Adapting Materialized Views after Redefinitions: Techniques and a Performance Study*

Ashish Gupta†
IBM Almaden Research Center
ashish@almaden.ibm.com

Inderpal S. Mumick
AT&T Laboratories
mumick@research.att.com

Jun Rao‡
Columbia University
junr@cs.columbia.edu

Kenneth A. Ross‡
Columbia University
kar@cs.columbia.edu

Columbia University Technical Report CUCS-010-97
Mar. 14, 1997

Abstract

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 see the changes in the results fast. Data archaeology, data visualization, and dynamic queries are examples of such applications.

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 more efficient than rematerialization in most cases. Certain adaptation techniques can be up to 1,000 times better. We also point out the physical layouts that can benefit adaptation.

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

†Research supported by NSF grants IRI–91–16646 and IRI–92–23405.

‡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.

1 Introduction

Many applications try to visualize views over data stored in a database. The view is materialized, 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].

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 of the query, and then changing the query iteratively as the archaeologist's understanding improves.

We consider the problem of recomputing a materialized view 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."

1.1 Motivating Example

Example 1.1: Consider the following relations E (employees), W (works), and P (projects):

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

The key of each relation is underlined. Consider a graphical interface used to pose queries on the above relations using `SELECT`, `FROM`, `WHERE`, `GROUPBY`, and other SQL constructs. For instance, consider the following view defined by query Q_1.

`CREATE VIEW` V `AS`
`SELECT` $Emp\#$, $Proj\#$, $Salary$
`FROM` E & W
`WHERE` $Salary > 20000$ `AND` $Hours > 20$

The natural join between relations E and W on attribute $Emp\#$ is specified as a part of the `FROM` clause using the "&" sign. Query Q_1 might be specified graphically using a slider for the *Salary* attribute and another slider for the *Hours* attribute. As the position of these sliders is changed, the display is updated to reflect the new answer.

Say the user shifts the slider for the *Salary* attribute making the first condition $Salary > 25000$. The answer to this new query can be computed easily from the answer already displayed on the screen. All those tuples that have *Salary* more than 20000 but not more than 25000, are removed from the display. 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 the slider for *Salary* is moved to lower the threshold of interest to $Salary > 15000$, then the above computation is not possible. However, we can still infer that (a) the old tuples still need to be displayed and (b) some more tuples need to be added, namely, those tuples that have *salary* more than 15000 but not more than 20000. 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 user decides to change Q_1 by joining it with relation P and then computing an aggregate. That is view V now is defined by a new query Q_2:

`CREATE VIEW` V `AS`
`SELECT` $Proj\#$, $Location$, $SUM(Salary)$
`FROM` E & W & P
`WHERE` $Salary > 20000$ `AND` $Hours > 20$
`GROUPBY` $Proj\#$, $Location$

Thus Q_2 requires that Q_1 be joined with relation P on attribute *Proj#* and the resulting view be grouped by *Proj#* and *Location*. Note that the key for relation P is *Proj#* and *Proj#* is already in the answer to query Q_1. Thus, to compute Q_2 we need only look up the *Location* attribute from the relation P using the value of *Proj#* for each tuple in the current answer set. The resulting set of tuples is aggregated over the required attributes to compute the answer to query Q_2.

Finally, say the user changes view V to compute the sum of salaries for each *Location* that appears in Q_2. The answer to this query (call it Q_3) is computable using only the result of Q_2. Because the grouping attributes of Q_2 are a superset of the grouping attributes of Q_3, each group of Q_2 is a subgroup of a group in Q_3. Thus, multiple tuples in the result of Q_2 are combined together to compute the answer to Q_3. □

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

1.2 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 concatenating the maintenance process for each local change. In almost all cases, this concatenation 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. These algorithms integrate smoothly with a cost-based query optimizer. The optimizer considers the additional plans provided by the 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 far more efficient with this information.

Our work shows that (a) it is often significantly better to use previously materialized views, and (b) if you know in advance that you might change the views in certain ways, then you can include appropriate kinds of additional information in the views.

We then present a thorough experimental evaluation of the adaptation techniques. The results support our analysis and also lead to some interesting observations about physical design for materialized views.

1.3 Related Work

The problem of redefining materialized views is related to the problem of optimizing an arbitrary query given that the database has materialized a view V. The query can be considered to be a redefinition of the view V and one may compute the query by changing the materialization of V. However, there is an important difference. Consider a query that returns all the tuples in the view except one. When framed as a query optimization problem, the complexity of using the view is $O(|V|)$, where $|V|$ is the cardinality of the materialization of V. When framed as a view adaptation problem, the complexity of the maintenance process is $O(\log(|V|))$ since we can simply delete one tuple from V. This will impact the choice of the strategies for query answering and

view adaptation differently. Further, the view adaptation approach loses the old materialized view, while the querying approach keeps the old view in storage. Thus view adaptation is not just a special case of the problem of answering a query given some materialized views. If we omit in-place updates, this is a special case of answering queries using materialized views. But in-place updates may lead to more efficient solutions.

View adaptation differs from the problem of using materialized views to answer queries 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 would have to spend a considerable time determining how to use the existing views to *correctly* answer a given query. Thus, adaptation considers a smaller search space and yields a smaller but more efficient set of standard techniques that are easily incorporated in relational systems.

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.

[LY85, YL87] look at the question of answering queries using cached results or materialized views. [LY85, YL87] show how to transform an SPJ (select-project-join) query so that it is expressed completely using a given set of views, without any reference to the base relations. They also have the idea of augmented views where each view is extended with keys of the underlying base relations.

[CKPS95] tackle the broader problem of trying to answer any query given any set of view definitions. Because they look at this more general problem, they have a much larger search space (exponential size) in their optimization algorithm. We have a simple small set of extra plans to check. For the less general problem we can do more, and do it more efficiently.

[RSU95, LMSS95] also tackle the problem of answering a query given any set of view definitions. They do not consider aggregate queries. Subsequently, [DJLS95, GHQ95] discuss how to answer aggregate queries using materialized aggregate views. Their results subsume the results presented in Section 4.

[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 each correspond an to access to a 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).

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. (Equivalently, one could use the "NATURAL JOIN" keywords provided in SQL2.)

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

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 `SQL2` "`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 2.1: Key Attributes: A *key* of a relation R is a minimal subset of the attributes of R that uniquely identifies tuples in R. □

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 definition be called V'. When the extent of V' is obtained utilizing the previously materialized extent of view V, the process will be called *adapting* view V. When the extent of V' is obtained by evaluating the view definition, without utilizing the previously materialized extent of view V, the process will be called *recomputing* 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.

A view can be augmented only by adding more attributes and/or more tuples. Thus, the original view has to be a selection and/or projection of the augmented view. The additional attributes may be useful to adapt the view in response to changing selections, projections, grouping, and unions.

Next, as the user redefines a view, the redefinition is translated into the sequence of primitive changes, and the system must analyze the augmented view and the redefinition changes to determine (1) whether the augmented view can be adapted, and (2) the various algorithms for adapting the augmented view. The adaptation algorithms can also be expressed in SQL; For example, the redefined view can be materialized as an SQL query over the old view and the base relations. Alternatively, the redefined view can be defined by one or more SQL inserts, deletes and updates into the old materialization of the view, or even by simply recomputing the view from base relations. The system can use an optimizer to choose the most cost-effective alternative for adapting the view.

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.

2.4 In-place Adaptation

When view V is redefined to yield V', the new view must be materialized, the old materialization for V must be deleted, and the new materialization must be labeled V. The adaptation process can try to use the old materialization of V as much as possible to avoid copying tuples. Thus, we consider adaptation methods that change the materialization of V in place. If a small number of tuples are being changed, then in-place adaptation is likely to be superior to constructing a new version of the materialization. If every tuple is being changed, then the relative merits of in-place updates and constructing new versions will depend on the performance characteristics of the system. In-place adaptation is done using SQL `INSERT`, `DELETE`, and `UPDATE` commands. We use the following `SQL` syntax for updates:

```
UPDATE V SET A = (SELECT B
                  FROM   R1 & ··· & Rm
                  WHERE  C1 AND ··· AND Ck).
```

The conditions in the `WHERE` clause of the subquery can refer to the tuple variable V being updated. The subquery is required to return only one value. It is possible that attribute A does not appear in the old definition of view V, in which case an `ALTER TABLE` statement should precede the `UPDATE` statement. An in place extension of the table may not be possible due to physical space restrictions, making the `ALTER TABLE` command expensive. On the other hand, systems may choose to keep some free space in each tuple to accommodate frequent adaptation, or use space created by deleted attributes.

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 maintain the redefinition, and (b) What extra information may be kept to facilitate maintenance.

A generic materialized view V may be defined as

```
CREATE  VIEW V AS
SELECT  A_1, ..., A_n
FROM    R_1 & ··· & R_m
WHERE   C_1 AND ··· AND C_k
```

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: In one pass of the old view we can project out the unneeded attributes to get the new view. Alternatively, one could simply keep the old view V, and make sure that accesses to the new view V' are obtained by pipelining a projection at the end of an access to V.

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* into the base relations.

Example 3.1: Suppose our database consists of three relations E, W, and P as in Example 1.1. 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. □

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. A test for the possibility of adaptation by joining the old view with the relation from which the extra columns are to be obtained can be constructed as follows: Define query $Q1$ to be the new view definition. Define query $Q2$ by joining the old view name with the relation R from which the extra column(s) need to be obtained. All attributes in the old view that were derived or equated to attributes from relation R are used as join attributes. The view is adaptable if queries $Q1$ and $Q2$ are equivalent [Ull89, GSUW94]. The above test is similar to tests in [RSU95, LMSS95] to check 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:

```
DELETE FROM V WHERE NOT C_1'

INSERT INTO V
 (SELECT   A_1, ..., A_n
 FROM      R_1 & ... & R_m
 WHERE     C_1' AND NOT C_1 AND C_2 AND ... AND C_k)
```

Alternatively, if the attributes of C_1' are not available in the view, the view adaptation algorithm for the SELECT clause could have materialized some extra attributes in an augmented relation W, or obtained these attributes using joins with the relation containing the attribute, 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 using the techniques of the previous section.

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 3.2: Let E and W be as defined in Example 1.1. Consider a view V defined by

CREATE VIEW V AS
SELECT * FROM E & W WHERE *Salary* > 50000

Suppose that we wish to adapt V to

SELECT * FROM E & W WHERE *Salary* > 60000

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

V^- = SELECT * FROM V WHERE
Salary ≤ 60000 AND *Salary* > 50000
V^+ = SELECT * FROM E & W WHERE
Salary > 60000 AND *Salary* ≤ 50000

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

SELECT * FROM E & W WHERE *Salary* > 49000.

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

SELECT * FROM E & W WHERE *Salary* > 49000 AND *Salary* ≤ 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 3.3: 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 3.2. □

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.

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 [G. Lohman, personal communication].

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` $D_1, \ldots, D_j$."

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

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

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

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 3.4: 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, and the key of T is in the old view.

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 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 5 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. Keep the keys of referenced relations from which attributes may be added. Store the view with padding in each tuple for future in-place expansion. Keep attributes referenced by the selection conditions in the view definition, or at least keep the keys of referenced relations from which these attributes may be added. Keep the count of the number of derivations for each tuple.

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 4.1: Consider again the relations of Example 1.1. 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.

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

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 4.1, 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 4.1, 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 4.2: Consider the aggregation view defined first in Example 4.1, 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, SUM((Salary × 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 4.3: Consider the following view based on the view of Example 4.1. 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 × 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 4.3 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 paper. 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 original qualifying tuples. In order to adapt the old view, we need to access those tuples satisfying the new criteria in the base tables, compute the aggregate functions on them and then adjust the old results. 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`	$A_1, \ldots, A_n$

We can update the aggregate values in the old view using the formula:

$M_i = H_i(M_i, \Delta F_i(B_i))$,
where $\Delta F_i(B_i) =$ `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` $\cdots$ $A_n = V.A_n$
for each $(A_1, \ldots, A_n)$ in the old view V.

H_i depends on the aggregate function F_i as shown in Table 2.

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.

Given the expression above, we can use an `UPDATE` statement to adapt the old view. But there are situations where $\Delta F_i(B_i)$ returns a NULL value (no tuple satisfies the condition), which may cause the final results to be NULL if we do a naive update. What we want actually is to leave the old value unchanged. So we need to specify that the `UPDATE` will only be done when necessary.

After the update, those groups containing no tuple should be deleted. But this can't be done without the help of a *COUNT*(*) in the original view (since you can't distinguish whether a zero means zero value or no tuples). So the original view needs to be augumented when necessary.

Example 4.4: Let P be as defined in Example 1.1. 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 is used to eliminate the side effect of a NULL change. □

We can treat the situation when there are scalar aggregates (no groupby clause) in the view definitions as a special case since there is only one group in the view. The DELETE clause is not necessary since no group needs to be deleted. The details can be found in Table 7 in Appendix A.

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
```

The tuples satisfying the deleted condition may 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 according to tuples in the first set and then insert new groups into the old view by using tuples in the second set. The update can be performed in a similar way:

$M_i = H'_i(M_i, \Delta F_i(B_i))$,
where $\Delta F_i(B_i) =$

```
SELECT   F_i(B_i)
FROM     R_1 & ··· & R_m
WHERE    NOT C_1 AND C_2 AND ··· AND C_k AND
         A_1 = V.A_1 AND ··· A_n = V.A_n
```

for each $(A_1, \ldots, A_n)$ in the old view V.

We summarize H'_i in Table 3. The details of the technique are listed in Table 7 in Appendix A.

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.

4.6 Summary: GROUPBY Views

We assume that the initial view definition is

```
CREATE    VIEW V AS
SELECT    A_1, ..., A_n, F_1(B_1), ..., F_j(B_j)
FROM      R_1 & ··· & R_m
WHERE     C_1 AND ··· AND C_k
GROUPBY   A_1, ..., A_p
```

where $p \geq n$.

The full list of adaptation techniques for aggregate views is given in Appendix A in Table 6. The assumptions used are listed in Table 8. Table 6 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. 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 5.1: Consider the schema from Example 1.1. 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.

5.2 EXCEPT

Example 5.2: Consider again the schema from Example 1.1. 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 9. The assumptions used are listed in Table 10. Table 9 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,

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

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

$$\texttt{SELECT * FROM } V \texttt{ WHERE } C.$$

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 6.1: Let us define a materialized view V by

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

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 between 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 compare some of our adaptation techniques with rematerialization. We also give physical design suggestions for making adaptation more efficient.

7.1 Description of the environment

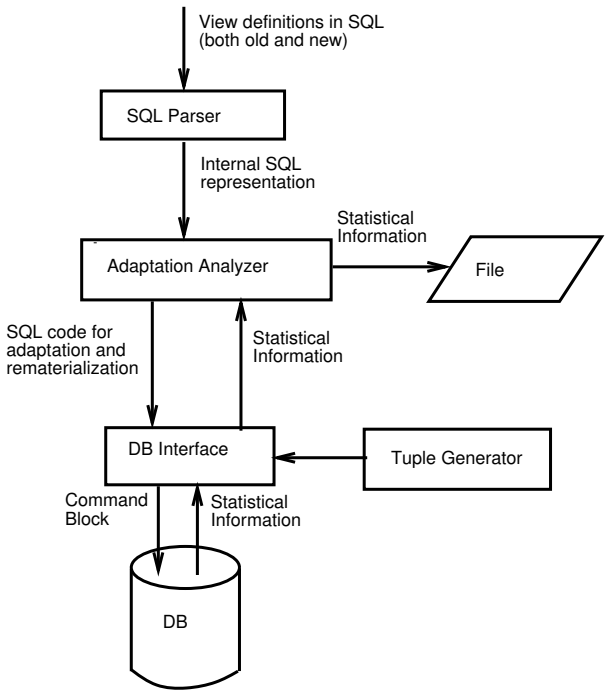


Figure 1: Architecture of our testing system

We implemented a system 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. We start a timer before sending the SQL code and stop the timer when we get the results back. The time we measured is wall time. (Response time, as measured by the database server, was always within 1 second's interval from the wall 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.

The sample 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.

Some of the techniques have both in-place and non-in-place versions. Rematerialization is done by using the "select into" clause that doesn't perform logging. Some adaptation techniques (the in-place versions) will need to do logging for recovery; there was no easy way to turn off logging in our database. Although logging doesn't take much time when the amount of update is small, our setup actually favors rematerialization a little bit.

7.2 TPC-D Tables

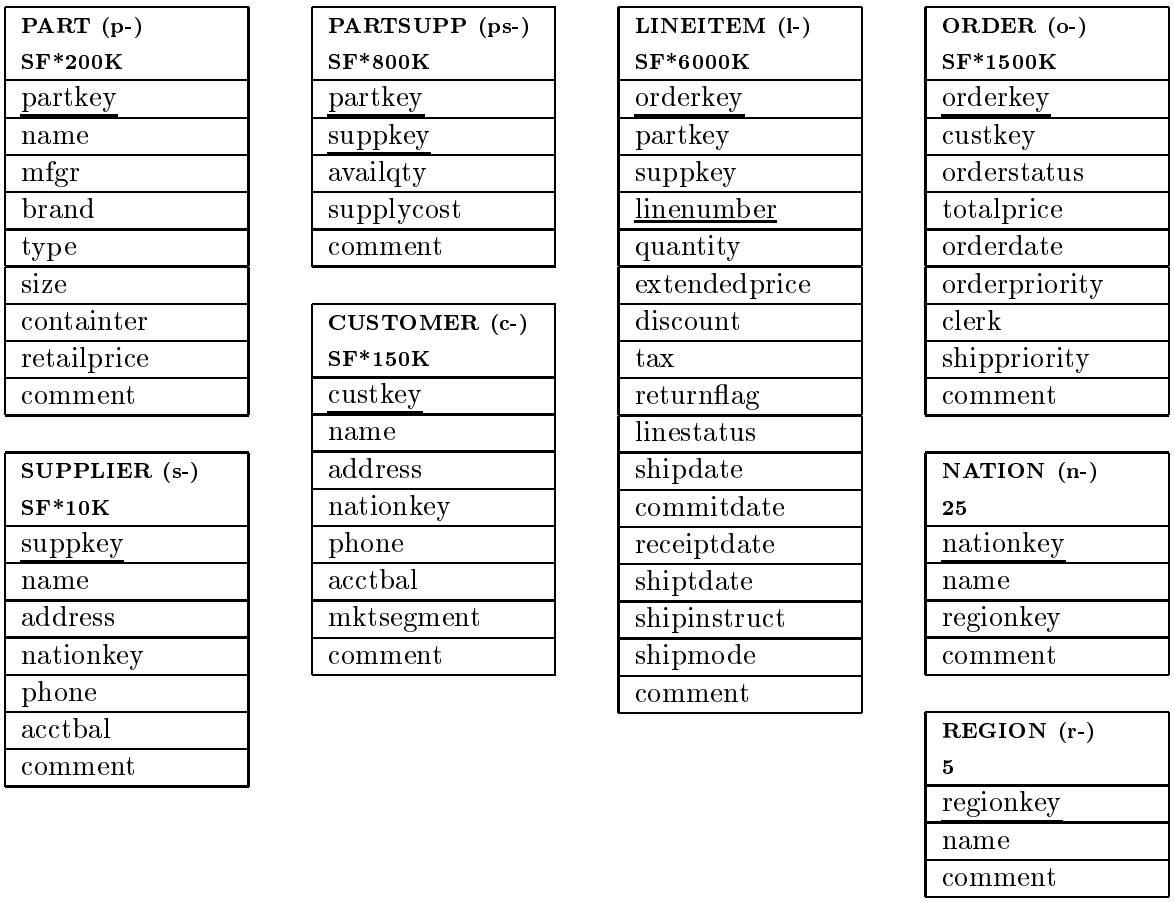


Figure 2: Tables of TPCD Benchmark

There are eight tables in the database shown in Figure 2. For each table, we show the name of the table, its cardinality (some are factored by SF, the Scale Factor) and the attributes (with primary

keys underlined). The parentheses following each table name contain the prefix of the column names for that table.

7.3 Design of test sets

We designed five test sets, each testing a group of related adaptation techniques. In the figures (Figure 3, 4, 7, 9, 12) 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. 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 Figure 3 and 7, the tables we use will be too small at scale factor 0.1 and therefore are populated at scale factor 1. All the base tables are physically ordered by their primary keys.

7.4 Changes in the SELECT Clause

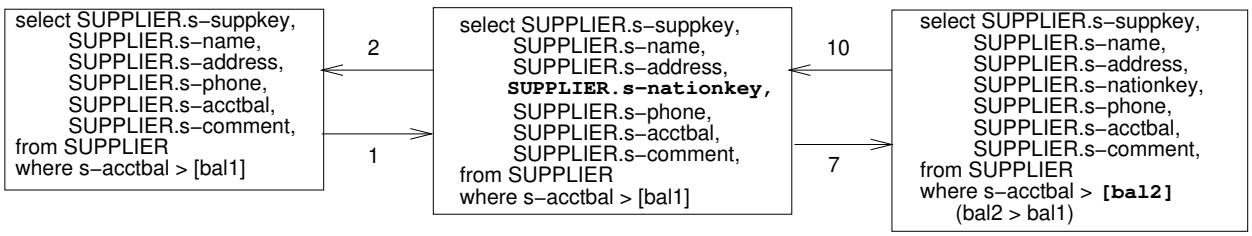


Figure 3: Test Set 1

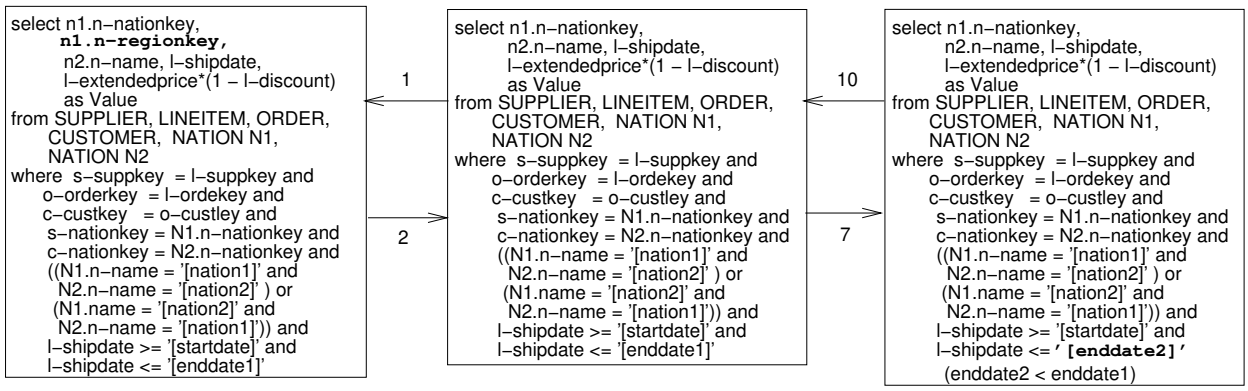


Figure 4: Test Set 2

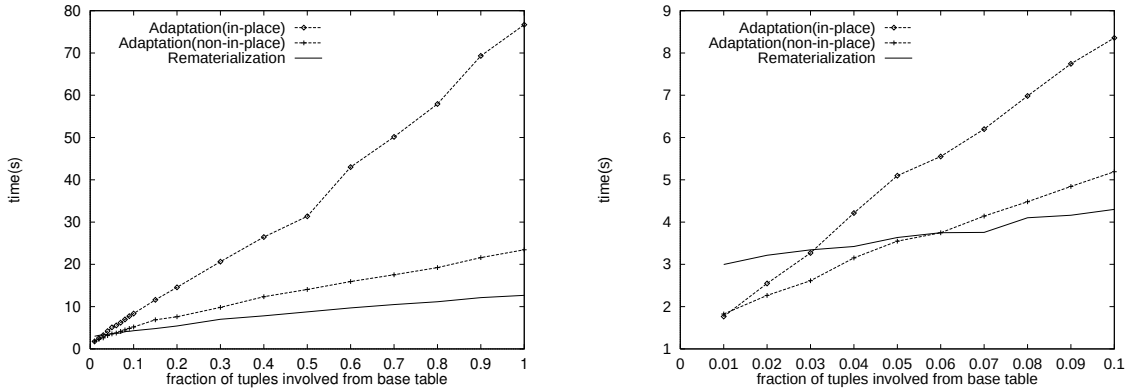
We use Test Set 1 and 2 (Figure 3 and 4) to measure the performance of adaptation techniques 1 and 2 for handling changes in the `SELECT` clause. Graphs in Figure 5(a)-5(d) summarize our results. Figure 5(a) and 5(c) are based on Test Set 1 (Techniques 1 and 2). The graphs are derived by varying `[bal1]` from -1,000.00 to 10,000.00. The x-axis is measured as the fraction of tuples from `SUPPLIER` that'll be included in the view. Figure 5(b) and 5(d) are based on Test Set 2 (Techniques 1 and 2). `[nation1]` and `[nation2]` are chosen as 'CHINA' and 'JAPAN', respectively. `[startdate]` is chosen as '1992-01-01'.[1] 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 5(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 5(a)).

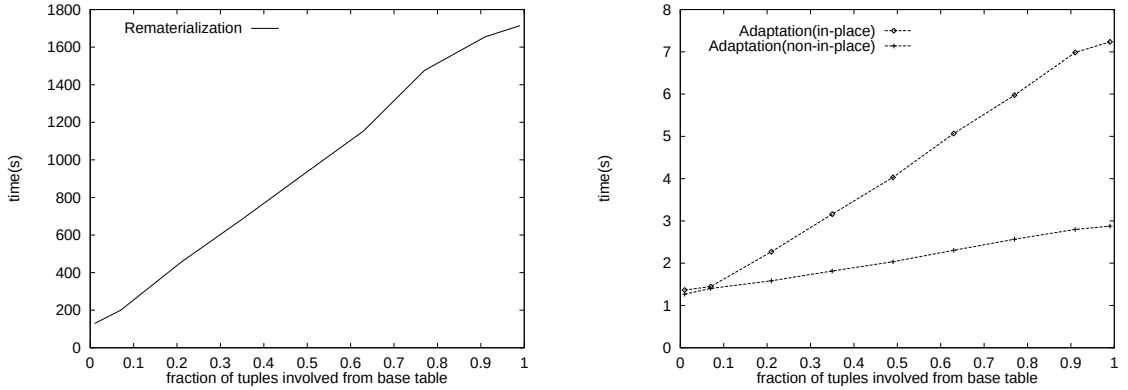
The results are strikingly different for SPJ views as shown in Figure 5(b). Adaptation wins by a wide margin over rematerialization. The reason is that rematerialization needs to do a six-way join, while adaptation only has to do look-ups from one base table. In both 5(a) and 5(b), the non-in-place version is always better than the in-place version. Although not shown here, the size of the base table where the new column being added to the view comes from also plays a role.

Dropping a column Figure 5(c) shows that adaptation outperforms rematerialization. This is because rematerialization needs to read in the whole base table but adaptation only has to read from the old view which is smaller. In Figure 5(d), adaptation wins significantly because rematerialization needs to do an expensive join while adaptation just accesses the old view.

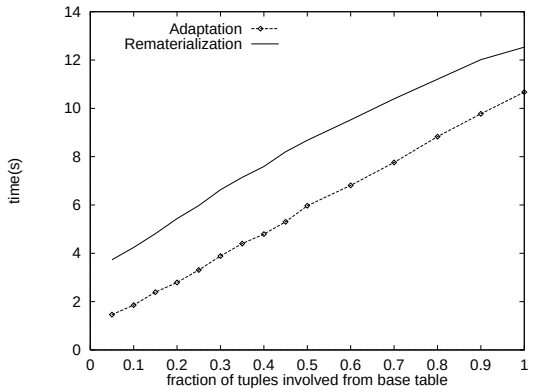
[1] We have chosen a different set of values and rerun the experiments. The results remain the same.



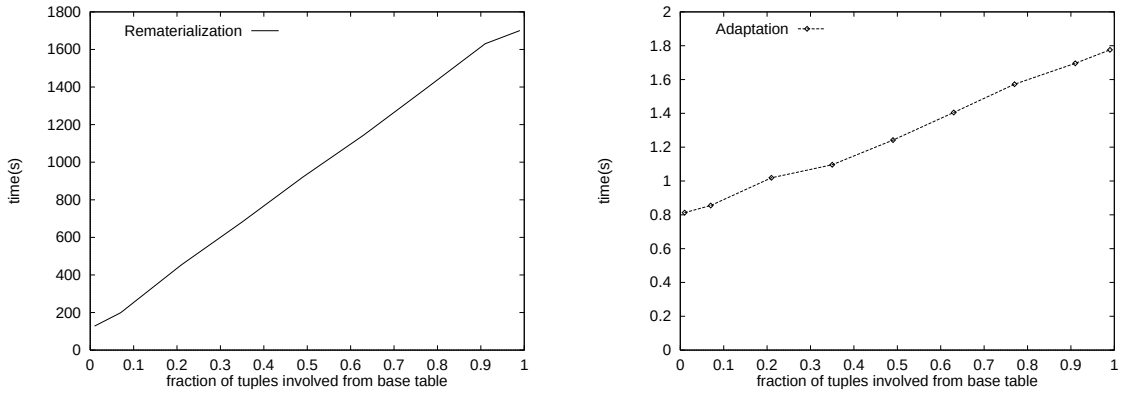
(a) Adding a column (without join)



(b) Adding a column (with join)



(c) Dropping a column (without join)



(d) Dropping a column (with join)

Figure 5: Changes in the `SELECT` Clause

7.5 Changes in the `WHERE` Clause (no aggregation)

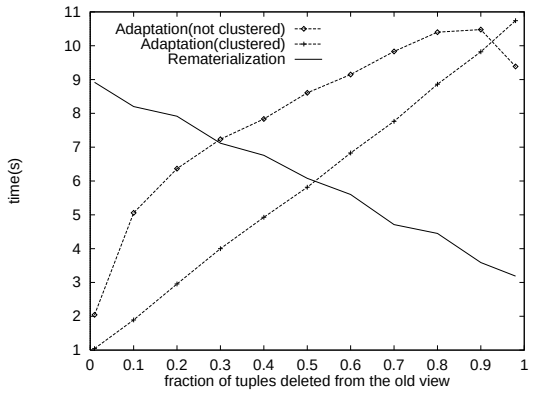
We measure the performance of adaptation techniques 7 and 10 for handling changes in the `WHERE` clause using again Test Set 1 and 2 (Figure 3 and 4). Graphs in Figure 6(a)-6(d) summarize the results. Figure 6(a), 6(b) and 6(d) are based on Test Set 1 (Techniques 7 and 10). For Figure 6(a) and 6(b), `[bal1]` is fixed at 4500.00 and `[bal2]` varies from 4500.00 to 10,000.00. For Figure 6(d), `[bal2]` is fixed at 4500.00 and `[bal1]` varies from 4500.00 to 10,000.00. The two lines of the adaptation in Figure 6(a), 6(b) and 6(d) correspond to no clustering in the old view and clustering on `s-acctbal` in the old view (i.e., the old view is physically ordered by `s-acctbal`). Clustering is achieved by building a clustering index on `s-acctbal`. Figure 6(c) and 6(e) are based on Test Set 2 (Techniques 7 and 10). For Figure 6(c), `[enddate1]` is fixed at '1998-11-01' and `[enddate2]` varies from '1992-01-01' to '1998-10-01'. For Figure 6(e), `[enddate2]` is fixed at '1995-06-01' and `[enddate1]` varies from '1995-06-15' to '1998-10-01'.

Adding a condition As shown in Figure 6(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) when the deletion percentage reaches a certain point. 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 many 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[2], which lowers the cost of adaptation. 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. It should be noted that the clustered index is built on the old materialized view, not on the base table. Additionally, in both cases, adaptation loses to rematerialization when the fraction of deletions is high.

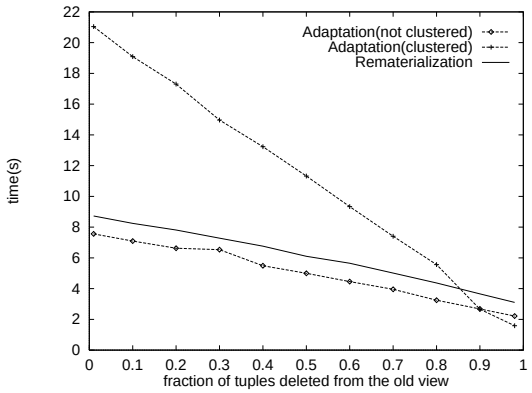
Figure 6(b) looks different from its in-place counterpart due to their different implementations. Without an index in the old view, adaptation always outperforms rematerialization. The reason is that adaptation only needs to read from the old view rather than a base table which is larger. Adaptation 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 non-in-place version has the feature of doing better in the whole range, but having an index on the view is not a good idea. Figure 6(c) shows a dramatically different result. Since rematerialization needs to do a six-way join while adaptations (both the in-place and the non-in-place version) only need to access the old view, adaptations win considerably.

Deleting a condition In Figure 6(d), both adaptation and rematerialization need to read from the base table. But adaptation saves the time of writing out those tuples already in the old view. This explains why adaptation wins when there is no clustered index in the view. With a clustered index in the old view, it costs adaptation some time to maintain the index. That's the reason 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 slope of the adaptation (not clustered) is 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 6(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.

[2]The cost of maintaining the clustered index is included in the time we measured.

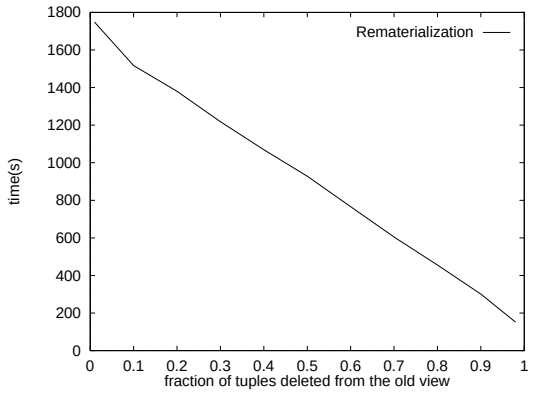


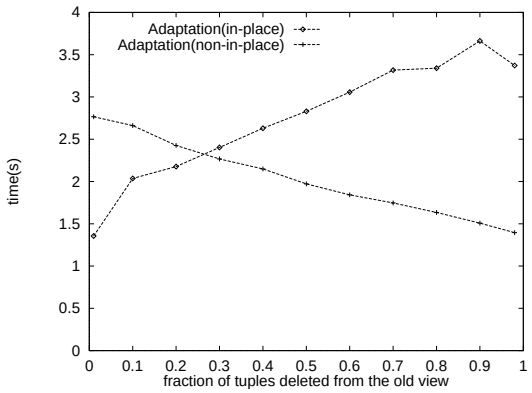
(a) in-place version



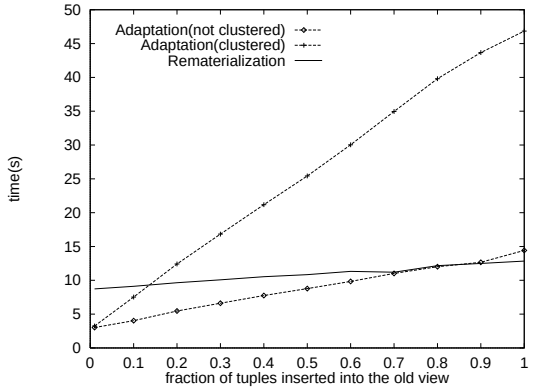
(b) non-in-place version

Adding a condition (without join)

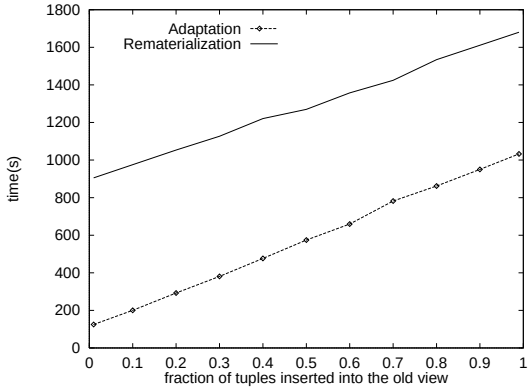




(c) Adding a condition (with join, in-place and non-in-place version)



(d) without join



(e) with join

Deleting a condition

Figure 6: Changes in the `WHERE` Clause (no aggregation)

7.6 Changes in the FROM Clause

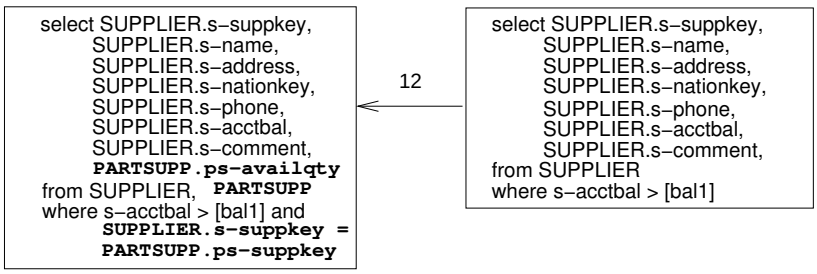


Figure 7: Test Set 3

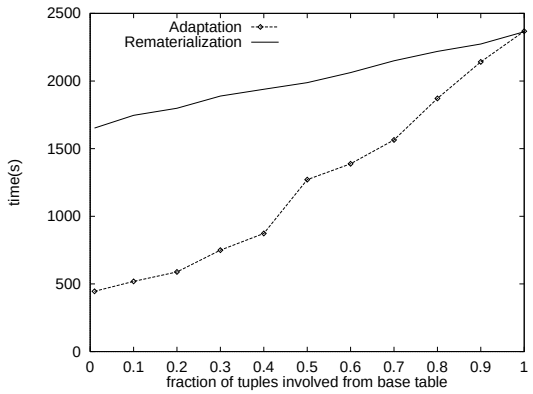


Figure 8: Adding a table

We use Test Set 3 (Figure 7) to measure the performance of adaptation technique 12 for adding a table in the FROM clause. The result is shown in Figure 8. The graph is derived by varying [bal1] from -1,000.00 to 10,000.00. The x-axis is measured as the fraction of tuples selected from SUPPLIER.

Rematerialization needs to join two base tables while adaptation only 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 definitions. So when there are more than two tables in the new view definition, adaptation will win significantly in a larger range by doing a cheaper join.

7.7 Changes in the GROUPBY Clause

We use Test Set 4 (Figure 9) to measure the performance of adaptation techniques 18 to 21 for handling changes in the GROUPBY clause. Graphs in Figure 10(a)-10(d) summarize our results. The graphs are derived by choosing [nation] as 'CHINA' and varying [partkey] within the range. The x-axis is measured as the fraction of tuples from PARTSUPP participating the join.

All the graphs in Figure 10 look similar. Since rematerialization needs to perform a three-way join while all the adaptations only need to access the old view, adaptations win significantly. Additionally, adaptations in Figure 10(b) and 10(c) also save computation time for the groupby clause. An interesting observation is that when the fraction of tuple involvement reaches 0.9, the optimizer changes its plan which causes the bumps in the graphs.

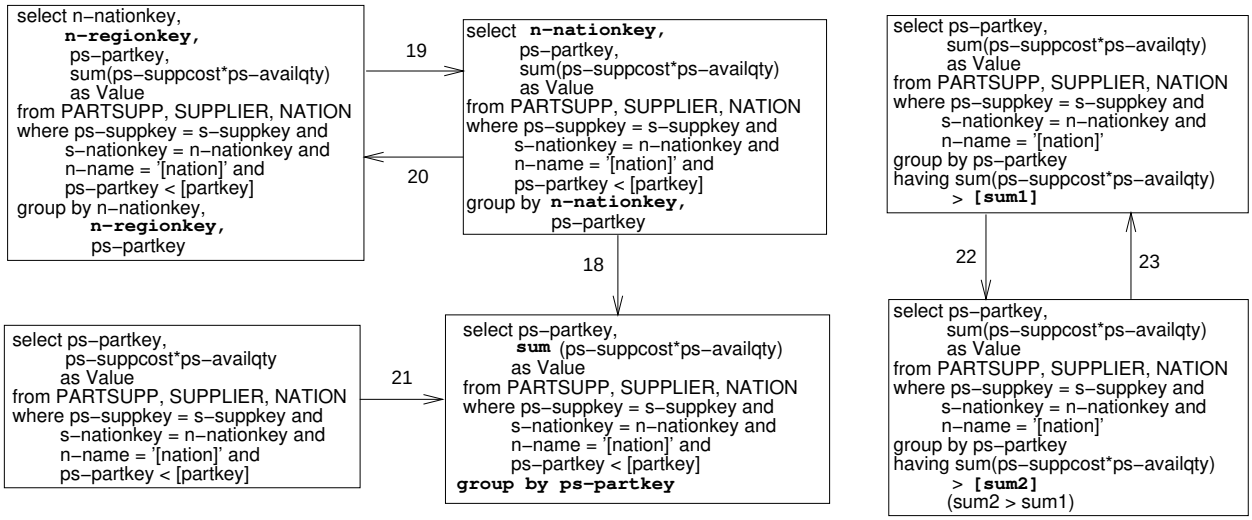
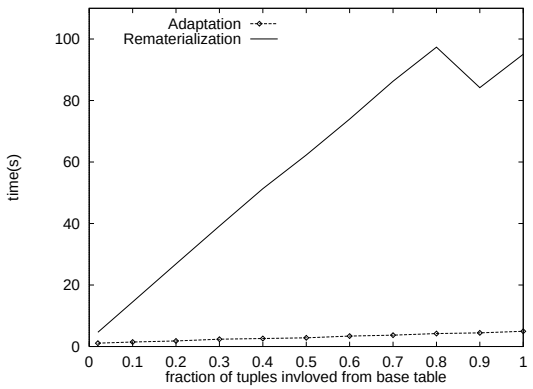
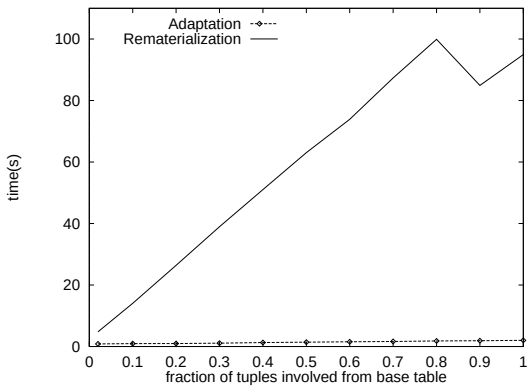


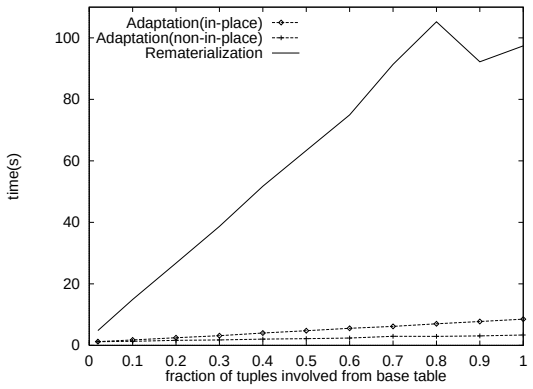
Figure 9: Test Set 4



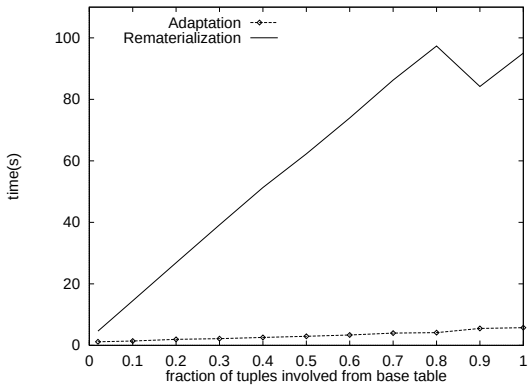
(a) Dropping a groupby column (18)



(b) Dropping a groupby column (19)



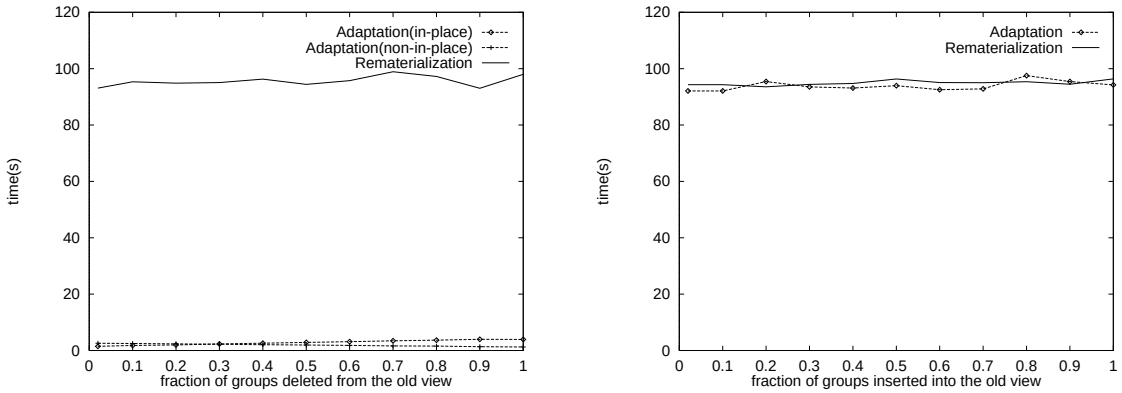
(c) Adding a groupby column



(d) Adding the groupby clause

Figure 10: Changes in the GROUPBY Clause

7.8 Changes in the HAVING Clause



(a) Adding a conjunct (b) deleting a conjunct

Figure 11: Changes in the HAVING Clause

We also use Test Set 4 (Figure 9) to measure the performance of adaptation techniques 22 and 23 for handling changes in the HAVING clause. Graphs in Figure 11(a)-11(b) summarize the results. For Figure 11(a) (Technique 22), [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 11(b) (Technique 23), [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.

Figure 11(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 11(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 may help (similar to Section 7.5).

7.9 Changes in the WHERE Clause (in the presence of aggregates)

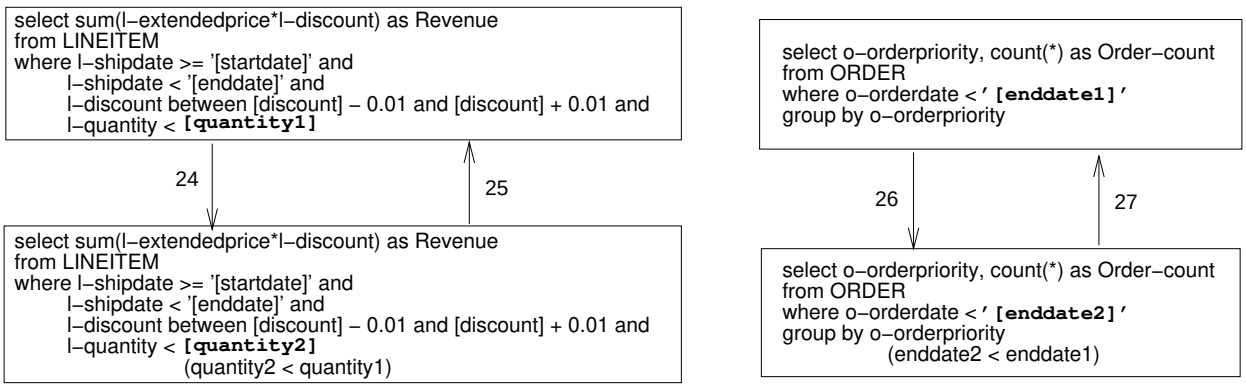


Figure 12: Test Set 5

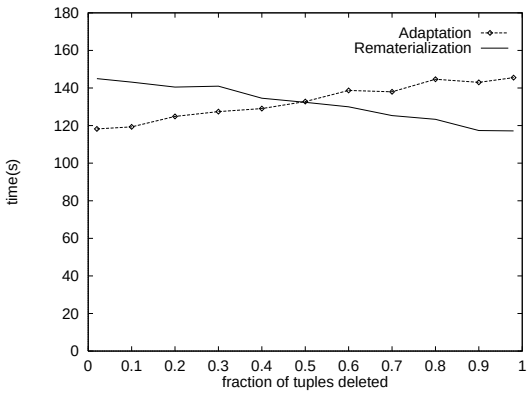
We use Test Set 5 (Figure 12) to measure the performance of adaptation techniques 24 to 27 for handling changes in the WHERE clause in the presence of aggregates. Graphs in Figure 13(a)-13(d) summarize the results when there are only scalar aggregates in the view definition (Techniques 24 and 25). Graphs in Figure 14(a)-14(d) and Figure 15(a)-15(d) show the results when there exists a GROUPBY clause in the view definition (Techniques 26 and 27).

Scalar Aggregates For all graphs in Figure 13, [startdate] and [enddate] are chosen as '1992-01-01'. and '1998-11-01' respectively. [discount] is chosen as 0.03. For Figure 13(a) and 13(b), [quantity1] is fixed at 50 and [quantity2] varies from 0 to 49. For Figure 13(c) and 13(d), [quantity2] is fixed at 25 and [quantity1] varies from 26 to 49. For Figure 13(b) and 13(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 the old view and the number of tuples satisfying the where clause in the old view.

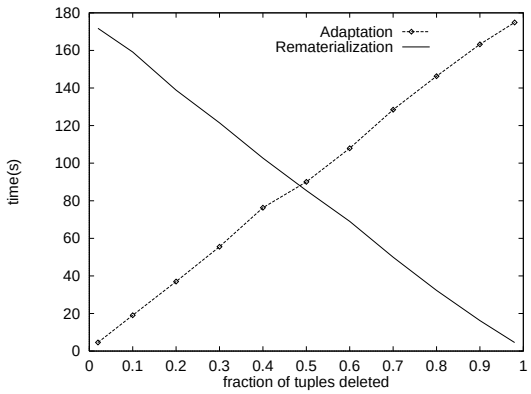
The graphs here are different from those in Section 7.5, This is because we have only one tuple—the aggregation results in the old view. So, while some techniques in Section 7.5 only need to delete some tuples from the old view, all the techniques in this section have to access the base table.

In Figure 13(a), adaptation also needs to read in all the pages of the base table as rematerialization, since 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 13(b), both adaptation and rematerialization can take advantage of the clustered index in the base table. When deletion percentage is small, adaptation saves a lot of I/O time. So does rematerialization when deletion percentage is large. In Figure 13(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 13(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.

The situation will change when there are joins in the view definitions. Although adaptation techniques here need to perform the same join as rematerialization, one of the base tables (the one containing the attribute in the changing predicate) can be restricted to include only the difference. The join for adaptation is cheaper than that for rematerialization, especially when the difference is small. We expect adaptation will outperform rematerializtion (in a way similar to Figure 6(e)).

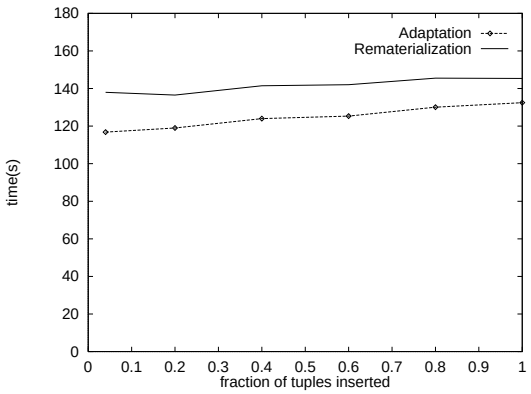


(a) clustered index on key

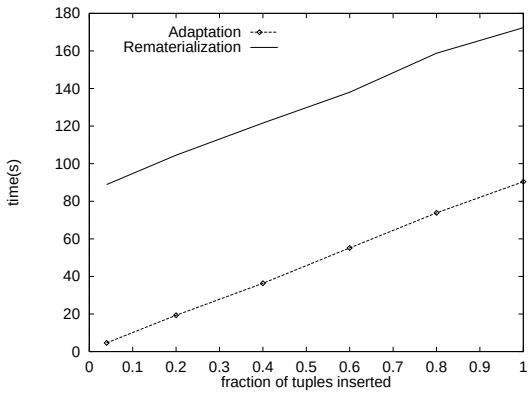


(b) clustered index on `l-quantity`

Adding a condition



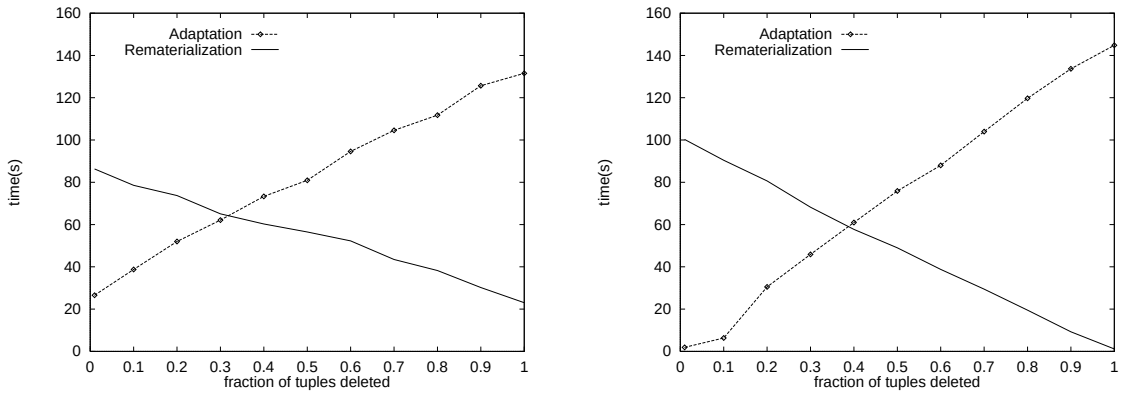
(c) clustered index on key



(d) clustered index on `l-quantity`

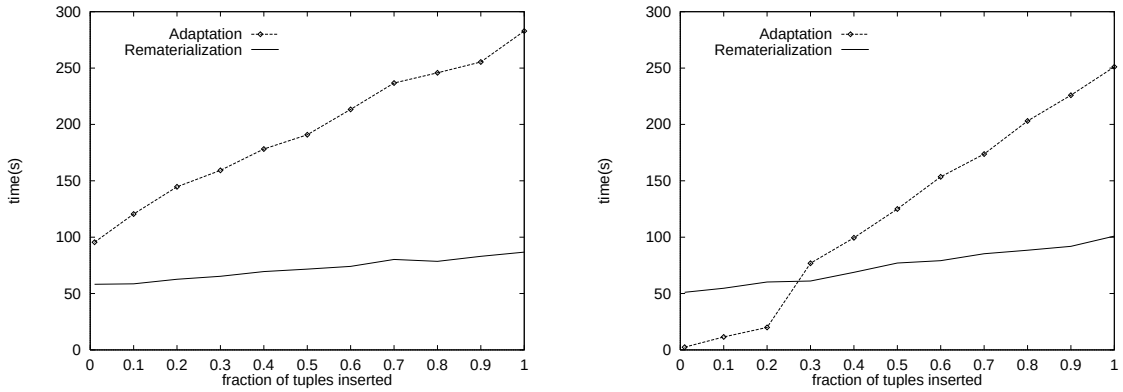
Dropping a condition

Figure 13: Changes in the `WHERE` Clause (in the presence of scalar aggregates)



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

Adding a condition



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

Deleting a condition

Figure 14: Changes in the `WHERE` Clause (in the presence of the groupby clause, 5 groups)

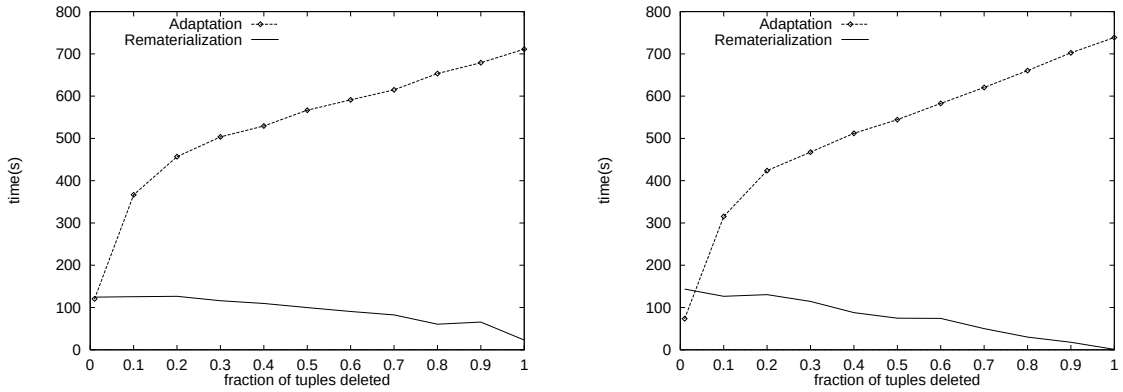
With Groupby For Figure 14(a) and 14(b), `[enddate1]` is fixed at '1998-08-01' and `[enddate2]` varies from '1992-01-01' to '1998-07-01'. For Figure 14(c) and 14(d), `[enddate2]` is fixed at '1995-04-01' and `[enddate1]` varies from '1995-08-01' to '1998-08-01'. For Figure 14(b) and 14(d), a clustered 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 Figure 13. There are only five groups in the view.

The adaptation techniques in this section also need to access the base table. In Figure 14(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 small. In Figure 14(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 14(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, resulting in the poor performance of adaptation. In Figure 14(d), for the same reason, adaptation loses in most cases. But adaptation still wins when the fraction of insertion 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

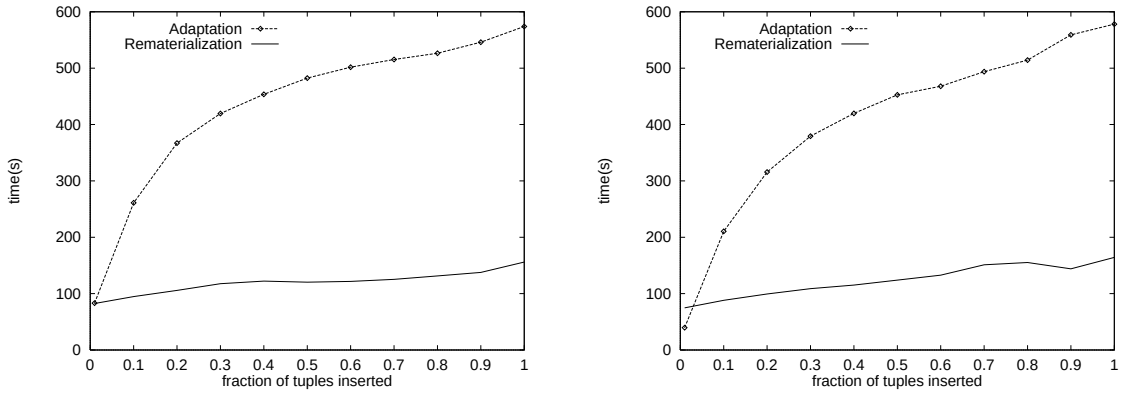
the view. We repeated the test on a different groupby attribute—`O-CUSTKEY` (15K distinct values) and obtained the results in Figure 15. The result suggets that the adaptation techniques here are not suitable for a large number of groups.

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



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

Adding a condition



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

Deleting a condition

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

7.10 Summary

What do we learn from the experiments? First, the "common wisdom" that adaptation will always win is not true. We have seen cases that adaptation performs comparably or even worse than rematerialization. Second, adaptation can do better with the help of certain physical layouts. Here are the high-level conclusions we have reached:

- Adaptation can help in most cases.
- Adaptation is more efficient when
  - there are joins in the view definitions.
  - there exists an appropriate physical order on the view.
  - there exists an appropriate physical order on the base tables.

– the changes in the view definitions are small.

The idea of having a particular physical order on the view is practical, but it may not be practical to have a particular physical order on the base tables since the base tables themselves may already be clustered on some other attributes for various reasons.

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 some other techniques.

8 Conclusions

When the definition of a materialized view changes we need to bring the materialization up-to-date. In this paper 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 is more efficient to adapt a view rather than recompute it, sometimes by an order of magnitude; a number of examples have been described in this paper.

A number of applications, like data-archaeology and visualization, require *interactive*, and thus quick, response to changes in the definition of a materialized view.

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. Almost all 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 paper are (a) the derivation of a comprehensive set of view adaptation techniques, (b) the smooth integration of such techniques into the framework of current relational database systems using existing optimization technology, (c) the identification of guidelines that can be provided to users and database administrators in order to facilitate view adaptation, and (d) 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 maintenance 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.

[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.

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

[DJLS95] Shaul Dar, H.V. Jagadish, Alon Levy, and Divesh Srivastava. Answering SQL queries with aggregation using views. AT&T technical report, 1995.

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

[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. To appear 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.

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

[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.

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, Section 4.6 for Table 6, and Section 5.3 for Table 9). 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 which represents the new materialization. In the event that basic adaptations are pipelined, the tuples may not actually be stored in an intermediate relation.

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	ALTER TABLE V ADD A UPDATE V SETA = (SELECT A FROM 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	ALTER TABLE V DROP A_1	
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_1' AND $\cdots$ AND C_k	DELETE FROM V WHERE NOT C_1'	$C_1' \Rightarrow C_1$, (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' 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)
9	SELECT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_0 AND C_1 AND $\cdots$ AND C_k	DELETE FROM V WHERE NOT C_0	(4)
10	SELECT $A_1, \ldots, A_n$ FROM R_1 & $\cdots$ & R_m WHERE C_2 AND $\cdots$ AND C_k	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	
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 4: Adaptation Techniques for SELECT-FROM-WHERE Views

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 that is also present in V.

Table 5: Assumptions for the Adaptation Techniques in Table 4

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 6: 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 7: 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 8: Assumptions for the Adaptation Techniques in Table 6 and 7

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 9: 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. $\texttt{SQL}(V_i)$, $\texttt{SQL}(V_i^+)$ and $\texttt{SQL}(V_i^-)$ correspond to the `SQL` code for V_i, V_i^+ and V_i^- respectively.

Table 10: Assumptions for the Adaptation Techniques in Table 9