A PERFORMANCE ANALYSIS OF VIEW MATERIALIZATION STRATEGIES

by

Eric Hanson

Memorandum No. UCB/ERL M86/98

12 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

A PERFORMANCE ANALYSIS OF VIEW MATERIALIZATION STRATEGIES

by

Eric Hanson

Memorandum No. UCB/ERL M86/98

12 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A Performance Analysis of View Materialization Strategies

Eric Hanson

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Abstract

The conventional way to process commands for relational views is to use query modification to translate the commands into ones on the base relations. An alternative approach has been proposed recently, whereby materialized copies of views are kept, and incrementally updated immediately after each modification of the database. This paper presents a related scheme in which update of materialized views is deferred until just before data is retrieved from the view. A performance analysis is presented comparing the cost of query modification, immediate view maintenance, and deferred view maintenance. Three different models of the structure of views are given: a simple selection and projection of one relation, the natural join of two relations, and an aggregate (e.g. the sum of values in a column) over a selection-projection view. The results show that the choice of the most efficient view maintenance method depends heavily on the structure of the database, the view definition, and the type of query and update activity present.

1. Introduction

A materialized view is a stored copy of the result of retrieving the view from the database. In this paper, the types of materialized views considered are those that could be defined using SELECT, PROJECT and JOIN, and also simple aggregates such as *sum* or *count* over the result of such expressions.

Conventional systems do not materialize views in advance, but rather use query modification to turn a query referring to a view into one on the base relations [Ston75]. An alternate method for materializing views which updates the copy of the view after each transaction [Blak86] will be called *immediate view maintenance* or simply *immediate* in this paper. A third proposal generates and periodically refreshes *database snapshots*, which are copies of views consisting of selections and projections of a single base table [Adib80, Lind86]. In the context of evaluating complex trigger and alerter conditions, Buneman and Clemons presented a method for analyzing each update command *prior to execution* to see whether it could cause a view to change [Bune79]. If the system could not rule out the possibility that the command might alter the state of the view, the view

This research was sponsored by the National Science Foundation under Grant DMC-8504633 and by the Navy Electronics Systems Command under contract N00039-84-C-0039.

would be completely recomputed. Hence, this represents a fourth view refresh algorithm. Lastly, this paper proposes a final alternative, called *deferred view maintenance*, or *deferred*, that incrementally updates a materialized view just before data is retrieved from it. In this paper, we will analyze and compare the performance of query modification, deferred, and immediate for materializing views.

An important way to improve the performance of view materialization algorithms is to use a *screening* algorithm to test each tuple inserted into or deleted from the base relations. If a tuple passes the screening test, then its insertion or deletion may cause the state of the view to change, so the tuple must be used to try to update the view. If the tuple fails the screening test then it cannot cause the view to change, so it does not need to be used to refresh the view. In the scheme described in [Blak86] screening is done by substituting a tuple into a view predicate, which is then tested to see if it is still *satisfiable*. If so, the tuple passes the screening test, otherwise it fails. This test is performed for every tuple inserted into a relation, incurring a significant CPU cost.

The screening test proposed in [Bune79] has a compile-time phase and a run-time phase. In the first phase, when the command for a transaction is compiled, the system checks to see whether any fields the command proposes to update are read by the view definition. If no such fields are updated, then the command is called a *readily ignorable update* (RIU) with respect to the view. If a command is an RIU, it cannot cause the view to change. In the second phase, if the command is not an RIU, the individual tuples updated are screened further at run time. If a command is an RIU, there is only a per-transaction cost associated with this screening test. If it is not an RIU, then there is a per-tuple cost, similar the screening test of [Blak86].

An alternative test that will usually be more efficient than the two just described is to apply the rule wake-up scheme in [Ston86] to the screening problem. Using this mechanism, called *rule indexing*, the index intervals covered by one or more clauses of the view predicate are locked using special markers called *trigger-locks* or *t-locks*. When a tuple is inserted into the relation, if an index record containing a t-lock is disturbed, then the tuple passes the screening test. Otherwise, the tuple fails the test implicitly. Since this screening test can produce "false drops" (i.e. tuples which pass the screening test but do not satisfy the view predicate), a second stage screening test, substituting the tuple into the view predicate, is appropriate. This strategy is assumed for both immediate and deferred view maintenance in the performance analysis of this paper.

To provide the background necessary for the performance analysis, Section 2 reviews the immediate view maintenance algorithm and describes the proposed deferred view maintenance scheme in detail. In Section 3, cost formulas for each of the algorithms are derived for three different view models:

1. selection-projection views
2. two-way natural join views
3. aggregates over selection-projection views

The performance of the algorithms is compared for each model. Finally, Section 4 presents conclusions, and suggests directions for future research.

2. View Materialization Strategies

In this section, the algorithm for incrementally updating materialized views after each update transaction is described briefly (see [Blak86] for a complete discussion). Our proposed variant of this algorithm to allow deferred view maintenance is then presented.

2.1. The Differential View Update Algorithm

The differential view update algorithm operates on the following sets of tuples:

$R_1, R_2, \ldots R_N$	the N base relations
$A_1, A_2, \ldots A_N$	the N sets of tuples inserted into the base relations by the current transaction
$D_1, D_2, \ldots D_N$	the N sets of tuples deleted from the base relations by the current transaction

The sets $A_1 \cdots A_N$ and $D_1 \cdots D_N$ must contain the *net* changes to the database made by one transaction. Hence:

$$A_i \cap D_i = \varphi$$
$$A_i \cap R_i = \varphi \quad \text{for } 1 \leq i \leq N$$
$$D_i \subseteq R_i$$

The definition of a view V can be represented by *a select-project-cross-product* expression as follows, where σ_X represents selection based on a predicate X, π_Y represents projection of the set of attributes Y, and $\times$ represents cross-product.

$$V = \pi_Y(\sigma_X(R_1 \times R_2 \times \cdots \times R_N))$$

Consider an example with two relations, $R_1(a,b)$ and $R_2(b,c)$, and a view V defined as follows, where $Y = \{a,c\}$ and $X = (R_1.a = 5 \text{ and } R_1.b = R_2.b)$:

$$V = \pi_Y(\sigma_X(R_1 \times R_2))$$

The following expression shows the subsequent value of V, V_1, after an append-only transaction updating both R_1 and R_2.

$$V_1 = \pi_Y(\sigma_X((R_1 \cup A_1) \times (R_2 \cup A_2)))$$

Selection and projection both distribute over union, so the above expression simplifies as follows:

$$V_1 = \pi_Y(\sigma_X(R_1 \times R_2 \cup A_1 \times R_2 \cup R_1 \times A_2 \cup A_1 \times A_2))$$

$$= \pi_Y(\sigma_X(R_1 \times R_2)) \cup \pi_Y(\sigma_X(A_1 \times R_2)) \cup \pi_Y(\sigma_X(R_1 \times A_2)) \cup \pi_Y(\sigma_X(A_1 \times A_2))$$

$$= V_0 \cup \pi_Y(\sigma_X(A_1 \times R_2)) \cup \pi_Y(\sigma_X(R_1 \times A_2)) \cup \pi_Y(\sigma_X(A_1 \times A_2))$$

This algebraic simplification shows that V can be refreshed by computing the value of the last three expressions shown above, and then unioning the results to the stored copy of V (V_0). In practice, the query optimizer can be used to find the most efficient method available for computing these subexpressions. Since all these subexpressions are computed at the same time, performance advantages can be gained by optimizing all three together, and the technique described in [Sell86] can be applied to this problem.

If deletions as well as insertions occur in transactions, the differential update algorithm becomes slightly more complicated. One problem is that tuples in V may have been contributed by more than one source, since the projection operation can map multiple input tuples to the same value. If it appears that a tuple should be deleted from V, but V is stored with duplicates removed, it impossible to decide what action to take without totally recomputing V from the base relations. To overcome this difficulty without wasting disk space by physically storing duplicates, each tuple in V must contain a *duplicate count*, indicating how many potential sources could have contributed the tuple. With the duplicate count, when a tuple is inserted into V, if an identical value is already stored, then its duplicate count is incremented. Otherwise, the tuple is inserted with a duplicate count of 1. Similarly, the duplicate count of the stored value is decremented on tuple deletion. If the count becomes 0, the tuple is physically removed from V.

Extending the previous example, consider a transaction that inserts *and* deletes tuples from both R_1 and R_2. The new version of the view, V_1, is thus represented as follows:

$$V_1 = \pi_Y(\sigma_X(((R_1-D_1) \cup A_1)\times((R_2-D_2) \cup A_2)))$$

Using

$$R_1' = (R_1-D_1)$$
$$R_2' = (R_2-D_2)$$

we can rewrite the above as simply

$$V_1 = \pi_Y(\sigma_X((R_1' \cup A_1)\times(R_2' \cup A_2)))$$

Multiplying out this expression yields

$$V_1 = \pi_Y(\sigma_X(R_1'\times R_2' \cup R_1'\times A_2 \cup A_1\times R_2' \cup A_1\times A_2))$$

Expanding the $R_1'\times R_2'$ term of the above gives the following (the remaining terms are indicated by elipses):

$$V_1 = \pi_Y(\sigma_X((R_1-D_1)\times(R_2-D_2) \cup \cdots))$$

$$= \pi_Y(\sigma_X(R_1\times(R_2-D_2) - D_1\times(R_2-D_2) \cup \cdots))$$

$$= \pi_Y(\sigma_X(R_1\times R_2 - R_1\times D_2 - D_1\times(R_2-D_2) \cup \cdots))$$

Re-writing the second occurrence of R_1 as $(R_1' \cup D_1)$ gives

$$V_1 = \pi_Y(\sigma_X(R_1\times R_2 - (R_1' \cup D_1)\times D_2 - D_1\times(R_2-D_2) \cup \cdots))$$

Multiplying the second term through, and substituting R_2' for (R_2-D_2) leaves

$$V_1 = \pi_Y(\sigma_X(R_1\times R_2 - R_1'\times D_2 - D_1\times D_2 - D_1\times R_2' \cup \cdots))$$

If the operator $-$ is implemented as deletion and $\cup$ as insertion using duplicate counts as described previously, then the projection operation π will have the distributive property for *both* $-$ and $\cup$ [Blak86]. Applying these distributive properties to the expression above, we are left with

$$V_1 = \pi_Y(\sigma_X(R_1\times R_2))- \pi_Y(\sigma_X(R_1'\times D_2)) \cdots$$

$$= V_0- \pi_Y(\sigma_X(R_1'\times D_2))- \pi_Y(\sigma_X(D_1\times R_2'))- \pi_Y(\sigma_X(D_1\times D_2))$$

$$\cup\ \pi_Y(\sigma_X(R_1'\times A_2))\ \cup\ \pi_Y(\sigma_X(A_1\times R_2'))\ \cup\ \pi_Y(\sigma_X(A_1\times A_2))$$

As expected, the first term of this expression is V_0, the previous stored value of V. To update the stored copy of V so that its value becomes V_1, the remaining expressions must be evaluated, and either inserted into or deleted from V as required, maintaining the correct duplicate counts. (The differential view update algorithm presented here is slightly different than that given in [Blak86]. A discussion of the differences appears in Appendix A.)

2.2. The Deferred Refresh Algorithm

The algorithm described above is performed after every database update. However, in certain situations, it will be advantageous to save the sets of tuples inserted and deleted for a period of time, and *then* apply the differential update algorithm to the whole group. Given a method to compute the net changes (A_i-net and D_i-net) for each relation, R_i, for $1 \le i \le N$, over a period encompassing more than one transaction, incremental view maintenance can be done whenever desired. To refresh the materialized view, A_i-net and D_i-net must be calculated and then input to the standard differential view update algorithm.

A previously developed technique called *hypothetical relations* [Wood83] can be adapted to the purpose of computing A_i-net and D_i-net. The basic algorithm for implementing hypothetical relations is briefly described below. Efficient implementation of hypothetical relations to support deferred view maintenance will be discussed after the basic algorithm is presented.

2.2.1. Hypothetical Relations

Fortunately, we can find the net changes to R_i to use in deferred refresh using a modified hypothetical relation (HR) algorithm proposed in [Agra83]. The HR scheme uses three tables for each relation rather than one. Each relation has associated with it tables R, D and A, for base tuples, deletions and insertions, respectively. The data value of a tuple will simply be called "value." Each tuple will also have a unique identifier field "id." This yields the following schema for each relation:

R(id, value)
D(id, value)
A(id, value)

The true value of the relation (R_T) is $(R \cup A) - D$. The set difference operation "$-$" above has the normal meaning, based on all fields of the tuple, including id.

To append a tuple to R_T, a transaction inserts that tuple in A, placing the value of the system clock or other monotonicly increasing source in the id field. If duplicate-free semantics are desired, the system must ensure that the tuple is not already in $(R \cup A) - D$ before appending it to A. To delete the tuple from the relation, a copy of its value, including the id it had in R or A, is placed in D. To modify an existing tuple, its old value will be put in D, and its new value in A. When retrieving data from R_T, queries are processed against both R and A, and any tuples found are checked to make sure they are not already in D, (if they are, they are ignored).

Given this structure of the HR, the expressions for computing A-net and D-net from R, A and D as described above are the following:

A-net := A - D
D-net := D - A

After a view refresh that uses A-net and D-net, the files used to store the hypothetical relation will be reset as follows:

$R := (R \cup A) - D$
$A := \varphi$
$D := \varphi$

2.2.2. Efficient Implementation of Hypothetical Relations

The problem with the most straightforward implementation of hypothetical relations is that retrieving a tuple from R requires three disk accesses rather than just one, as in a standard relational database. To retrieve a tuple t from R using the HR scheme this way, an attempt must be made to read t from both R and A, and then D must be read to make sure that t has not been deleted.

Fortunately, a method developed in [Seve76] can be used to screen out most accesses to the differential file(s). In this method, a *Bloom filter* [Bloo70] is used for each differential file, consisting of an array of bits $B[1..m]$, with each entry initially zero. We assume that some subset of the fields of each record called the *key* uniquely identifies the record. For each record in the differential file, a hash function h mapping the key of a record to an integer in the range 1 to m is computed, and the corresponding entry in B is set to 1. Then, to test whether a record t is in the differential file, if $B[h(t.key)]=0$, t is not present; otherwise, if $B[h(t.key)]=1$, it might be present, so the differential file must be searched to see if it is there. Using the method proposed in [Seve76] one can design a Bloom filter with any desired ability to screen out accesses to records not present in the differential file by increasing the value of m.

As another measure to help speed up accesses to the differential file, A and D for each relation R will be combined into a single file, AD. An extra attribute "role" will be added to tuples in AD to indicate whether they are appended or deleted tuples. This storage structure will speed up the majority of updates, which modify existing records without changing the key. For example, if AD is maintained using a clustered hashing access method on the key, then when a tuple t is updated to t' without having its key changed, t' will hash to the same page as t. Thus, a maximum of only three disk I/Os will be required to update a single tuple t in R given the key for t. This update procedure is as follows:

I/O #1: Read the tuple.
(Check the Bloom filter to see if t could be in AD. If not, read t from R. Otherwise, read AD to see if it is there. If t is not in AD, read R. This might require 2 I/O's, but the probability can be made arbitrarily small by increasing m. Hence, we count only one I/O here for simplicity.)

I/O #2: Read the page where the new value of t (t') will lie in AD.
(Place both t and t prime on the page. The role values of t and t' are "deleted" and "appended" respectively.)

I/O #3: Write this page back to disk.

This is only one more I/O than necessary to perform this type of update using a standard relational data structure. If separate files for A and D were used, at least *five* I/O's would be required rather than three since R must be read, and A and D must both be read and written.

In the remainder of the paper, the sets of inserted and deleted tuples will still be referred to as A and D, even though they are stored in the AD table. It is assumed that AD will be partitioned to form A and D when necessary.

3. Performance Comparison

Each of the view materialization methods presented will have different performance characteristics. This section discusses the factors affecting performance and derives cost functions for each method for three different view models.

3.1. Models to be Analyzed

Given that view materialization can be done using query modification, or immediate or deferred view maintenance, we will determine the situations in which each scheme performs best. Three different models of the structure of view are considered:

model	*view structure*
Model 1	selection and projection of a single relation R
Model 2	natural join of two relations, R_1 and R_2, on a key field
Model 3	aggregates (e.g. sum, average) over a Model 1-type view

Only two types of operations will be considered in the models: updates to the base relations, and queries to the view. It is assumed that exactly k update operations, and q queries to the view will be run. For each model, a formula for the *average* cost per query, over all k updates and q queries, will be derived.

The relations involved have the following access methods:

base relations	
R, R_1	clustered B^+-tree on field used in view predicate
R_2	clustered hashing on join field
materialized view (V)	clustered B^+-tree on field used in view predicate
differential file (AD)	clustered hashing on a key field

Generous assumptions will be made for all view materialization schemes regarding how queries and other operations are performed using these clustered indexes. Since these performance benefits will be given to all algorithms, the results should not be biased toward any one scheme.

The following parameters are important to the analysis:

parameter	*definition*
N	number of tuples in relation
S	bytes per tuple
B	bytes per block
b	total blocks ($b = NS/B$)
T	number of tuples per page ($T=B/S$)
n	number of bytes in a B^+-tree index record
k	number of update transactions on base relation
l	number of tuples modified by each update transaction
q	number of times view queried
u	number of tuples updated between view queries ($u=kl/q$)
P	probability that a given operation is an update ($P=k/(k+q)$)
f	view predicate selectivity for Model 1
f_v	fraction of view retrieved per query
f_{R_2}	size of R_2 as a fraction of R_1
C_1	CPU cost to screen a record against a predicate in milliseconds (ms)
C_2	Cost in ms of a disk read or write
C_3	Cost in ms per tuple per transaction to manipulate A and D data structures in *immediate*

The default values of these parameters, which will be used unless stated otherwise, are as follows:

N	100,000	f	.1
S	100	f_v	.1
B	4,000	f_{R_2}	.1
k	100	C_1	1
l	25	C_2	30
q	100	C_3	1
n	20		

3.2. Model 1 Cost Analysis

In Model 1, the view is formed by projecting exactly one half of the attributes of tuples from R, and applying a predicate with selectivity f. Thus, the result will contain f times N tuples. The value we will measure for each view maintenance scheme is the average cost of a query that retrieves a fraction f_v of the tuples in the view.

3.2.1. Cost of Deferred View Maintenance Assuming Model 1

In deferred view maintenance, it is assumed that the view is refreshed every time it is queried. After the refresh is finished, the result of the query is computed. The average cost of a query to the view, which will be called $\text{TOTAL}_{\text{deferred1}}$, has several components. The first is the cost to read the result of the query from the copy of the view stored on disk. The second is the cost to refresh the view. The third is the cost to screen incoming and deleted tuples to see if they might affect the state of the view. Finally, the fourth is the cost to maintain the hypothetical relation(s). The average value of each of these costs are added together to get the average cost per query, $\text{TOTAL}_{\text{deferred1}}$. In summary,

$\text{TOTAL}_{\text{deferred1}}$ =
(cost to retrieve result of query from stored copy of view)
+ (cost to refresh the view)
+ (average cost per query to screen tuples to see if they affect view)
+ (average cost per query to maintain hypothetical relation(s))

We assume that no duplicates are formed by projecting half the attributes, so the view has fN tuples and $fb/2$ pages. A fraction f_v of the view is read during each access, requiring $ff_vb/2$ page reads, at a cost of C_2 each. One search of the B^+-tree will also be necessary to locate the position in the view to begin scanning. Since there are n bytes per index record, the height of the B^+-tree, not including the data pages, is determined as follows. The number of index records per page, and thus the index fanout, is B/n. There is one index record for each of the fN tuples in the view. Assuming as a simplification that all pages are packed full, the height of the view index (H_{vi}) is thus

$$H_{vi} = \lceil \log_{\lceil B/n \rceil} fN \rceil$$

Additionally, each tuple read from the view must be screened against the query predicate, at a cost of C_1, for a total cost per view access of $C_1 f_v fN$. Thus, the total cost C_{query1} to query a materialized view is

$$C_{\text{query1}} = C_2 \frac{ff_vb}{2} + C_2 H_{vi} + C_1 ff_v N$$

The next cost to consider is that for the hypothetical relation overhead. It is only necessary to measure the cost in excess of that required to perform normal base relation updates. As a simplification, the assumption is made that only tuples in R are updated, and never tuples in AD. The cost to maintain the HR for a single insertion into R in this situation is the following:

1. read the original tuple from R
2. read the page in AD where the modified tuple will be placed
3. write this page in AD

Step (2) is the only extra I/O required over using just a single table (R). The normal cost

to update R would be one read and one write, or $2C_2$, per tuple updated. If the cost of step (2) is averaged over all queries and updates, the cost per query to maintain the HR is at most the cost of one I/O (C_2) times the number of tuples update per view query (u). The total cost is likely to be somewhat less than this, however, since AD often has a small number of pages, and there are l tuples modified per transaction. The cost can be modeled more accurately using a function for estimating the number of pages touched when accessing k out of n records in a file occupying m disk pages. This function, which will be called $y(n,m,k)$, has been previously derived [Yao77] (See Appendix B for a description of y). The number of tuples in AD will be twice the number of tuples updated per view query ($2u$). The number of pages in AD will thus be $2u$ divided by the number of tuples per page (T). The number of pages in AD touched per transaction is thus $y(2u,2u/T,l)$. Averaged over q queries and k updates, the total cost of the extra accesses to AD is thus the following:

$$C_{AD} = C_2 \frac{k}{q} y(2u, \frac{2u}{T}, l)$$

Consider now the cost to refresh the view V once. This first involves the cost to read all of AD. Since u tuples are updated per view query, AD has approximately $2u$ elements. There are T records per page, so AD has $2u/T$ pages. Thus, the cost C_{ADread} of reading AD is

$$C_{ADread} = C_2 \frac{2u}{T}$$

Another cost is incurred to screen updates to see whether they have a chance of affecting the view. Recall that to screen incoming tuples to see whether they can affect a view, rule indexing is used in combination with a more stringent satisfiability test. For the view maintenance methods analyzed, it is assumed that the screening is performed as follows:

if
(1) a tuple breaks a t-lock for the predicate of view V, and
(2) the predicate for V with t substituted into it is still satisfiable,
then
a marker indicating this is placed on t.

In both the deferred and immediate view update algorithms, a tuple will be used to update a stored view V only if the tuple has a marker for V. A fraction f of the u tuples inserted into R per query will conflict with a t-lock set for V in step (1) above, and thus must be passed on to step (2). Step (1) has essentially no overhead, and step (2) costs C_1. Thus, the average overhead per query to screen tuples to see if they affect V is:

$$C_{screen} = C_1 fu$$

Also, approximately fu tuples per query will be inserted into and deleted from the view, respectively, for a total of $2u$ tuple updates. Each insertion or deletion from the view requires reading the B^+-tree view index, and reading and writing a data block. However, somewhat less than $2fu$ *pages* of the view may actually have to be updated during a refresh, since there may be more than one record per block in the view. Using the Yao function, since there are fN tuples and $fb/2$ blocks in the view, the number of view blocks accessed (X_1) is approximately

$$X_1 = y(fN, \frac{fb}{2}, 2fu)$$

Each access requires reading the index, reading and writing a data block, and writing a leaf-level index block (splits of internal index pages are infrequent, so their cost will be ignored as a simplification). This requires 3 I/Os, plus a number of I/Os equal to the height of the index on V (H_{vi}). Thus, the cost to refresh the view, $C_{def\text{-}refresh1}$, is as follows

$$C_{def\text{-}refresh1} = C_2 (3+H_{vi}) X_1$$

The following is the final expression for the cost per query to the view V using deferred refresh:

$$TOTAL_{deferred1} = C_{AD}+C_{ADread}+C_{query1}+C_{def\text{-}refresh1}+C_{screen}$$

3.2.2. Cost of Immediate Assuming Model 1

The cost per view access of performing immediate view maintenance, $TOTAL_{immediate1}$, is as follows:

$TOTAL_{immediate1}$ = (cost to query view)
+ (total cost to modify stored view)/(# of view accesses)
+ (total cost to screen tuples inserted into R
to see if they should enter view)/(# of view accesses)
+ (overhead per query to maintain A and D sets in
a data structure during transaction processing)

The cost C_{query1} to query the view is the same as for deferred view maintenance. The cost to update the stored view when a transaction modifies R, which will be called $C_{imm\text{-}refresh1}$, is computed much like $C_{def\text{-}refresh1}$. The difference is that approximately $2fl$ tuples in the view must be modified *once per transaction*, rather than modifying $2fu$ view tuples once per query. Since some of these $2fl$ tuples may lie on the same page, the number of view pages touched (X_2) can be estimated using the Yao function as follows:

$$X_2 = y(fN, \frac{fb}{2}, 2fl)$$

Similar to the case for deferred view maintenance, updating a tuple in V requires a B^+-tree search, the read and write of a data block, and the write of an index block. This requires $(3+H_{vi})$ I/Os for each view page touched, as before. Since there are k updates for every q queries, the average cost per query to update the view is:

$$C_{imm\text{-}refresh1} = \frac{k}{q} C_2 (3+H_{vi}) X_2$$

The cost C_{screen} to screen the kl tuples inserted into R is unchanged.

Finally, since immediate view maintenance must update the view after every transaction, the data structures used to maintain the A and D sets must be reset once per transaction. The overhead per query to do this, which will be called $C_{overhead}$, will be estimated as C_3 for each of the fl tuples in A and D, multiplied by the number of updates per query (k/q), i.e.

$$C_{overhead} = (C_3 2fl)\frac{k}{q}$$

This gives the following expression for the total cost of immediate view maintenance:

$$TOTAL_{immediate1} = C_{query1} + C_{imm\text{-}refresh1} + C_{screen} + C_{overhead}$$

3.2.3. Cost Using Query Modification Assuming Model 1

Here we consider using query modification rather than materializing the view in advance (this option will perform best in some circumstances, e.g. if the ratio of updates to queries is high). Three different methods for retrieving the view from R will be considered:

(1) a clustered (primary) index scan for which no extra tuples must be read (**clustered**)
(2) an unclustered (secondary) index scan (**unclustered**)
(3) a sequential scan of the entire relation (**sequential**)

Using a clustered index scan, the number of pages that must be read from R is equal to the size of the view, which is fb, times the fraction of the view retrieved, f_v. The number of tuples retrieved is ff_vN, and each of these tuples must be tested against the view predicate at a cost of C_1. Thus, for the clustered scan (1), the total cost to retrieve the view per access is

$$TOTAL_{clustered} = C_2 bff_v + C_1 Nff_v$$

Using an unclustered scan (2), a larger number of pages must be read from R. Searching for ff_vN tuples out of a total of b pages will require approximately $y(N,b,Nff_v)$ reads. The system must still test Nff_v tuples against the view predicate. Thus, the total cost for case (2) is

$$TOTAL_{unclustered} = C_2 \cdot y(N,b,Nff_v) + C_1 Nff_v$$

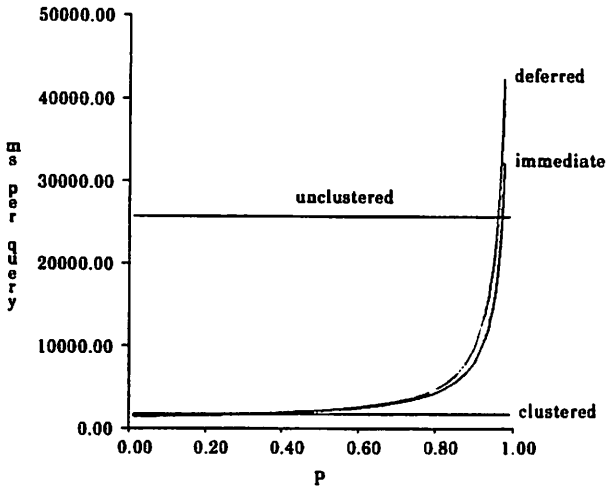
Using a sequential scan of the entire relation (3), all b pages must be read, and all N tuples must be screened against the view predicate, resulting in the following total cost:

$$TOTAL_{sequential} = C_2 b + C_1 N$$

3.3. Performance Results for Model 1

To indicate the differences in cost with respect to the probability P that an operation is an update, Figure 1 plots the total cost of deferred, immediate, clustered and unclustered vs. P for the standard parameter settings (sequential is not shown since it is off the scale). This setting of the parameters models a situation where the view contains 10,000 tuples, and each query retrieves 1,000 tuples. In this situation, query modification using a clustered access path has performance equal or superior to deferred and immediate. One would expect that clustered would perform well here since the number of pages that must be read is small when using a clustered index. The only advantage that deferred and immediate have over clustered is that there are twice as many tuples per page in the view compared with the base relation. However, the extra overhead paid by deferred and immediate to maintain the materialized copies of the view offsets this.

It is surprising that deferred and immediate view maintenance have almost identical cost under these circumstances. Once reason for this is that for low values of P, materialization methods have nearly equal cost for virtually any parameter setting. This occurs since for low update probability, a large fraction of the cost of both algorithms is for

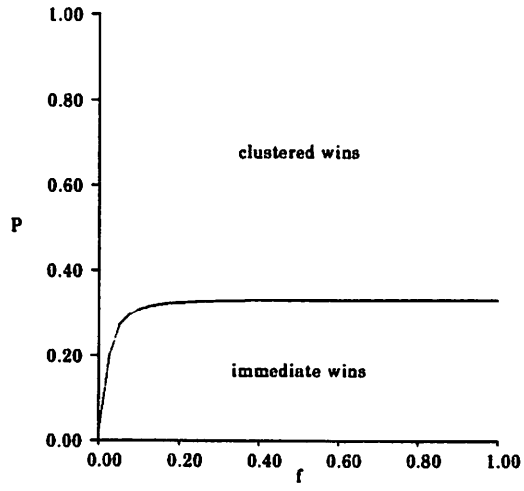


Model 1: Total cost of all algorithms vs. update probability P.
Figure 1

processing queries against the materialized view, and both algorithms do this the same way. Another cause of the close match is that the hypothetical relation overhead in deferred view maintenance counteracts the other advantages it holds over immediate view maintenance. If more than one disk is available, and I/O operations can be issued concurrently by a program, then it would be possible to significantly decrease the cost of maintaining hypothetical relations (e.g. by putting R, A and D on separate disks and reading from them simultaneously). This would give deferred maintenance an advantage over the immediate scheme for a wider range of parameter settings. However, these assumptions are not made in this paper since they would require extra hardware, and operating system functionality not readily available in all computer systems.

Assuming the view is maintained with a clustered index on a commonly used access path, the view materialization methods are significantly superior to query modification when only an unclustered access path is available on the base relation. This has implications for database design, since a materialized view could be clustered on one attribute, and the base relation on another. In this situation, a query optimizer could chose to process a view query in one of two ways, depending on the query predicate. If the predicate could be processed most efficiently using the clustered index on the base relation, query modification would be chosen to execute the query. Otherwise, the query could be processed against the materialized view, using the clustered view index as an alternate access path.

An interesting tradeoff among the algorithms centers around the parameters f, P, and f_v. To illustrate the relationship between these parameters, Figure 2 plots the region where each algorithm has lowest cost for different values of P and f, with f_v fixed at .1.



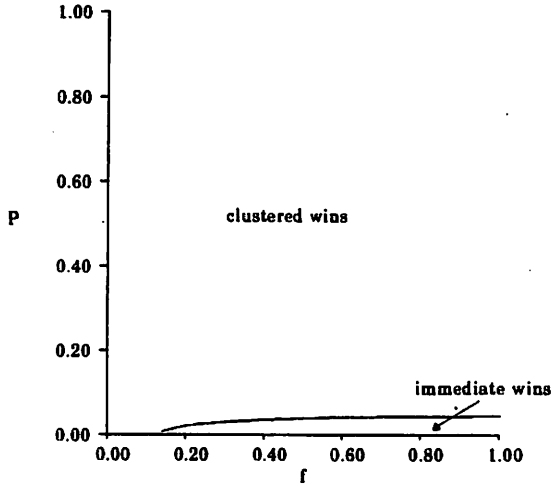
Model 1: Regions where each algorithm performs best for f vs. P (f_v=.1).
Figure 2.

Although deferred is never the most efficient algorithm under these parameter settings, larger values for f improve the performance of deferred relative to immediate view maintenance. This occurs due to the nature of the Yao function, combined with the fact that increasing f increases the size of A and D proportionately. Larger values of P tend to favor the algorithm with the least overhead per update transaction (i.e. query modification). Reducing the total fraction f_v of the view retrieved also tends to favor using query modification, since the overhead of the view maintenance schemes is independent of f_v, but the cost per query decreases with f_v. When the value of f_v is lowered to .01, as shown in Figure 3, clustered performs best over an even larger area. In Figure 4, C_3, the overhead per tuple for maintaining the A and D sets was increased from 1 to 2 ms, while setting f_v=.1. The affect of this change can be seen by comparing Figure 4 and Figure 2. The fact that deferred view maintenance now performs best in part of Figure 2 shows that the cost of the view materialization methods is very sensitive to the overhead for maintaining the A and D sets.

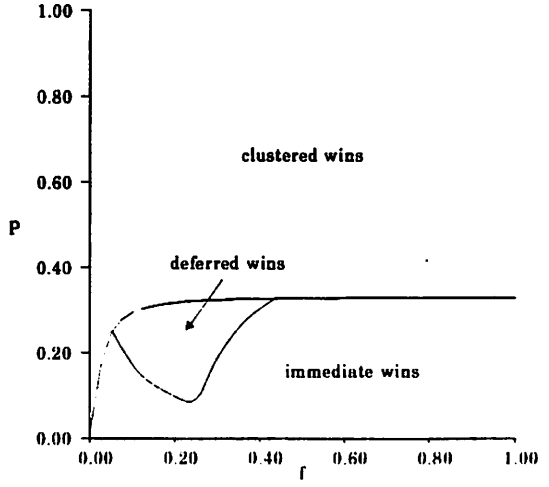
3.4. Model 2: 2-Way Join View

In this section, the performance of the different view maintenance algorithms is compared, assuming a more complex view model. The view V in Model 2 is a join of two relations, R_1 and R_2, where R_1 contains N tuples, and R_2 has $f_{R_2} \cdot N$ tuples. The definition of V is:

define view V (R_1.fields, R_2.fields)
where R_1.x = R_2.y
and C_f



Model 1: Regions where each algorithm performs best for f vs. P ($f_v = .01$).
Figure 3.



Model 1: Regions where each algorithm performs best for f vs. P ($C_3 = 2$, $f_v = .1$).
Figure 4.

The clause C_f in the view predicate restricts relation R_1 with selectivity f. We assume

that every tuple of R_1 that matches C_f joins to exactly one tuple in R_2, so V has $f \cdot N$ tuples total. Also, both R_1 and R_2 contain tuples of size S bytes, and only half the attributes of each relation are projected in the target list of the view definition. Thus, the tuples in V also contain S bytes each. The query and update activity assumed is the same as for Model 1, except that all updates are to R_1 rather than R (R_2 is never updated).

3.4.1. Cost of Deferred Assuming Model 2

For Model 2, the cost per query of doing deferred view maintenance is determined as follows:

$\text{TOTAL}_{\text{deferred2}}$ = (cost to read AD)
+ (cost to refresh view)
+ (cost to query view)
+ (cost per query to screen new tuples against view predicate)

The costs C_{AD} and C_{ADread} of updating and reading the HR, respectively, from Model 1 are unchanged for Model 2. The cost to refresh the view before it is queried (using deferred view maintenance), which will be called $C_{\text{def-refresh2}}$, will be determined as follows. To refresh V, the value of the following expression must be computed (the notation $V(X,Y)$ means the expression for V evaluated with X and Y in place of R_1 and R_2, respectively:

$$V(R_1,R_2) \cup V(A_1,R_2) - V(D_1,R_2)$$

The $V(R_1,R_2)$ term is already computed and stored as the previous version of the view (V_0). No terms containing A_2 and D_2 are shown since R_2 is never updated. Thus, only $V(A_1,R_2)$ and $V(D_1,R_2)$ must be computed. Recall that there is a clustered hashing index on R_2 that can be used as an access path to join tuples in A_1 and D_1 to R_2. The cost to join the A_1 and D_1 sets to R_2 is determined as follows: R_2 has $f_{R_2}N$ tuples and $f_{R_2}b$ pages, and there are u tuples in each of A_1 and D_1 at refresh time. Thus, the total number of pages that must be read from R_2 to perform these two joins is

$$X_3 = y(f_{R_2}N, f_{R_2}b, 2fu)$$

It is assumed that pages read for the first join stay in the buffer pool for the second.

There is also a CPU cost of C_1 for matching each of the $2u$ tuples in A_1 and D_1 with the joining tuple in R_2. Furthermore, for each joining tuple, a page must be read and written from the stored view. Using the Yao function, since the view has fN tuples of size S bytes, and a fraction f of the tuples in A_1 and D_1 join to exactly one tuple in R_2, the actual number of view pages that will be updated is approximately

$$X_4 = y(fN, fb, 2fu)$$

Each page update requires reading the B^+-tree index on the view, as well as reading and writing the data page, and writing the index leaf page (i.e. $3+H_{vi}$ I/Os). Thus, the total cost $C_{\text{def-refresh2}}$ to update the view every time it is queried is:

$$C_{\text{def-refresh2}} = C_2X_3 + C_1 2u + C_2(3+H_{vi})X_4$$

When the view is queried, both deferred and immediate view maintenance pay the same cost, C_{query2}. This consists of searching the view index to find the starting point, and then performing a clustered index scan to retrieve a fraction f_v of the view. This costs C_2 per page, and C_1 per tuple scanned. Summing the cost of the index search and scan yields the following expression for C_{query2}:

$$C_{query2} = C_2 H_{vi} + C_2 f_v f b + C_1 f_v f N$$

Both deferred and immediate view maintenance pay an average screening cost of C_{screen} per query to the view. Given $C_{def\text{-}refresh2}$, C_{query2}, and C_{screen}, we have the following expression for the total cost using deferred view maintenance assuming Model 2:

$$TOTAL_{deferred2} = C_{ADread} + C_{def\text{-}refresh2} + C_{query2} + C_{screen}$$

3.4.2. Cost of Immediate View Maintenance Assuming Model 2

The cost $TOTAL_{immediate2}$ of doing immediate view maintenance combined with rule indexing in Model 2 is

$TOTAL_{immediate2}$ = (cost per query to update view)
+ (cost to query view once)
+ (total overhead per query to maintain A and D sets)
+ (cost to screen new tuples against view predicate)

To find the cost per query $C_{imm\text{-}refresh2}$ of maintaining the materialized view, we must first find the cost to refresh the view after each transaction. The components of this refresh cost are the I/O cost of reading the pages of R_2 to which tuples in A_1 and D_1 join and reading and writing modified pages of V, plus the CPU cost of handling each tuple in A_1 and D_1. Since A_1 and D_1 both contain l tuples at the end of each transaction, and a fraction f of these match the view predicate and must be joined to R_2, the number of pages that must be read from R_2 is

$$X_5 = y(f_{R_2}N, f_{R_2}b, 2fl)$$

Each tuple in A_1 and D_1 joins to some tuple in R_2, so each causes one tuple to enter or leave V. The number of modified pages of V is

$$X_6 = y(fN, fb, 2fl)$$

Again, for each of these pages, the index on V must be read, the page must be read and written, and an index leaf page is written, requiring $3+H_{vi}$ page I/Os. There is also a CPU cost of C1 for handling each of the $2l$ tuples in A_1 and D_1. Averaging the per-transaction cost of updating V over k transactions and q queries, the estimated cost per query is as follows:

$$C_{imm\text{-}refresh2} = \frac{k}{q}(C_2 X_5 + C_2(3+H_{vi})X_6)$$

Given $C_{imm\text{-}refresh2}$ and C_{query2}, the following expression shows the total cost of immediate view maintenance using rule indexing, assuming Model 2:

$$TOTAL_{immediate2} = C_{imm\text{-}refresh2} + C_{query2} + C_{overhead} + C_{screen}$$

3.4.3. Cost Using Query Modification Assuming Model 2

Another important cost to measure is that to materialize a view directly from the base relations. A frequently used join strategy called *nested-loops* (or *loopjoin*) involves scanning one (outer) relation, and for each of its elements, searching the other (inner) relation to find all joining tuples. If an index is present on the join field of the inner relation, it can be used for the search.

It is assumed that the nested-loops join algorithm is used to join R_1 and R_2 in Model 2. R_1 will be the outer relation, and R_2 will be the inner one. Since there is a hash index on the join field of R_2, it will be used for the inner search. The assumption is made that pages of R_2 stay in the buffer pool throughout the computation of the join after they are read the first time. With the advent of very large main memories, this is reasonable since R_2 contains only $f_{R_2}NS$ bytes, which is approximately 1 Mbyte using the standard parameter settings. Under these assumptions, nested loop join has the following cost components, with the actual costs shown below:

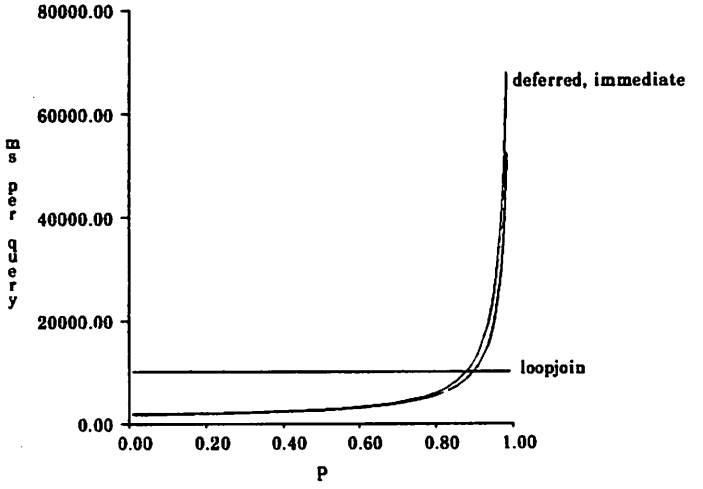
cost component	actual cost
read B^+-tree on R_1	$C_2\lceil \log_{\lceil B/n \rceil} N \rceil$
read part of R_1 using clustered scan	$C_2 f f_v b$
CPU cost to screen R1 tuples scanned	$C_1 f f_v N$
read pages from R_2 using hash index	$C_2 y(f_{R_2}N, f_{R_2}b, f f_v N)$
CPU cost to match R_1 tuples to R_2 tuples	$C_1 N f f_v$

Summing the above cost components gives the following formula $TOT_{loopjoin}$ for the total cost to compute the join using nested loops:

$$TOT_{loopjoin} = C_2\lceil \log_{\lceil B/n \rceil} N \rceil + C_2 f f_v b + C_2 y(f_{R_2}N, f_{R_2}b, f f_v N) + 2C_1 N f f_v$$

3.5. Performance Results for Model 2

The actual cost per query for deferred, immediate, and loopjoin using the standard parameter settings are plotted in Figure 5. The results for Model 2 are significantly different than to those for Model 1. When the view joins data from more than one relation, incremental view maintenance algorithms (deferred and immediate) perform better relative to query modification. By maintaining a materialized copy of the view, the query cost is greatly reduced, since each result tuple is stored on exactly one page. In effect, maintaining the view serves as an effective way of *clustering* related data on the same page. However, as P increases, the overhead for maintaining the materialized view overwhelms the advantage gained by clustering, so query modification becomes more attractive. Also, similar to Model 1, as the fraction of the view retrieved (f_v) is decreased, the advantage of query modification grows. This follows since lowering f_v reduces the query cost, while the amount of overhead paid by deferred and immediate algorithms for updating the view stays the same. An important special case to consider is when the view is large, and the queries read a small amount of data. This occurs, for example, using the standard EMPLOYEE and DEPARTMENT relations, and a view EMP-DEPT joining the two on the department-number field. The majority of queries in this situation might retrieve only a single tuple from EMP-DEPT. Also, updates usually change only one EMPLOYEE tuple. We modeled this example by setting $f = 1$, $f_v = 1/N$ and $l = 1$, and the results showed that query modification is superior to deferred and immediate under these circumstances for all values of $P \geq .08$. Thus, query modification is almost always the preferred method for answering small queries against large views. Other effects of varying f_v are shown using two figures. Figure 6 plots the areas where deferred view maintenance, immediate view maintenance and query modification using nested loops each have



Model 2: Actual cost per query for deferred, immediate, and loopjoin ($f_v = .1$).
Figure 5.

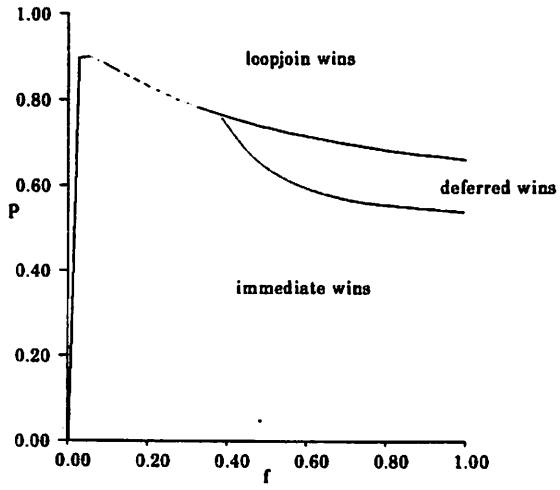
best performance for different values of P and f, with f_v set to .1. Figure 7 shows the same information with f_v set to .01.

3.6. Model 3: Aggregates Over Model 1 Views

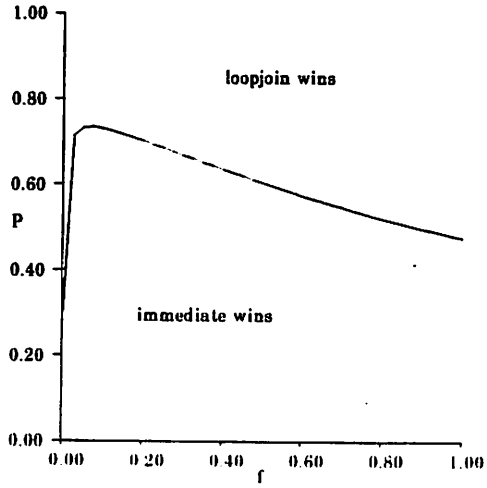
Aggregates such as sum, count and average are an often-used feature of database systems. Many aggregates (including all the ones listed above) can be incrementally updated as changes occur to the data from which they are computed. This is done by defining a *state* for the aggregate, functions for updating it in case of deletion or insertion of values in the set being aggregated, and a function for computing the current value of the aggregate from the state. The notion of incrementally maintaining aggregates is extremely attractive since the aggregate state can be read quickly because it normally requires less than one disk block of storage, while it often takes a large amount of I/O to recompute the aggregate from scratch. Thus, it would appear that an aggregate need not be used often to justify the expense of maintaining a materialized version of it.

To compare the value of maintaining aggregates vs. computing them from scratch, a modified view model (Model 3) is used, in which the views are simply aggregates over views of the same type as Model 1. In Model 3, the tuples for which the aggregate is computed do not need to be kept in a separate materialized view. Only the aggregate state must be stored.

For this model, a query to the view consists of simply reading the state of the aggregate. Using the deferred view maintenance scheme in Model 3, the cost $TOTAL_{deferred3}$ per query to the view is



Model 2: Regions where each algorithm performs best for f vs. P ($f_v = .1$).
Figure 6.



Model 2: Regions where each algorithm performs best for f vs. P ($f_v = .01$).
Figure 7.

$TOTAL_{deferred3}$ = (cost to read hypothetical database)

+ (cost to read the aggregate state)
+ (cost per query to update the aggregate state if necessary)
+ (cost per query of screening tuples to see if aggregate is affected)

The cost to read the hypothetical database is C_{ADread}, unchanged from Model 1. The cost to query the aggregate is the cost to read a single page, i.e.

$$C_{query3} = C_2$$

The cost to update the aggregate is the cost of one write times the probability that at least one tuple modified since the last query to the view lies in the set being aggregated (no read is necessary since the aggregate must be read to answer the query). There are $2u$ modified tuples in the hypothetical database per query to the view, and each has probability f of lying in the aggregated set. The probability that at least one of these tuples will lie in the aggregated set is equal to 1 minus the probability that none of the tuples lie in the set. Thus, the probability that at least one of the tuples lies in the set is $(1-(1-f)^{2u})$. This yields the following expression for the cost per query to update the view:

$$C_{def\text{-}refresh3} = C_2(1-(1-f)^{2u})$$

The final value of $TOTAL_{deferred3}$ is the following:

$$TOTAL_{deferred3} = C_{ADread}+C_{query3}+C_{def\text{-}refresh3}+C_{screen}$$

Using the immediate view update algorithm, the cost per query to maintain the aggregate is

$TOTAL_{immediate3}$ = (cost to read the aggregate state)
+ (cost per query to update the aggregate state if necessary)
+ (cost per query of screening tuples to see if aggregate is affected)

The cost to read the aggregate state is C_{query3}. The cost *per transaction* to update the aggregate state is C_2 times the probability that at least one tuple modified by the transaction lies in the aggregate, which is $(1-(1-f)^{2l})$. The cost per query to update the aggregate state is thus as follows:

$$C_{imm\text{-}refresh3} = \frac{C_2 k}{q}(1-(1-f)^{2l})$$

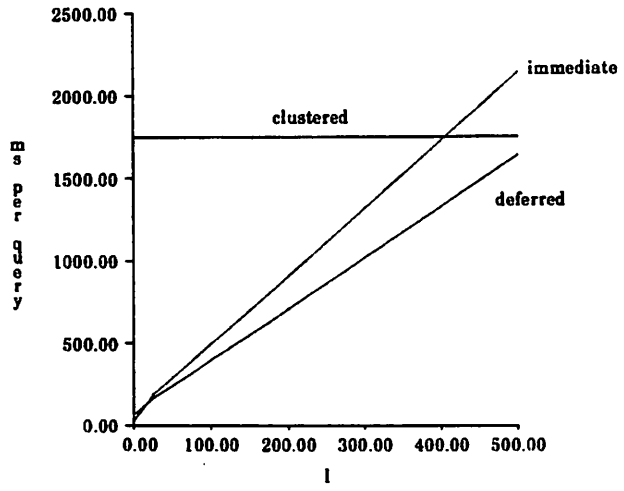
The cost of screening tuples is again C_{screen}, yielding the following expression for $TOTAL_{immediate3}$:

$$TOTAL_{immediate3} = C_{query3}+C_{imm\text{-}refresh3}+C_{screen}$$

$TOTAL_{deferred3}$ and $TOTAL_{immediate3}$ will be compared to the actual cost of recomputing the aggregate for each query using a clustered index scan, which is the same as $TOTAL_{clustered}$.

3.7. Performance Results for Model 3

To compare the total cost of using deferred view maintenance, immediate view maintenance, and a clustered index scan to compute an aggregate, the total cost of all three is plotted vs. l in Figure 8. Note that the most significant part of the curve is for small values of l, e.g. $l<100$. In this region, maintaining the aggregate costs only a small



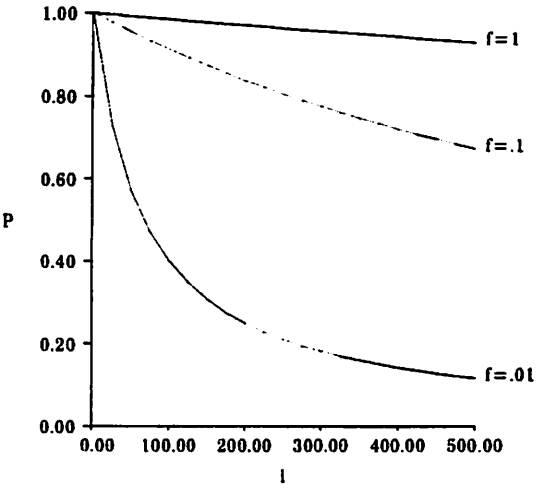
Model 3: Average cost of an aggregate query vs. l for deferred and immediate view maintenance, and standard processing using a clustered index scan.
Figure 8.

percentage as much as computing it from scratch.

To show the trade-off between a materialization algorithm and standard aggregate processing, Figure 9 plots curves for P vs. l showing where a clustered scan and immediate view maintenance have equal cost for different values of f (the fraction of the relation that is being aggregated). Query modification using the clustered scan performs best above each curve, and immediate maintenance performs best below. It is interesting to note that maintaining materialized aggregates is most attractive when the fraction of the relation being aggregate (f) is largest. Also, since realisticly l will probably be small, it is likely to be worthwhile to maintain materialized aggregates even for small values of f. Cost savings can be obtained by materializing aggregates in significantly more cases than for other views.

4. Conclusion

The performance analysis presented has shown that the choice of the most efficient view materialization algorithm is highly application-dependent. The results are most sensitive to the following parameters:



Model 3: Equivalent cost curves for immediate view maintenance, and standard aggregate processing using a clustered index scan. Above curves standard processing is best; immediate maintenance wins below.
Figure 9.

1. the total fraction of operations that are updates (P).
2. the selectivity factor of the view predicate (f).
3. the fraction of the view retrieved by each query (f_v).
4. the number of tuples written by each update (l).
5. the cost of maintaining the sets of inserted and deleted tuples (either in main memory, or in disk-based hypothetical relations).

Situations where P is high, f is high, or f_v is small, tend to favor not materializing the view at all. Rather, it is best to perform query modification, and retrieve the result from the base relations using a good access plan selected by the query optimizer. An important example of this is for large views (e.g. the EMP-DEPT view) and queries that always retrieve a single record. When this example was modeled using $f=1$, $l=1$, and $f_v=1/$(number of tuples in the view), we found that query modification nearly always outperforms materializing the view in advance.

If f_v is large, and P is not extremely high, then it becomes desirable to maintain views in materialized form. Higher values of P, f_v, and l favor deferred view maintenance over the immediate scheme. Conversely, if P is low, immediate view maintenance has a slight advantage over deferred maintenance. An interesting phenomenon observed was that immediate and deferred view maintenance have very nearly equal cost for Models 1 and 2, especially for low values of P. For higher P, reducing the cost of maintaining the net change sets (A and D) in either view materialization algorithm significantly improves performance.

An important question concerns selection of the best time to refresh the view (e.g. should the view be refreshed more often than done in deferred view maintenance, which refreshes only when absolutely necessary?). If the A and D sets in the hypothetical relational data structure use up all available disk space, then of course the refresh algorithm must be used to update the materialized view. However, if disk space is not a limiting factor, then waiting as long as possible between refreshes uses the least system resources. The reason for this is that *triangle inequality* holds for the Yao function, which is a main determinant of the cost of view maintenance. More precisely,

$$y(n,m,a+b) \leq y(n,m,a)+y(n,m,b).$$

for all a, $b>0$. The net result of this is that fewer I/O's are necessary to refresh a view only on demand than to refresh it multiple times for each query.

In cases where more than one materialized view draws data from the same hypothetical relation, it may be worthwhile to refresh all the views whenever it is necessary to read the contents of the A and D sets for the relation from disk, since this would eliminate the need to read the hypothetical database again. Also, if there is idle CPU and disk time available, it is likely to be useful to put it to work refreshing views asynchronously. This would improve the response time of view queries in some situations since the views would not have to be refreshed first. It would also allow update transactions to be completed more quickly than using immediate view maintenance, since the view would not have to be updated within the transaction. The evaluation of the usefulness of these possible optimizations is an interesting topic for future study.

As a final observation, we speculate that the best applications of incremental view update may not be related to processing queries against views, since this study has shown that query modification is still very effective. Rather, view materialization could be better employed where a complete copy of the answer to a query is always needed. For example, materialization could support conditions for complex triggers and alerters, as described in [Bune79]. Moreover, it could be used as a basis for a "window on a database" facility, where the result of a query would be displayed and updated in real time. Other applications of incremental view update may be forthcoming.

References

[Adib80] Adiba, M. E. and B. G. Lindsay, "Database Snapshots", *Proceedings of the International Conference on Very Large Data Bases*, October 1980, 86-91.

[Agra83] Agrawal, R. and D. J. DeWitt, "Updating Hypothetical Data Bases", *Information Processing Letters 16* (April 1983), 145-146, North Holland .

[Blak86] Blakeley, J. A., P. Larson and F. W. Tompa, "Efficiently Updating Materialized Views", *Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data*, Washington DC, May 1986, 61-71.

[Bloo70] Bloom, B. H., "Space/Time Trade-offs in Hash Coding with Allowable Errors", *Comm. of the ACM 13*, 7 (July 1970).

[Bune79] Buneman, O. P. and E. K. Clemons, "Efficiently Monitoring Relational Databases", *ACM Transactions on Database Systems 4*, 3 (September 1979), 368-382.

[Card75] Cardenas, A. F., "Analysis and Performance of Inverted Data Base Structures", *Comm. of the ACM 18*, 5 (May 1975), 253-263.

[Lind86] Lindsay, B. G., L. Haas, C. Mohan, H. Pirahesh and P. Wilms, "A Snapshot Differential Refresh Algorithm", *Proceedings of the 1986 ACM-SIGMOD International Conference on Management of Data*, June 1986, 53-60.

[Sell86] Sellis, T., "Global Query Optimization", *Proceedings of the 1986 ACM-SIGMOD International Conference on Management of Data 15*, 2 (June 1986), 191-205.

[Seve76] Severance, D. and G. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases", *ACM Transactions on Database Systems 1*, 3 (September 1976), 256-267.

[Ston75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification", *Proceedings of the 1975 ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, June 1975.

[Ston86] Stonebraker, M., T. Sellis and E. Hanson, "An Analysis of Rule Indexing Implementations in Data Base Systems", *Proceedings of the First Annual Conference on Expert Database Systems*, Charleston SC, April 1986.

[Wood83] Woodfill, J. and M. Stonebraker, "An Implementation of Hypothetical Relations", *Proceedings of the Ninth Very Large Data Base Conference*, Florence, Italy, December 1983.

[Yao77] Yao, S. B., "Approximating Block Accesses in Database Organizations", *Comm. of the ACM 20*, 4 (April 1977).

Appendix A

The method presented in [Blak86] for determining how to refresh the view when both deletions and insertions occur is slightly different than the one shown here, and is in fact not always correct. Using the scheme of [Blak86], the expression below would be used to refresh the view:

$$V_1 = \pi_Y(\sigma_X(R_1 \times R_2 \cup A_1 \times A_2 \cup A_1 \times R_2 \cup R_1 \times A_2 - D_1 \times D_2 - D_1 \times R_2 - R_1 \times D_2))$$

Using this expression can cause improper update of the duplicate counts. For example, suppose tuples t_1 in R_1 and t_2 in R_2 joined together to produce a result tuple in V_0. If a transaction deleted both t_1 and t_2, then the result of joining t_1 to t_2 would be deleted from V_0 three times, not just one as it should. This happens since t_1 is in both R_1 and D_1, and t_2 is in both R_2 and D_2. The formulation given in this paper (using $R_1' = R_1 - D_1$ and $R_2' = R_2 - D_2$) does not have this problem.

Appendix B

Given that there are n total records on m blocks, a formula giving the expected number of blocks that will be accessed to modify k records is as follows [Yao77]. Let C_a^b be the number of ways that b items can be selected from a items ($a \geq b$). If the number of records per block is $p = n/m$, then the formula giving the expected number of block accesses is C_k^{n-p}/C_k^n. An approximation to the above that is very close if the blocking factor is large (e.g. $n/m > 10$) is $m(1-(1-1/m)^k)$ [Card75]. The notation $y(n,m,k)$ is used to represent the Yao function.