old view. Since those tuples are randomly distributed in the pages of the base table, we need almost one page access per tuple. On the other hand, rematerialization needs just one scan through the base relation. Analytically, the crossover point can be calculated to be when the proportion of tuples involved in the view is 3.5%, which agrees with our experiment (Figure 4(a)).

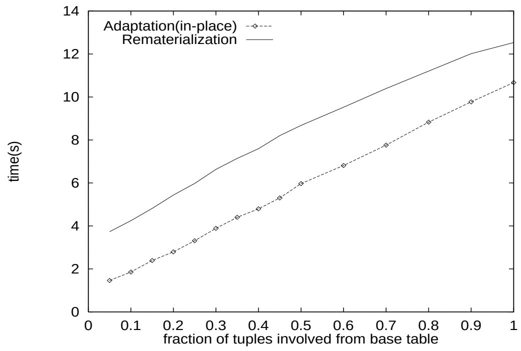
†We have chosen a different set of values and rerun the experiments. The results remain the same.



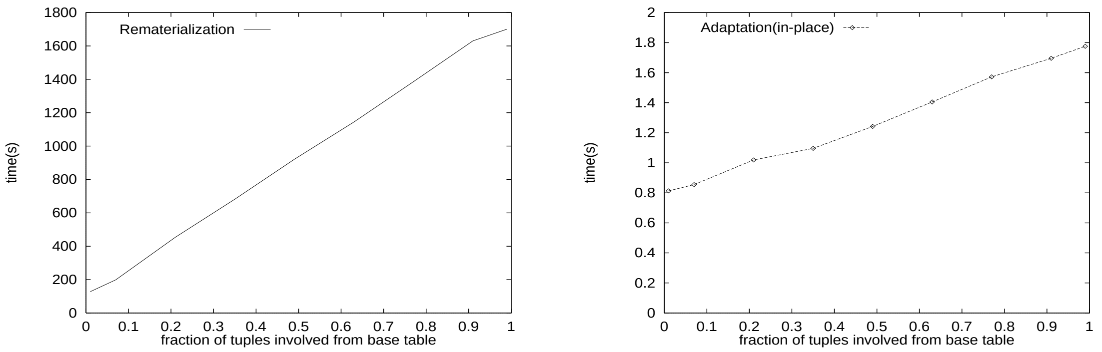
(a) Adding a column (Test Set 1, Technique 1, without join)



(b) Adding a column (Test Set 2, Technique 1, with join)



(c) Dropping a column (Test Set 1, Technique 2, without join)



(d) Dropping a column (Test Set 2, Technique 2, with join)

Fig. 4: Changes in the `SELECT` Clause

The results are strikingly different for SPJ views as shown in Figure 4(b). Adaptation wins by a wide margin over rematerialization. The reason is that rematerialization needs to do a six-way join, while adaptation only has to do look-ups from one base table. In both 4(a) and 4(b), the non-in-place version is always better than the in-place version. It seems that reorganizing a table takes more time than creating a new one. Although not shown here, the size of the base table from which the new column comes from, also plays a role. The larger the table, the longer it takes to retrieve the column.

Dropping a column

Figure 4(c) shows that adaptation outperforms rematerialization. Rematerialization needs to read in the whole base table but adaptation only has to read from the old view which is smaller.† In Figure 4(d), adaptation wins significantly because rematerialization needs to do an expensive join while adaptation just accesses the old view. The tradeoffs between in-place and non-in-place methods are similar to that of adding a column.

7.4. Changes in the `WHERE` *Clause (no aggregation)*

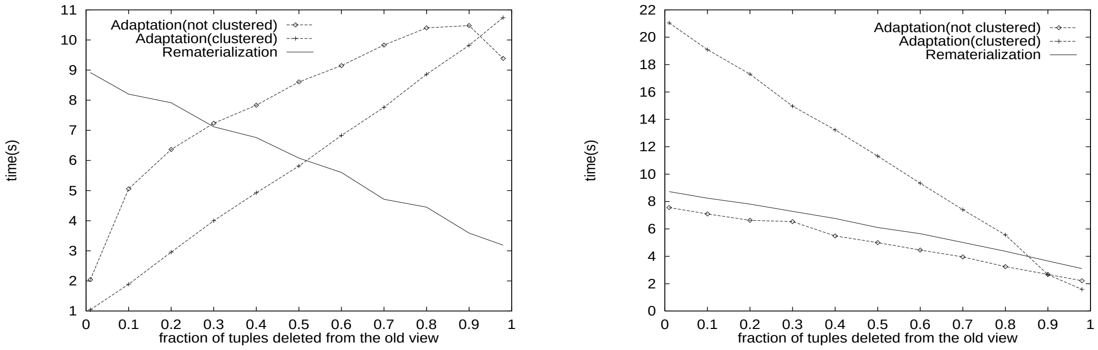
We measure the performance of adaptation techniques 7 and 10 for handling changes in the `WHERE` clause using again Test Set 1 and 2 (Figures 2 and 3). Graphs in Figures 5(a)-5(e) summarize the results. Figures 5(a) and 5(b) are based on Test Set 1 (labeled 7). The old view keeps `[bal1]` at 4500.00 and the new view varies `[bal2]` from 4500.00 to 10,000.00. Figure 5(d) is based on Test Set 1 (labeled 10). The old view keeps `[bal2]` at 4500.00 and the new view varies `[bal1]` from -1,000.00 to 4500.00. We consider no clustering and clustering on `s-acctbal` in the old view (i.e., the old view is physically ordered by `s-acctbal`). The cost of maintaining the clustered index is included in the time we measured. Figure 5(c) is based on Test Set 2 (labeled 7). The old view keeps `[enddate1]` at '1998-11-01' and the new view varies `[enddate2]` from '1992-01-01' to '1998-10-01'. Figure 5(d) is based on Test Set 2 (labeled 10). The old view keeps `[enddate2]` at '1995-06-01' and the new view varies `[enddate2]` from '1995-06-15' to '1998-10-01'.

Adding a condition

As shown in Figure 5(a), rematerialization time goes down from left to right because the higher the fraction of deletions, the smaller the size of the output. When the old view is not clustered, adaptation has to read in all the pages of the old view. Adaptation time first goes up rapidly, and then the slope decreases after the fraction of deletion reaches 0.1. The reason is that the old view is not physically ordered by `s-acctbal`, so almost all the pages of the old view will contain at least one tuple to be deleted (and thus need to be written back). There is also a drop of adaptation time after the fraction of deletion reaches 0.9. This is because that deletion percentage is now so high that some of the pages will have no data to be written back. When the old view is physically ordered by `s-acctbal`, only those pages having tuples to be deleted need to be read in, which lowers the cost of adaptation. In both cases, adaptation loses to rematerialization when the fraction of deletions is high.

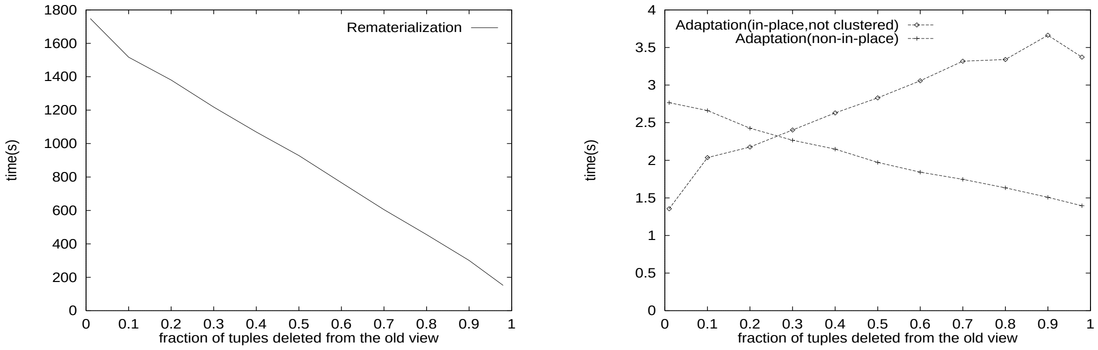
Figure 5(b) looks different from its in-place counterpart due to their different implementations. Without an index in the old view, adaptation always outperforms rematerialization. The reason is that adaptation only needs to read from the old view rather than a base table which is larger. Adaptation loses to rematerialization in the presence of a clustered index on the old view (in most cases), because the non-in-place version needs to rebuild the clustered index from scratch (the old view will be removed). The result here says that if the range of an attribute in a view definition is known to shrink quite often in the future, it is worthwhile to build a clustered index on it. Note that the clustered index is built on the materialized view, not on the base table.

†The database we used will update the data pages (not just the schema) when a column is dropped.

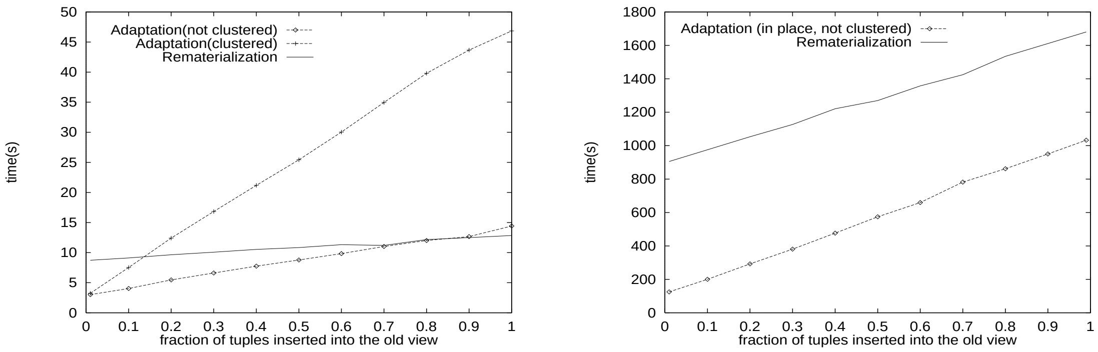


(a) in-place version (b) non-in-place version

Adding a condition (Test Set 1, Technique 7, without join)



(c) Adding a condition (Test Set 2, Technique 7, with join)



(d) without join (e) with join

Deleting a condition (Technique 10)

Fig. 5: Changes in the `WHERE` Clause (no aggregation)

Figure 5(c) shows the result when there are joins in the view. Since rematerialization needs to do a six-way join while adaptations (both the in-place and the non-in-place versions) only need to access the old view, adaptations win by a large margin. The comparison between in-place and non-in-place adaptation methods is similar to the "without join" case.

Deleting a condition

In Figure 5(d), both adaptation and rematerialization need to read from the base table. The in-place adaptation saves the time of writing out those tuples that are present in the original view into a new view. This explains why adaptation wins when there is no clustered index in the view. With a clustered index in the view, adaptation has to maintain the index. That's why adaptation time becomes longer, but it can still beat rematerialization when the fraction of tuples to be inserted is small (i.e., less than 15%). The non-in-place method first has to copy the original view to a new view and then select additional tuples from the base table. Since it has to read in more data (the original view), it loses to rematerialization. The slopes of the adaptation methods are higher than that of rematerialization since in addition to the output, the amount of logging is also proportional to the fraction of insertions.

In Figure 5(e), the time to perform the join is dominant. Although adaptation also needs to perform a six-way join as for rematerialization, it can trim down the size of one of the base tables considerably, making itself more efficient.

7.5. Changes in the `FROM` *Clause*

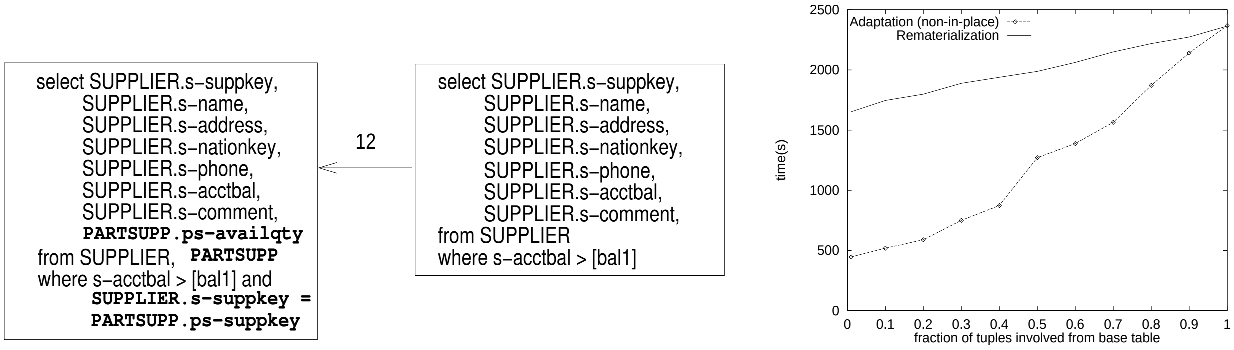


Fig. 6: Test Set 3 for Adding a table (technique 12)

We use Test Set 3 (Figure 6) to measure the performance of adaptation technique 12 for adding a table in the `FROM` clause. The result is shown in the right picture Figure 6. The graph is derived by varying `[bal1]` within its range. The x-axis is measured as the fraction of tuples selected from `SUPPLIER`.

Rematerialization needs to join two base tables while adaptation joins one base table with the old view. When the fraction of tuple involvement is low, the old view is much smaller than the base table, which makes adaptation more efficient.

It should be noted that adaptation always performs a two-way join regardless of the number of tables in the view definition. So when there are more than two tables in the new view definition, adaptation will win significantly in a larger range by doing a much cheaper join. The comparison between the in-place and non-in-place methods is similar to that in Section 7.3.

7.6. Changes in the GROUPBY Clause

We use Test Set 4 (Figure 7) to measure the performance of adaptation techniques 18 to 21 for handling changes in the GROUPBY clause. We omitted the results in this section since it's quite similar to that in Section 7.3 and 7.4.

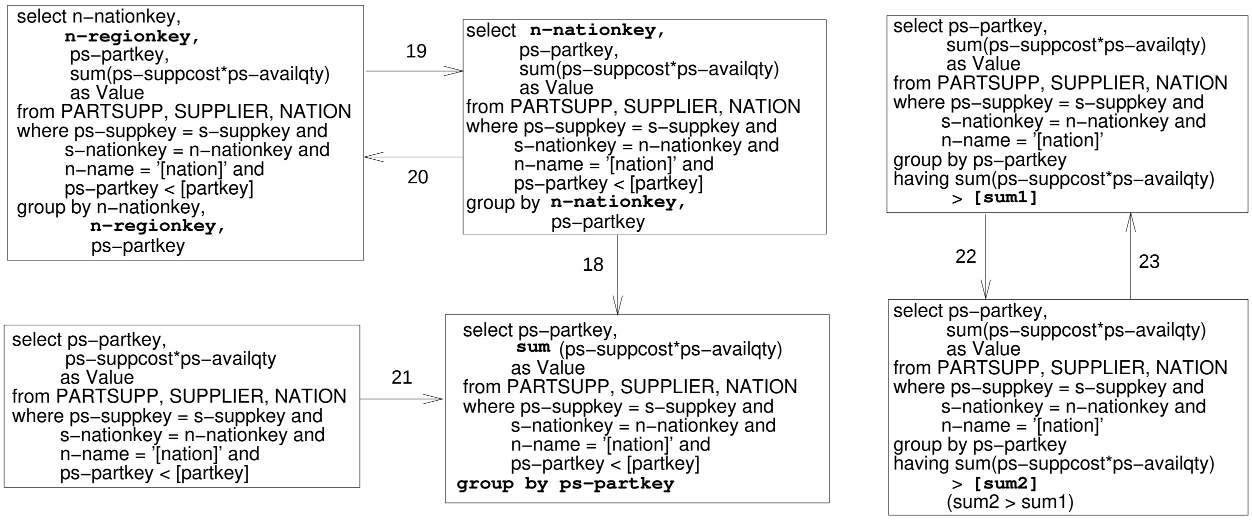
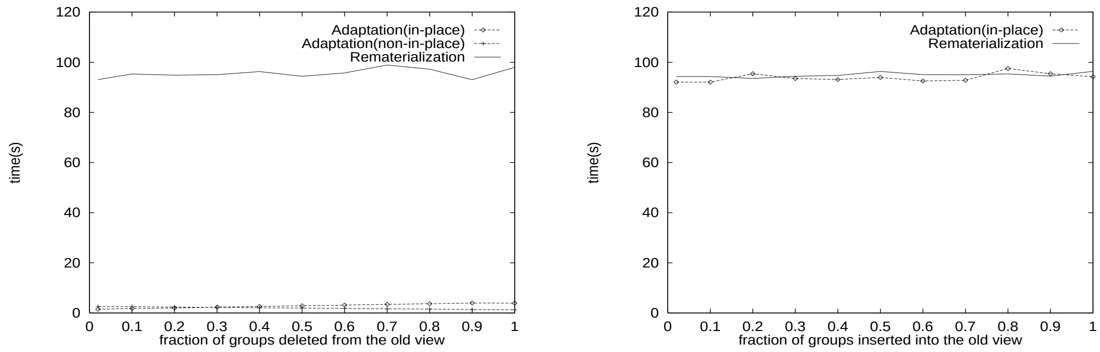


Fig. 7: Test Set 4

7.7. Changes in the HAVING Clause



(a) Adding a conjunct (technique 22) (b) deleting a conjunct (technique 23)

Fig. 8: Changes in the HAVING Clause (Test Set 4)

We also use Test Set 4 (Figure 7) to measure the performance of adaptation techniques 22 and 23 for handling changes in the HAVING clause. Graphs in Figures 8(a)-8(b) summarize the results. For Figure 8(a), [sum1] is fixed to include all the groups. [sum2] is chosen as different values to include a specific percentage of groups. The x-axis is measured as the fraction of groups deleted from the old view. For Figure 8(b), [sum2] is fixed as the value such that half of the groups will be selected. [sum1] is chosen as different values to insert a specific percentage of groups. The x-axis is measured as the fraction of groups inserted into the old view. Again, the tradeoffs between the in-place and non-in-place methods are similar to that in Section 7.4.

Figure 8(a) shows a significant win by adaptation when adding a conjunct. This is because rematerialization has to do an expensive three-way join but adaptation only needs to delete some tuples from the old view. Additionally, rematerialization needs to recompute a groupby clause which is also very expensive. But Figure 8(b) gives a totally different result for deleting a conjunct. Adaptation time is almost the same as that of rematerialization. The reason is that adaptation needs to perform the same join and groupby as rematerialization. Only the size of the output is smaller, which is less significant here. This suggests that if there are groups to be inserted, it's better to store all the groups and then select those groups needed. Although not shown here, a clustered index on the aggregate column in the old view will help (similar to Section 7.4).

7.8. Changes in the `WHERE` *Clause (in the presence of aggregates)*

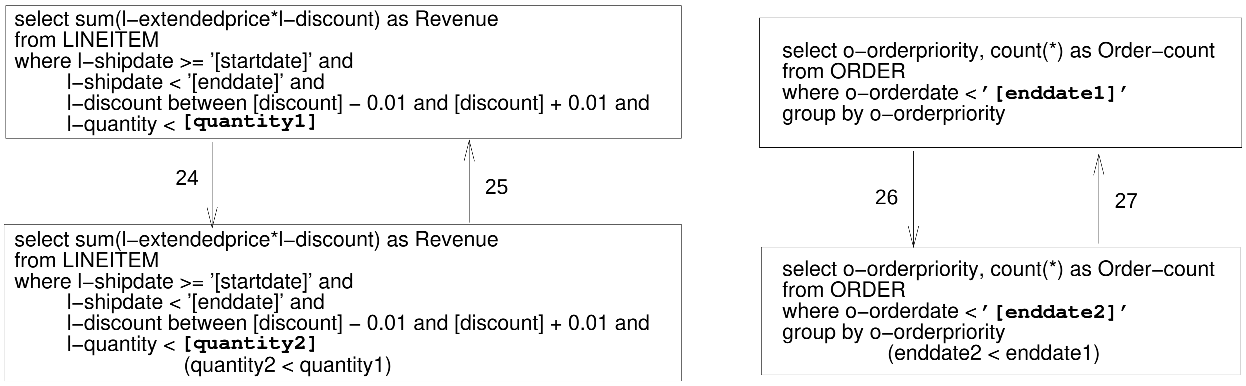
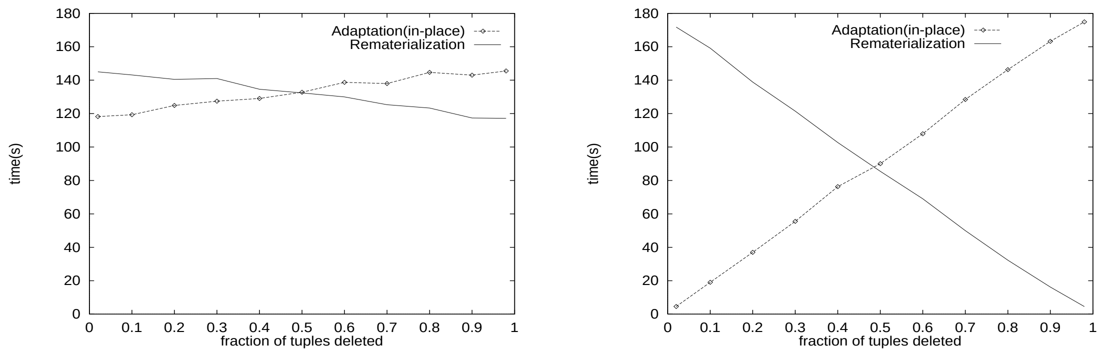
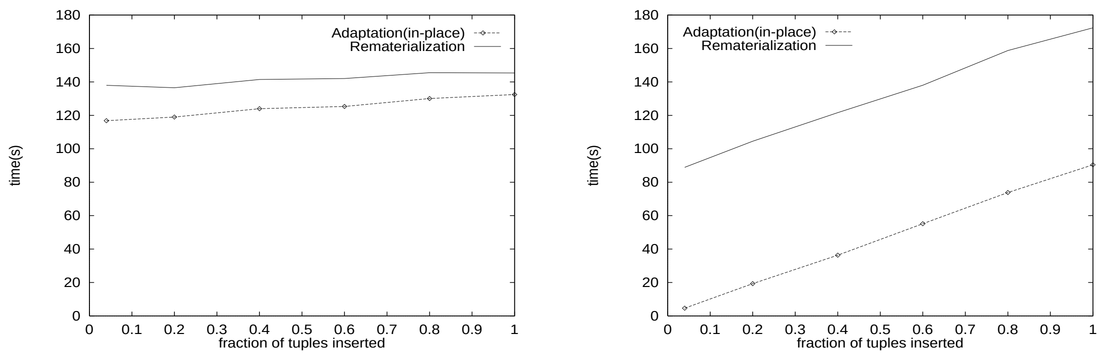


Fig. 9: Test Set 5



(a) clustered index on key (b) clustered index on `l-quantity`

Adding a condition (technique 24)



(c) clustered index on key (d) clustered index on `l-quantity`

Dropping a condition (technique 25)

Fig. 10: Changes in the `WHERE` Clause (in the presence of scalar aggregates)

We use Test Set 5 (Figure 9) to measure the performance of adaptation techniques 24 to 27 for handling changes in the `WHERE` clause in the presence of aggregates. Graphs in Figures 10(a)-10(d) summarize the results when there are only scalar aggregates in the view definition (techniques 24 and 25). Graphs in

Figures 11(a)-11(d) and Figures 12(a)-12(d) show the results when there exists a `GROUPBY` clause in the view definition (techniques 26 and 27). In this section, all the adaptation methods are in-place and there are no corresponding non-in-place methods.

Scalar Aggregates

For all graphs in Figure 10, `[startdate]` and `[enddate]` are chosen as '1992-01-01' and '1998-11-01' respectively. `[discount]` is chosen as 0.03. For Figures 10(a) and 10(b), `[quantity1]` is fixed at 50 (maximal value) and `[quantity2]` varies from 0 to 49. For Figures 10(c) and 10(d), `[quantity2]` is fixed at 25 and `[quantity1]` varies from 26 to 49. For Figures 10(b) and 10(d), a clustered index is built on `l-quantity` in base table `LINEITEM` (i.e., `LINEITEM` is physically ordered by `l-quantity`). The x-axis is measured as the ratio of the number of tuples satisfying the where clause in the new view but not in the old view, and the number of tuples satisfying the where clause in the old view.

The graphs here are different from those in Section 7.4 since we have only one tuple—the aggregation results in the old view. So, while some techniques in Section 7.4 only need to delete some tuples from the old view, all the techniques in this section have to access the base table.

In Figure 10(a), adaptation also needs to read in all the pages of the base table as rematerialization because the base table is not physically ordered on `l-quantity`. When the deletion percentage is low, the computation time for adaptation is less than for rematerialization and when deletion percentage is high, the opposite. But the computation time here is less significant than the I/O time, which explains why the difference is small. In Figure 10(b), both adaptation and rematerialization can take advantage of the clustered index in the base table. When deletion percentage is low, adaptation saves a lot of I/O time. So does rematerialization when deletion percentage is high. In Figure 10(c), adaptation always beats rematerialization by a small fraction since it saves the time to recompute the aggregation results already in the old view. Figure 10(d) shows that the gap between the two lines is much wider. The reason is that with the help of the clustered index, adaptation also saves the time of reading in those tuples in order to compute the aggregation results already in the old view.

Although not shown here, we can expect similar results when there are joins in the view definition. Adaptation techniques need to perform the same join as rematerialization, but one of the base tables (the one containing the attribute mentioned by the changing predicate) can be restricted to include only the difference.

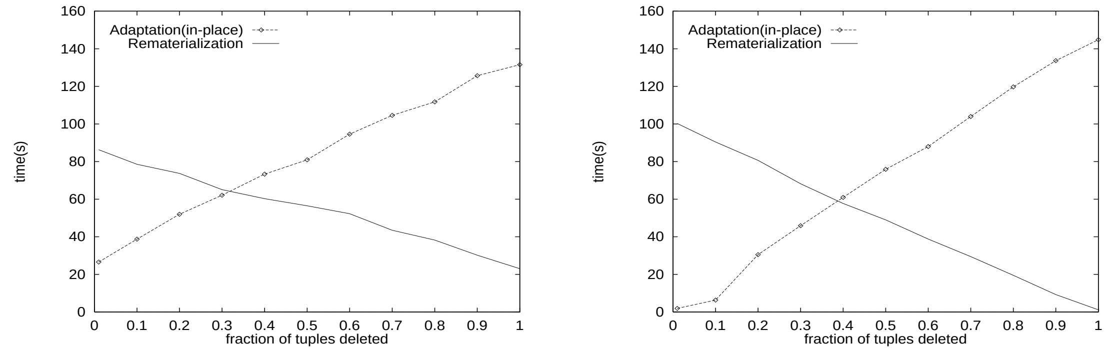
With Groupby

For Figures 11(a) and 11(b), `[enddate1]` is fixed at '1998-08-01' and `[enddate2]` varies from '1992-01-01' to '1998-07-01'. For Figures 11(c) and 11(d), `[enddate2]` is fixed at '1995-04-01' and `[enddate1]` varies from '1995-08-01' to '1998-08-01'. For Figures 11(b) and 11(d), a clustered index is built on `o-orderdate` in base table `ORDER` (i.e. `ORDER` is physically ordered by `o-orderdate`). The x-axis is measured in the same way as in Figures 10. There are only five groups in the view.

The adaptation techniques in this section also need to access the base table. In Figure 11(a), both adaptation and rematerialization need to read in the whole base table. The computation time for adaptation is less than that for rematerialization when deletion percentage is low. In Figure 11(b), since the base table has special physical order, only the needed pages will be read in by both adaptation and rematerialization. Adaptation wins within a wider range. In Figure 11(c), both subqueries in the adaptation technique need to access the base table. Although the two access patterns are quite similar, the optimizer didn't recognize it. The base table has to be read in twice (since the whole table is too large to be held in RAM completely), resulting in the poor performance of adaptation. In Figure 11(d), for the same reason, adaptation loses in most cases. But adaptation still wins when the fraction of insertions is below 0.2. The reason is that by having the clustered index on the base table, only the needed pages will be read in. When those pages can still fit into the memory, the second subquery can utilize it without accessing the disk. We would expect better performance with the help of a multiquery optimizer.

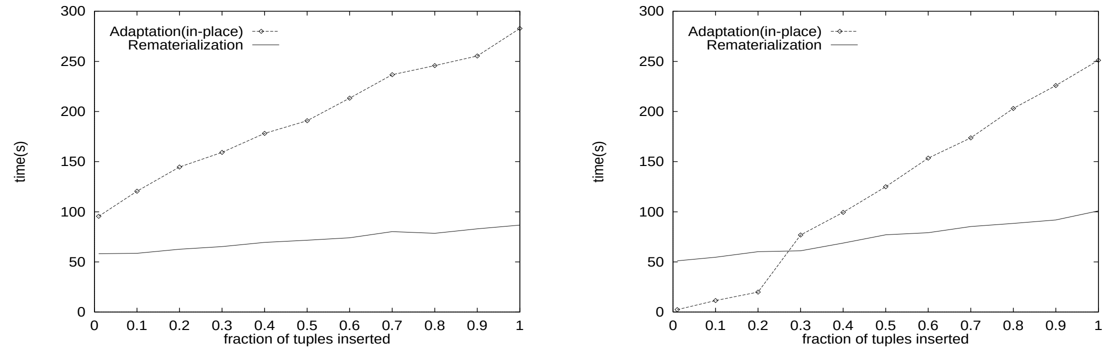
Additionally, the adaptation techniques here are sensitive to the number of groups involved in the view. We repeated the test on a different groupby attribute—`O-CUSTKEY` (15K distinct values) and obtained the results in Figure 12. The result suggests that the adaptation techniques here are not suitable for a large number of groups. The reason is that too many groups need to be updated.

The situation when there are joins in the view definition is similar to that of scalar aggregates.



(a) clustered index on key (b) clustered index on `o-orderdate`

Adding a condition (technique 26)



(c) clustered index on key (d) clustered index on `o-orderdate`

Deleting a condition (technique 27)

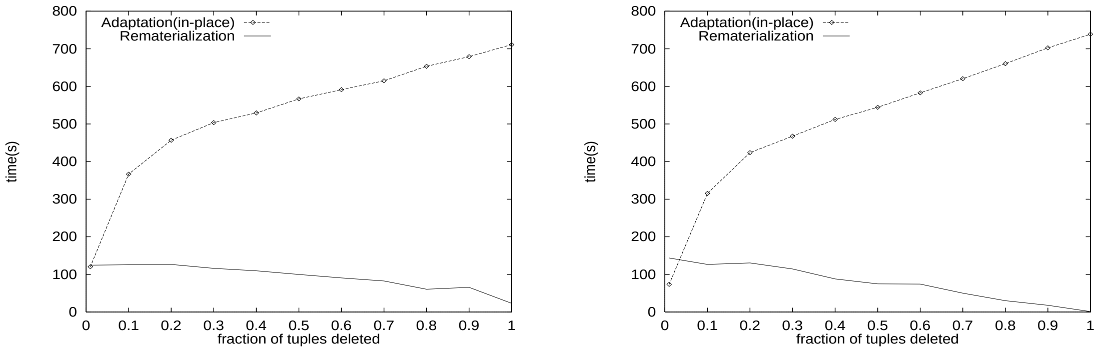
Fig. 11: Changes in the `WHERE` Clause (in the presence of the groupby clause, 5 groups)

7.9. Summary: Experiments

What do we learn from the experiments? First, although we do find out that adaptation is generally useful, the “common wisdom” that adaptation will always win is not true. In cases when we want to add new groups into the old view (by weakening the conjunct in the `HAVING` clause), the adaptation technique is not helpful. Also, when there are too many groups in the old view, adaptation techniques handling changes in the `WHERE` clause perform worse than rematerialization.

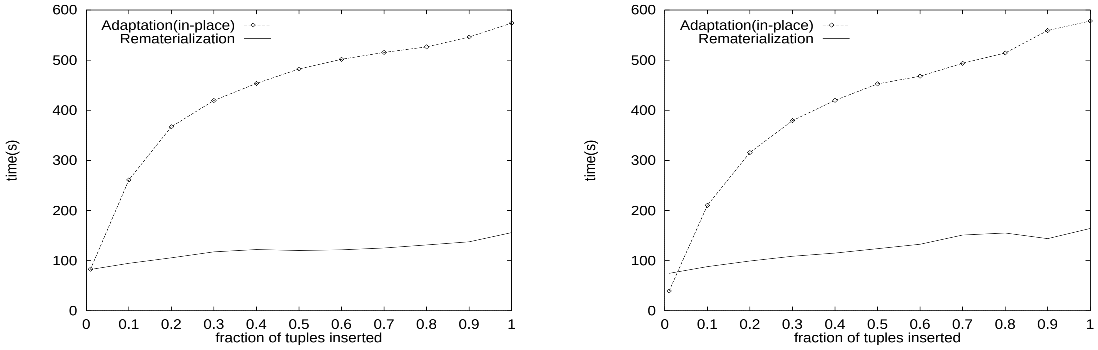
First we summarize the comparison between adaptation and rematerialization. There are several factors that can impact the efficiency of various adaptation techniques. We have seen that adaptation performs significantly better than rematerialization when there are joins in the view. Adaptation techniques usually only need to access the old view itself or do look-ups in one of the base tables. Rematerialization, on the other hand, always has to do a complex multi-way join. Adaptation saves both the I/O time and the computation time. This is encouraging since it means the more complex the view, the larger the benefit of using adaptation. Clustering is also very important, especially for single table views. Clustering on both the base table and the materialized view itself can be helpful. The intuition here is that appropriate clustering allows adaptation techniques to access only the necessary portion of the underlying data. We can expect a lot of savings in I/O time and in certain cases, the computation time also. While rematerialization can also benefit from the clustering on the base table (to a smaller extent though), it certainly can’t exploit the clustering on the materialized view. Selectivity in the original view is another factor. The higher the selectivity, the smaller the size of the view will be. Since most of the adaptation techniques have to access the old view whereas rematerialization has to access the base table, adaptation does better for more restrictive views. Finally, we also need to pay attention to the change itself. Small changes usually mean that fewer tuples need to be processed and thus adaptation is more desirable.

Next, we summarize the tradeoffs between in-place and non-in-place methods (when both exist for the same technique). There are basically two cases that a view is updated. In the first case, the view is changed by either adding or dropping a column. In-place methods shows no advantages over the non-in-place ones



(a) clustered index on key (b) clustered index on `o-orderdate`

Adding a condition (technique 26)



(c) cluster index on key (d) clustered index on `o-orderdate`

Deleting a condition (technique 27)

Fig. 12: Changes in the `WHERE` Clause (in the presence of the groupby clause, 15K groups)

in our experiments. However, the results could be different under a different system where tables are stored vertically. In the second case, the view is changed by either inserting or deleting some rows. In-place methods are usually more efficient than non-in-place methods, especially when the change is small. Additionally, in-place methods allow us to cluster the view. Clustering improves the in-place method further when the view is shrinking.

Here are the high-level conclusions we have reached:

- Adaptation can help in many cases.
- Adaptation is more efficient when
  - there are joins in the view definition.
  - there exists an appropriate physical order on the view.
  - there exists an appropriate physical order on the base table.
  - the number of groups in the view (with aggregates) is not too large.
  - the changes in the view definition are small.

Due to the limitation of space here, we omitted trivial adaptation techniques (such as remove `DISTINCT`) and techniques that have similar syntax to those we have tested. We also omitted the techniques in the presence of `UNION` and `EXCEPT` , since they are the combination of other techniques.

8. CONCLUSIONS

When the definition of a materialized view changes we need to bring the materialization up-to-date. In this article we focus on *adapting* a materialized view, i.e., using the old materialization to help in the materialization of the new view. The alternative to adaptation is to *recompute* the view from scratch, making no use of the old materialization. Often, it can be more efficient by an order of magnitude to adapt a view rather than recompute it.

A number of applications, like data-archaeology and visualization, require *interactive*, and thus quick, response to changes in the definition of a materialized view. Views defined over autonomous Internet sources tend to change their definitions. Our techniques are useful for adapting views in such applications.

We have provided a *comprehensive list* of view adaptation techniques that can be applied for basic view definition changes. Each of these adaptation techniques is itself expressed as an `SQL` query or update that makes use of the old materialization. Because the adaptation is itself expressed in `SQL`, it is possible for the query optimizer to estimate the cost of these techniques using standard cost-based optimization. In some cases there may be several adaptation alternatives, and each of the alternatives would be considered in turn.

Our basic adaptation techniques correspond to local changes in the view definition. We also describe how multiple local changes can be combined to give an adaptation technique for changes to several parts of a view definition. Some techniques for adapting a view in response to a local change can be *pipelined* thereby eliminating the need to store intermediate adapted views when multiple local changes are combined.

Often it is easier to adapt a view if certain additional information is kept in the view. Such additional information includes keys of base relations, attributes involved in selection conditions, counts of the number of derivations of each tuple, additional aggregate functions beyond those requested, and identifiers indicating which subquery in a union each tuple came from. Depending on the type of anticipated change, the view can be defined to contain the appropriate additional information. Additionally, it can be beneficial to reserve some physical space in each record to allow in-place adaptation involving addition of attributes.

We have derived tables of adaptation techniques (see Appendix A for a complete list) that can be used in three important ways. Firstly, the query optimizer can use the tables to find the adaptation technique (and compute its cost estimate) given the properties of the current schema vis-a-vis the assumptions stated in the table. Secondly, a database administrator or user can use the tables to see what assumptions would need to be satisfied in order to make view adaptation possible at the most efficient level, and define the view accordingly. Thirdly, the database administrator can interact with the query optimizer to build appropriate access methods and indexes on the base relations and on the materialized views, in order to facilitate efficient adaptation.

We have implemented a view adaptation prototype on top of a commercial relational database system and measured the relative speeds of adaptation versus rematerialization. The results give strong support for most of the adaptation techniques.

The main contributions of this article are (a) the derivation of a comprehensive set of view adaptation techniques, (b) the identification of guidelines that can be provided to users and database administrators in order to facilitate view adaptation, and (c) the experimental validation of quantitative improvements under a variety of conditions.

ACKNOWLEDGMENTS

We thank Arun Netravali for pointing out the importance of redefinition to data visualization, and Shaul Dar and Tom Funkhouser for discussions of the relationship between view maintenance and data visualization. Justin Vallon wrote much of the code used in the view adaptation experiments.

REFERENCES

[AWS93] Christopher Ahlberg, Christopher Williamson, and Ben Shneiderman. Dynamic Queries for information exploration: an implementation and evaluation. In Ben Shneiderman, editor, *Sparks of Innovation in Human-Computer Interaction*. Ablex Publishing Corp, 1993.

[BBMR89] Alex Borgida, et al. CLASSIC: A structural data model for objects. In *ACM-SIGMOD*, pages 59–67, June 1989.

[Bel00] Zohra Bellahsene. Adapting Materialized Views after Redefinition in Distributed Environments. In *International Conference on Conceptual Modeling*, 2000.

[BLT86] J. Blakeley, P. Larson and F. Tompa. Efficiently updating materialized views. In *ACM-SIGMOD*, 1986.

[BST+92] Ronald J. Brachman, et al. Knowledge representation support for data archaeology. In *First International Conference on Information and Knowledge Management*, pages 457–464, November 1992.

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

[CKLMR97] L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Supporting Multiple View Maintenance Policies. In *SIGMOD*, pages 405-416, 1997.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of International Conference on Data Engineering*, 1995.

[CR94] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: integrating query result caching and matching. In *EDBT*, 1994.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[Fre95] Clark D. French. “One size fits all” database architectures do not work for DDS. In *Proceedings of the ACM SIGMOD Conference*, pages 449–450, 1995.

[Fre97] Clark D. French. Teaching an OLTP database kernel advanced data warehousing techniques. In *Proc. IEEE Int'l Conf. on Data Eng.*, pages 194–198, 1997.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-Query Processing in Data Warehousing Environments. In *VLDB*, 1995.

[GL95] T. Griffin and L. Libkin. Increamental maintenance of views with duplicates. In *SIGMOD*, pages 211-222, 1995.

[GMR95] Ashish Gupta, Inderpal Singh Mumick, and Kenneth A. Ross. Adapting materialized views after redefinitions. In *SIGMOD*, pages 211-222, 1995.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–167, 1993.

[GSUW94] Ashish Gupta, Yehoshua Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Constraint Checking with Partial Information. In *PODS*, pages 45–55, 1994.

[Han87] Eric N. Hanson A performance analysis of view materialization strategies. In *SIGMOD*, pages 440-453, 1987.

[KLMQR97] A. Kawaguchi, D. Lieuwen, I. Mumick, D. Quass, and K. Ross. Concurrency Control for Materialized Views. In *ICDT*, pages 306-320, 1997.

[LMS94] Alon Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate movearound. In Bocca et al. *VLDB*, pages 96–107, 1994.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, 1995.

[LY85] P. A. Larson and H.Z. Yang. Computing queries from derived relations. In *VLDB* , pages 259–269, 1985.

[MPR90] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, 1990.

[OJ93] Vicki L. O'Day and Robin Jeffries. Orienteering in an information landscape: How information seekers get from here to there. In *ACM-INTERCHI*, pages 438–445, April 1993.

[RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[SDJY96] Divesh Srivastava, Shaul Dar, H. V. Jagadish and Alon Y. Levy. Answering querries with aggregation using views. In *VLDB*, 1996.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh and S. Potamianos. On rules, procedures, caching and views in database systems. In *SIGMOD*, 1990.

[Syb97b] Sybase Corporation. *Sybase IQ 11.2.1*, 1997.

[TPC95] TPC-D Benchmark Standard Specification (Revision 1.0), May, 1995.

[TSI94] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. In Bocca et al. *VLDB*, pages 367–378, 1994.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 2*. Computer Science Press, 1989.

[WS93] Christopher Williamson and Ben 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.

[YL87] H. Z. Yang and P. A. Larson. Query transformation for PSJ-queries. In *VLDB*, pages 245–254, 1987.

[YL95] Weipeng P. Yan and Per-Ake Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21th VLDB Conference*, pages 345–357, 1995.

Recommended by Patrick O'Neil, coeditor.

A. TABLES OF ADAPTATION TECHNIQUES

In this section we present the complete tables of adaptation techniques. The initial view for each table is described in the corresponding section of the text (Section 3.5 for Table 4 and Table 5, Section 4.6 for Table 7 and Table 8, and Section 5.3 for Table 10). The redefined view shows what the view looks like after the redefinition. The adaptation technique will either update the old materialization V, or insert tuples into a relation called New_V (non-in-place methods) which represents the new materialization. In the event that basic adaptations are pipelined, the tuples may not actually be stored in an intermediate relation. To avoid redundancy, for the same technique we list both the in-place and the non-in-place method only in Table 4.

We omit symmetric cases such as for the two arguments of unions.

No.	Redefined View	Adaptation Technique	Assumptions
1	SELECT $A, A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_1 AND $\cdots$ AND C_k	in-place: ALTER TABLE V ADD A UPDATE V SET A = (SELECT A FROM S WHERE $S.K = V.K$) non-in-place: INSERT INTO New_V SELECT $A, A_1, \ldots, A_n$ FROM V, S WHERE $S.K = V.K$	(1)
2	SELECT $A_2, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_1 AND $\cdots$ AND C_k	in-place: ALTER TABLE V DROP A_1 non-in-place: INSERT INTO New_V SELECT $A_2, \ldots, A_n$ FROM V	
3	SELECT DISTINCT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_1 AND $\cdots$ AND C_k	INSERT INTO New_V SELECT DISTINCT * FROM V	
4	SELECT DISTINCT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_1 AND $\cdots$ AND C_k	Mark view as being distinct.	(3)
5	Remove a DISTINCT qualifier.	INSERT INTO New_V SELECT $A_1, \ldots, A_n$ FROM V, $R_i, \ldots, R_j, R_{j+1}, \ldots, R_m$ [C] WHERE C_p AND $\cdots$ AND C_k	(2)
6	Remove a DISTINCT qualifier.	Mark view as having duplicates.	(3)
7	SELECT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_0 AND C_1 AND $\cdots$ AND C_k	in-place: DELETE FROM V WHERE NOT C_0 non-in-place: INSERT INTO New_V SELECT $A_1, \ldots, A_n$ FROM V WHERE NOT C_0	(4)
8	SELECT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_1' AND $\cdots$ AND C_k	DELETE FROM V WHERE NOT C_1'	$C_1' \Rightarrow C_1$, (4)
9	SELECT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_1' AND $\cdots$ AND C_k	DELETE FROM V WHERE NOT C_1' INSERT INTO V SELECT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_1' AND NOT C_1 AND $\cdots$ AND C_k	$C_1' \not\Rightarrow C_1$, (4)
10	SELECT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_2 AND $\cdots$ AND C_k	in-place: INSERT INTO V SELECT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE NOT C_1 AND C_2 AND $\cdots$ AND C_k non-in-place: INSERT INTO New_V (SELECT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE NOT C_1 AND C_2 AND $\cdots$ AND C_k UNION SELECT $A_1, \ldots, A_n$ FROM V)	

Table 4: Adaptation Techniques for SELECT-FROM-WHERE Views

No.	Redefined View	Adaptation Technique	Assumptions
11	SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$ FROM R_1 & $\cdots$ & R_m & R_{m+1} WHERE C_1 AND $\cdots$ AND C_k	ALTER TABLE V ADD $D_1, \ldots, D_j$ UPDATE V SET $D_1, \ldots, D_j$ = (SELECT $R_{m+1}.D_1, \ldots, R_{m+1}.D_j$ FROM R_{m+1} WHERE $R_{m+1}.A = V.B$).	(5,6,7)
12	SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$ FROM R_1 & $\cdots$ & R_m & R_{m+1} WHERE C_1 AND $\cdots$ AND C_k	INSERT INTO New_V SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$ FROM V, R_{m+1} WHERE $A = B$	(5,6)
13	SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$ FROM R_1 & $\cdots$ & R_m & R_{m+1} WHERE C_1 AND $\cdots$ AND C_k	ALTER TABLE V ADD $D_1, \ldots, D_j$ UPDATE V SET $D_1, \ldots, D_j$ = (SELECT $R_{m+1}.D_1, \ldots, R_{m+1}.D_j$ FROM R_{m+1}, R_i WHERE $R_{m+1}.A = R_i.B$ AND $V.K = R_i.K$).	(5,7,8)
14	SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$ FROM R_1 & $\cdots$ & R_m & R_{m+1} WHERE C_1 AND $\cdots$ AND C_k	INSERT INTO New_V SELECT $A_1, \ldots, A_n, D_1, \ldots, D_j$ FROM V, R_i, R_{m+1} WHERE $A = B$ AND $V.K = R_i.K$	(5,8)
15	SELECT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_{m-1} WHERE C_1 AND $\cdots$ AND C_k	No adaptation needed.	(9,10)
16	SELECT $A_1, \ldots, A_j$ FROM R_1 & $\cdots$ & R_{m-1} WHERE C_1 AND $\cdots$ AND C_k	ALTER TABLE V DROP $A_{j+1}, \ldots, A_n$	$j < n$, (9,10)

Table 5: Adaptation Techniques for SELECT-FROM-WHERE Views, Continuted

1. Attribute A is from relation S and the key K for S is in view V.
2. The view contains keys for $R_1, \ldots, R_j$, $C_p, \ldots, C_k$ and C are the join conditions relating attributes of $R_{j+1}, \ldots, R_m$ to each other and to $R_1, \ldots, R_j$, and $R_i, \ldots, R_j$ are those relations in $R_1, \ldots, R_j$ that have an attribute both mentioned by $C_p, \ldots, C_k$ or C and not in $A_1, \ldots, A_n$.
3. An augmented view that keeps a count of number of derivations of each tuple is used.
4. Attribute of condition is either an attribute of the view, or of a wider augmented stored view.
5. $D_1, \ldots, D_j$ and A are attributes of R_{m+1}, and the join condition is $A = B$.
6. B is an attribute of V.
7. A is a key for relation R_{m+1}.
8. B is an attribute of R_i, K is a key of R_i, and K is an attribute of V.
9. Join with R_m is on a key of R_m.
10. Either V contains a SELECT DISTINCT, or the join of R_m is on a key attribute.

Table 6: Assumptions for the Adaptation Techniques in Table 4 and Table 5

No.	Redefined View	Adaptation Technique	Assumptions
17	CREATE VIEW V AS SELECT $A_1, \ldots, A_n,\ F_1(B_1), \ldots, F_j(B_j)$ FROM $R_1\ \&\ \cdots\ \&\ R_m$ WHERE C_1 AND $\cdots$ AND C_k GROUPBY $A_1, \ldots, A_{p-1}$	INSERT INTO New_V SELECT $A_1, \ldots, A_n,\ G_1(E_1), \ldots, G_j(E_j)$ FROM V GROUPBY $A_1, \ldots, A_{p-1}$	(1)
18	CREATE VIEW V AS SELECT $A_2, \ldots, A_n,\ F_1(B_1), \ldots, F_j(B_j)$ FROM $R_1\ \&\ \cdots\ \&\ R_m$ WHERE C_1 AND $\cdots$ AND C_k GROUPBY $A_2, \ldots, A_p$	INSERT INTO New_V SELECT $A_2, \ldots, A_n,\ G_1(E_1), \ldots, G_j(E_j)$ FROM V GROUPBY $A_2, \ldots, A_p$	(1)
19	CREATE VIEW V AS SELECT $A_1, \ldots, A_{n-1},\ F_1(B_1), \ldots, F_j(B_j)$ FROM $R_1\ \&\ \cdots\ \&\ R_m$ WHERE C_1 AND $\cdots$ AND C_k GROUPBY $A_1, \ldots, A_{p-1}$	ALTER TABLE V DROP A_n	(2)
20	CREATE VIEW V AS SELECT $A_1, \ldots, A_n, A_{p+1},\ F_1(B_1), \ldots, F_j(B_j)$ FROM $R_1\ \&\ \cdots\ \&\ R_m$ WHERE C_1 AND $\cdots$ AND C_k GROUPBY $A_1, \ldots, A_p, A_{p+1}$	ALTER TABLE V ADD A_{p+1} UPDATE V SET $A_{p+1} =$ (SELECT $R_i.A_{p+1}$ FROM R_i WHERE $R_i.A_j = V.A_j$)	(3)
21	CREATE VIEW V AS SELECT $A_1, \ldots, A_n, G_1(D_1), \ldots, G_q(D_q)$ FROM $R_1\ \&\ \cdots\ \&\ R_m$ WHERE C_1 AND $\cdots$ AND C_k GROUPBY $A_1, \ldots, A_r$	INSERT INTO New_V SELECT $A_1, \ldots, A_s, G_1(D_1), \ldots, G_q(D_q)$ FROM V GROUPBY $A_1, \ldots, A_r$	(4)
22	CREATE VIEW V AS SELECT $A_1, \ldots, A_n, G_1(D_1), \ldots, G_q(D_q)$ FROM $R_1\ \&\ \cdots\ \&\ R_m$ WHERE C_1 AND $\cdots$ AND C_k GROUPBY $A_1, \ldots, A_r$ HAVING H_0 AND H_1 AND $\cdots$ AND H_p	DELETE FROM V WHERE NOT H_0	(5)
23	CREATE VIEW V AS SELECT $A_1, \ldots, A_n, G_1(D_1), \ldots, G_q(D_q)$ FROM $R_1\ \&\ \cdots\ \&\ R_m$ WHERE C_1 AND $\cdots$ AND C_k GROUPBY $A_1, \ldots, A_r$ HAVING H_2 AND $\cdots$ AND H_p	INSERT INTO V SELECT $A_1, \ldots, A_n, G_1(D_1), \ldots, G_q(D_q)$ FROM $R_1\ \&\ \cdots\ \&\ R_m$ WHERE C_1 AND $\cdots$ AND C_k GROUPBY $A_1, \ldots, A_r$ HAVING NOT H_1 AND H_2 AND $\cdots$ AND H_p	(5)

Table 7: Adaptation Techniques for Aggregate Views

No	Redefined View	Adaptation Technique	Assumptions
24	CREATE VIEW V $(M_1, \ldots, M_n)$ AS SELECT $F_1(A_1), \ldots, F_n(A_n)$ FROM R_1 & $\cdots$ & R_m WHERE C_0 AND C_1 AND $\cdots$ AND C_k	UPDATE V SET $M_i =$ $H_i(M_i,$ SELECT $F_i(A_i)$ FROM R_1 & $\cdots$ & R_m WHERE NOT C_0 AND C_1 AND $\cdots$ AND C_k) WHERE EXISTS (SELECT * FROM R_1 & $\cdots$ & R_m WHERE NOT C_0 AND C_1 AND $\cdots$ AND C_k)	(6)
25	CREATE VIEW V $(M_1, \ldots, M_n)$ AS SELECT $F_1(A_1), \ldots, F_n(A_n)$ FROM R_1 & $\cdots$ & R_m WHERE C_2 AND $\cdots$ AND C_k	UPDATE V SET $M_i =$ $H'_i(M_i,$ SELECT $F_i(A_i)$ FROM R_1 & $\cdots$ & R_m WHERE NOT C_1 AND C_2 AND $\cdots$ AND C_k) WHERE EXISTS (SELECT * FROM R_1 & $\cdots$ & R_m WHERE NOT C_1 AND C_2 AND $\cdots$ AND C_k)	(6)
26	CREATE VIEW $V(A_1, \ldots, A_n, M_1, \ldots, M_j)$ AS SELECT $A_1, \ldots, A_n$, $F_1(B_1), \ldots, F_j(B_j)$ FROM R_1 & $\cdots$ & R_m WHERE C_0 AND C_1 AND $\cdots$ AND C_k GROUPBY $A_1, \ldots, A_n$	UPDATE V SET $M_i =$ $H_i(M_i,$ SELECT $F_i(B_i)$ FROM R_1 & $\cdots$ & R_m WHERE NOT C_0 AND C_1 AND $\cdots$ AND C_k AND $A_1 = V.A_1$ AND $\cdots$ $A_n = V.A_n$) WHERE EXISTS (SELECT * FROM R_1 & $\cdots$ & R_m WHERE NOT C_0 AND C_1 AND $\cdots$ AND C_k AND $A_1 = V.A_1$ AND $\cdots$ $A_n = V.A_n$) DELETE V WHERE $M_m = 0$	(6, 7)
27	CREATE VIEW $V(A_1, \ldots, A_n, M_1, \ldots, M_j)$ AS SELECT $A_1, \ldots, A_n$, $F_1(B_1), \ldots, F_j(B_j)$ FROM R_1 & $\cdots$ & R_m WHERE C_2 AND $\cdots$ AND C_k GROUPBY $A_1, \ldots, A_n$	UPDATE V SET $M_i = H'_i(M_i,$ SELECT $F_i(B_i)$ FROM R_1 & $\cdots$ & R_m WHERE NOT C_0 AND C_1 AND $\cdots$ AND C_k AND $A_1 = V.A_1$ AND $\cdots$ $A_n = V.A_n$) WHERE EXISTS (SELECT * FROM R_1 & $\cdots$ & R_m WHERE NOT C_0 AND C_1 AND $\cdots$ AND C_k AND $A_1 = V.A_1$ AND $\cdots$ $A_n = V.A_n$) INSERT INTO V SELECT $A_1, \ldots, A_n, F_1(B_1), \ldots, F_j(B_j)$ FROM R_1 & $\cdots$ & R_m WHERE NOT C_0 AND C_1 AND $\cdots$ AND C_k AND NOT EXISTS (SELECT * FROM V $A_1 = V.A_1$ AND $\cdots$ $A_n = V.A_n$) GROUPBY $A_1, \ldots, A_n$	(1, 6)

Table 8: Adaptation Techniques for Aggregate Views, Continued

1. Each of the aggregation functions $F_1(B_1), \ldots, F_j(B_j)$ are decomposable into the functions $G_1(E_1), \ldots, G_j(E_j)$ over the attributes of the view V, as listed in Table 1.
2. The dropped attribute, $A_p = A_n$ is functionally determined by the remaining grouping attributes $A_1, \ldots, A_{p-1}$.
3. The added attribute, A_{p+1} is functionally determined by a grouping attribute A_j which is the key for relation R_i.
4. There was no previous aggregation or grouping, i.e., $p = j = 0$, and the grouping attributes A_i, and aggregated attributes D_i are present in V. Also $r \geq s$.
5. Attribute of conjunct in `HAVING` clause is either an attribute of the view, or of a wider augmented stored view.
6. The choices of H_i and H'_i are described in Table 2 and 3.
7. $M_m = COUNT(*)$ is either an attribute of the view, or of a wider augmented stored view.

Table 9: Assumptions for the Adaptation Techniques in Table 7 and 8

No.	Redefined View	Adaptation Technique	Assumptions
28	V_1 UNION V_2'	INSERT INTO New_V (SELECT * FROM V WHERE $V.SubQ =$ "V_1" UNION Other adaptation technique applied to SELECT * FROM V WHERE $V.SubQ =$ "V_2")	(1,2)
29	V_1	DELETE * FROM V WHERE $V.SubQ =$ "V_2"	(1)
30	V_1 UNION V_2 UNION V_3	INSERT INTO V SELECT ..., $SubQ =$ "V_3" FROM ... WHERE ...	(1,3)
31	V_1 EXCEPT V_2'	DELETE FROM V WHERE $V.*$ IN SQL(V_2^+)	(4,6)
32	V_1 EXCEPT V_2'	DELETE FROM V WHERE $V.*$ IN SQL(V_2^+) INSERT INTO V (SQL(V_2^-) INTERSECT SQL(V_1))	(6)
33	V_1' EXCEPT V_2	DELETE FROM V WHERE $V.*$ IN SQL(V_1^-)	(5,6)
34	V_1' EXCEPT V_2	DELETE FROM V WHERE $V.*$ IN SQL(V_1^-) INSERT INTO V (SQL(V_1^+) EXCEPT SQL(V_2))	(6)
35	V_1 EXCEPT V_2 EXCEPT V_3	DELETE FROM V WHERE $V.*$ IN SQL(V_3)	(6)

Table 10: Adaptation Techniques for Union and Difference Views

1. An extra attribute determining which argument of the union the tuple came from is kept as part of the view.
2. If the other adaptation technique for V_2 can be expressed as an in-place update, then so can the adaptation technique for the union.
3. The given SQL outline is the definition of V_3.
4. V_2' can be shown to be weaker than V_2, i.e., $V_2 \subseteq V_2'$.
5. V_1' can be shown to be stronger than V_1, i.e., $V_1' \subseteq V_1$.
6. SQL(V_i), SQL(V_i^+) and SQL(V_i^-) correspond to the SQL code for V_i, V_i^+ and V_i^- respectively.

Table 11: Assumptions for the Adaptation Techniques in Table 10