## Title
Currency-Based Updates to Distributed Materialized Views



## Authors
Segev, A.
Fang, W.

## Publication Date
1989-06-01

LBL-27359

# Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

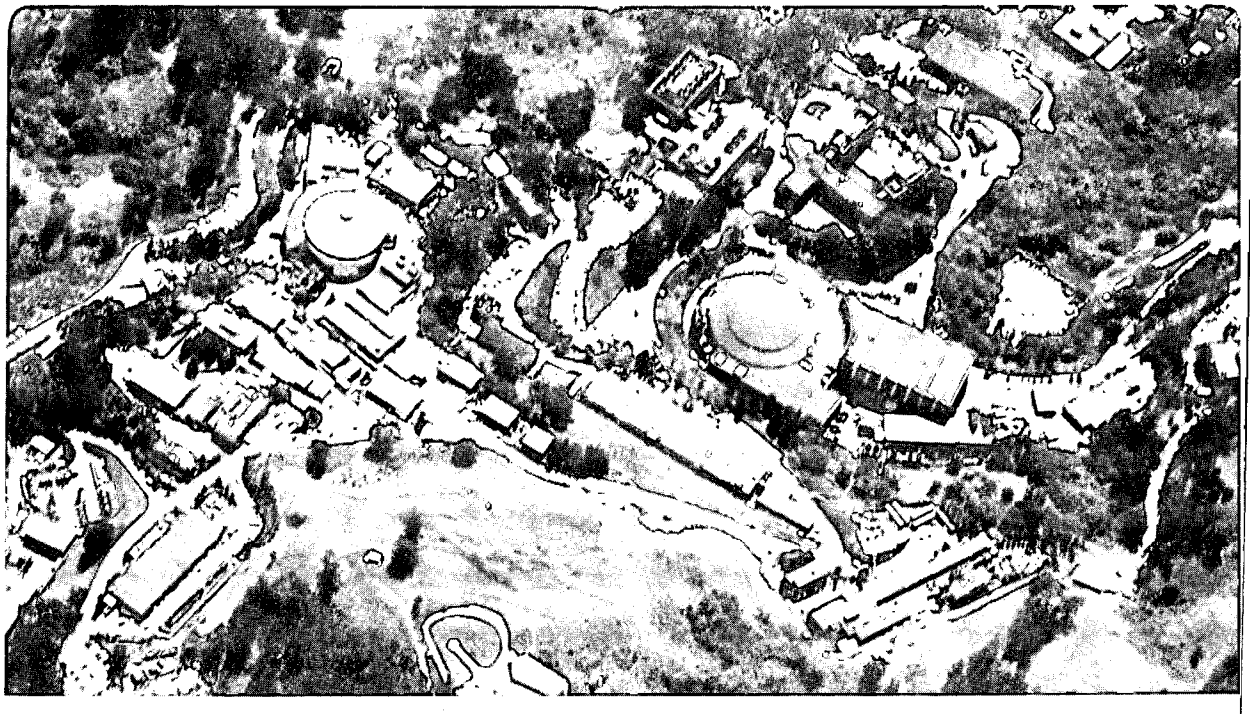
## Information and Computing Sciences Division

To be presented at the Data Engineering Conference,
Los Angeles, CA, February 2–5, 1990, and to be
published in the Proceedings

# Currency-Based Updates to Distributed Materialized Views

A. Segev and W. Fang

June 1989





Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

## DISCLAIMER

LBL-27359

# CURRENCY-BASED UPDATES TO DISTRIBUTED MATERIALIZED VIEWS

**Arie Segev† and Weiping Fang‡**

*† School of Business Administration*
*University of California at Berkeley*
*and*
*Computer Science Research Department*
*Lawrence Berkeley Laboratory*
*1 Cyclotron Road*
*Berkeley, California, 94720*

*‡ Department of Industial Engineering and Operations Research*
*University of California at Berkeley*
*and*
*Computer Science Research Department*
*Lawrence Berkeley Laboratory*
*1 Cyclotron Road*
*Berkeley, California, 94720*

## Abstract

In currency-based updates, processing a query to a materialized view has to satisfy a currency constraint which specifies the maximum time lag of the view data with respect to a transaction database. Currency-based update policies are more general than periodical, deferred, and immediate updates; they provide additional opportunities for optimization and allow updating a materialized view from other materialized views. In this paper, we present algorithms to determine the source and timing of view updates and validate the resulting cost savings through simulation results.

This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

## 1. Introduction

In a distributed environment, materialized views are a compromise between fully synchronized replicated data and single copies of data. Unlike synchronized replicated data, update transactions to the base data do not update the materialized data. After a base data transaction is committed, update transaction(s) to the materialized view(s) may be generated. The decoupling of base data transactions from updating materialized views raises three questions: when to update a materialized view? where to update it from? and how to perform the update?

In updating a materialized view, an obvious solution is to rematerialize it from the base data on which it is defined, but normally a differential update procedure (e.g., [LIND86]) is superior. Since it is possible that modified base data is irrelevant to the view, screen test procedures to determine its relevance have been devised [BUNE79, BLAK86]. Three general approaches to the timing of materialized view updates have been considered in previous research. The first approach is to update the view immediately after each update to the base tables [SHMU84, BLAK86], the second one defers the updates until issuing a query to the view [ROUS86, HANS87], and the third is to refresh the view periodically [ADIB80, LIND86, SEGE89a]. The tradeoff involved in choosing an approach is the currency of the materialized view vs. the cost of updating it.

Most of the studies of view management have been done for the case of a centralized DBMS. The subject of distributed materialized views has been addressed recently by [LIND86, KAHL87, SEGE89a]. These studies are concerned with "how" rather than "when" to update a materialized view. A recent work by [SRIV88] has modeled the "when" problem analytically for a single centralized view. The problem of quasi-copies which are similar to distributed materialized views is addressed in [ALON88]. In all the above works, the base data was assumed to be the source of updates to the materialized views. In [SEGE89b], currency-based updates policies are introduced and optimal update times are derived. The notion of currency (which measures the time lag of the view data relative to the base data) allows more flexibility in cost optimization including the possibility of updating a materialized view from another materialized view. in this paper, we extend the results of [SEGE89b] to determine the optimal update sources and timing for a set of distributed materialized

views defined on the same base data.

The paper is organized as follows. In Section 2, we introduce the problem definition, notation and assumptions. An optimization model and algorithms are presented in Section 3. Section 4 gives a detailed example of how to derive the input data used by the algorithms as well as how to apply them. Section 5 presents the results of simulation experiments that were done using DeNet [LIVN88]; the results demonstrate the cost savings that can be realized from our approach and algorithms. The paper is concluded in Section 6.

## 2. Problem Definition and Assumptions

Let $R$ be a base relation and assume that $n$ materialized† views $MV = \{V_i\}$, $i = 1, \cdots, n$, are defined over $R$. View $V_i$ is stored at view site $i$, and without loss of generality, we assume that all site numbers are distinct. Occasionally, we will use $v$ to mean $V_v$; it will be clear from the context. We are interested in finding an optimal view update policy for each $V_i \in MV$. The optimal policy is defined to be the one that minimizes the view update cost subject to a *currency constraint*. Currency constraints are defined as follows.

### View Currency

Let $\{State_B(t_i)\}$ be a description of the base table states at time points $t_i$. We assume that $t_i$ are expressed as integers and represent the lowest time granularity of interest. Similarly, let $\{State_v(t_i)\}$ be the state description of view $v$. We require that $State_v(t_i) \in \{State_B(t_j) \mid t_j \le t_i\}$, that is, the view state at time $t_i$ was a state of the base table at some time $t_j \le t_i$. The view currency at time $t_i$ is defined as

$$T_v^{(t_i)} = t_i - \max_{t \le t_i} \{t \mid State_v(t_i) = State_B(t)\}.$$

In practice one does not know that $State_v(t_i) = State_B(t)$ except for states that were reflected at view

† Unless stated otherwise, we will use the term 'view' to mean 'materialized view' in the remainder of this paper.

update times. Consequently, if the last update of the view was at time $t_u \leq t_i$ and that update reflected the base table state at time $t_j \leq t_u$, then the working definition of view currency is $T_v^{(t_i)} = t_i - t_j$. Informally, this definition means that the view data is at most $t_i - t_j$ time units 'old'. A currency constraint may be associated with a view and/or a query. Associating the constraint with the view implies that the view data has to satisfy it at all times. Associating the constraint with a query implies that data retrieved by the query has to satisfy it. In this work, we assume that the currency constraint is associated with queries (we denote it by $T_Q$).

**Query Processing and View Update Constraints**

When a query is to be processed at time $t_i$, $T_Q$ is satisfied by the view data if $T_v^{(t_i)} \leq T_Q$. We also assume that a query is answered from the view only, that is, if $T_Q$ is satisfied, the query is processed against the current state of the view; otherwise, the view is updated such that the new currency†, $T_v^{'(t_i)}$, is less than or equal to $T_Q$, and the query is processed against the new state. Since the view data is not synchronized with the base relation at the transaction level, $T_Q = 0$ or $T_v = 0$ should be interpreted as $0^+$, that is, the state of the view changes according to an immediate update policy (e.g. [SHMU84, BLAK86]).

Consider a view $\bar{v}$. Let $SV \subset MV$ be such that for each $v \in SV$, View_Predicate($\bar{v}$) = View_Predicate($v$) or View_Predicate($\bar{v}$) => View_Predicate($v$), and $T_v \leq T_Q$. The set $SV$ represents a set of views that can be used to update $\bar{v}$ such that the new currency of $\bar{v}$ will satisfy $T_Q$. See [FINK82] for the subject of identifying relationships between predicates. There are two advantages to having the option of updating a view from other views. First, it may be cheaper than using a base relation, and second, it frees the base relation processor (if the views are stored at other sites) from a portion of the view maintenance activity.

† In practice the various events, and the update process in particular, are not instantaneous. We assume that the new view currency is effective as of the beginning of the update process.

## View Update Policies

The foregoing discussion implies the following constraint on a view update policy: prior to processing a query $Q$, the currency of the view has to satisfy $T_Q$. Subject to this constraint there are several possible policies of timing the view update. These policies can be classified as follows:

P1: Periodical updates - view updates are done on a pre-determined cyclical basis.

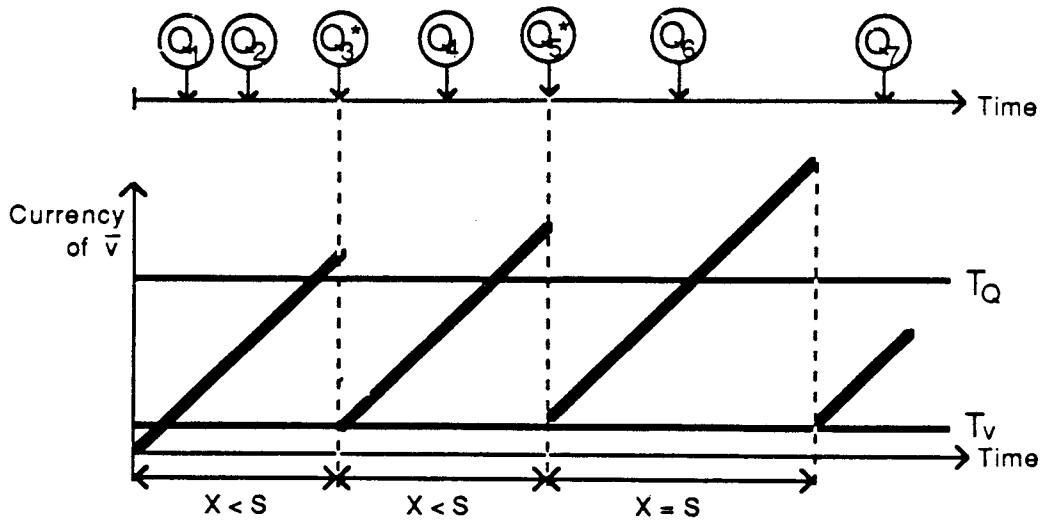
P2: On-Demand - view updates are done only at query processing time.

P3: Random Updates - view updates are done at random times.

P4: Hybrids - view updates are done according to combinations of the first three policies.

In [SEGE89b] we have derived optimal analytical results for hybrid policies of which P1 and P2 are special cases. The results of that paper are applicable to the case where the set $SV$ is given. However, a membership in $SV$ (relative to view $\bar{v}$) is qualified on both currencies and predicates, and therefore can not be determined a priori if the view update sources are not known. In this paper we are interested in determining optimal update policies for all the views in $MV$ given that a Periodical Or Demand (POD) policy is followed. Figure 1 illustrates the POD policy for updating $\bar{v}$ from $v$ where $T_v$ is constant (the results are applicable to random $T_v$ as well).

In this policy, an update to $\bar{v}$ is triggered by either of the following two events: (1) A query arrives and the currency of $\bar{v}$ is unsatisfactory; (2) The time from the last update is $s$. This policy provides a mechanism to balance the system's objective with the user's objective. In this policy, there are two types of updates; the first update type is triggered by a query, while the second type is clock-triggered when a cycle time elapsed. The cycle time is restarted after each update (either a query-triggered or a clock-triggered), and by changing $s$, one can control the cost of query-triggered updates. Note that the view currency is measured in time units relative to the states of the base table; Therefore, a higher currency value means that the view data is older. In the figure, we assumed that initially $\bar{v}$ is generated from the base table, and subsequently, is updated from $v$. The figure shows three updates; the first two updates are triggered by queries 3 and 5, and the third update is clock-triggered because $s$ time units elapsed from the second update. Queries that find the currency value below $T_Q$ do not




$Q_i$ → Arrival of query i

$Q_i^*$ → Query arrival that triggers a view update

X Time between update initiations

$T_Q$ Query currency requirement

$T_v$ Currency of update source

Fig. 1: The POD Update Policy.

trigger updates.

The goal of the analytical analysis is to derive the following results.

(1) Choose a $v \in SV \cup \{R\}$, and $s$ such that $\lim_{t \to \infty} \frac{\text{updating cost in } [0, t]}{\text{\# of query arrivals in } [0, t]}$ is minimized.

The above expression represents the average view update cost per query, and its minimization is a system's objective.

(2) We would like to minimize the expression in (1) subject to a user's response time constraint. The constraint is given as follows. let $UT_Q^{v,s}$ be the view update time for a query-triggered update (it is a function of $v$ and $s$). We require that $Pr\{UT_Q^{v,s} > H_1\} \leq H_2$, where $H_1$ and $H_2$ are user-provided threshold values.

The results (which are used in this paper) are summarized in Appendix 1. The underlying cost function is discussed next.

**Cost Functions**

The cost of updating a view is dependent on the particular algorithm, size of the data files, processing cost, communication costs, and the currency constraints. View update algorithms that were proposed in previous studies can be classified into two general categories: rematerializations and differential updates. An algorithm in the first category regenerates the view from the base table. in differential updates, only changes to the base table(s) since the last view update are processed against the view. There are two approaches to differential updates, one is to use a differential file (e.g. [SEGE89a]) and the other is to use the base table (e.g. [LIND86]).

Under the assumption that the size of relations and materialized views are stable over a relatively long period of time, we use an update cost function of the form $ax + b$. For rematerialization-type updates $a = 0$. For differential-type updates we assume that the cost is linear with respect to time $x$ (the precise definition of $x$ is given below). We feel that this function is an appropriate representation for a differential-file algorithm. For example, in [SEGE89a] the major cost components are a sequential backwards scan of the differential file, transmission of the differential tuples that pass a screening test, and updating the remote view. If the average selectivities and activity patterns are constant over a period of time, then in that period, the first two cost components are linear with respect to the time between view updates. The behavior of the third component is dependent on how the remote view update is done; it is possible that (even if the update volume is linear with time) a cost linear with time is an over-estimate because of Yao-function [YAO77] behavior† of view update cost.

In the remainder of the paper we will assume a differential-file-based view update algorithm. If we use $v = V_i$ to update $\bar{v} = V_j$, the update cost is $a_{ij}x + b_{ij}$, and the cost coefficients represent processing and communication costs; $x$ is the difference between the currencies of $\bar{v}$ after and before the

† The Yao-function is not related to the time between updates, but to the size of the view and number of records to be updated at any time. If the number of records to be updated is either very large or very small, a linear cost function is appropriate. Also, a very good piecewise-linear approximation of the Yao-function is given in [BERN81].

update. We also assume that the optimization procedure is static with respect to a given period of time, that is, during that period only one relation (or view) $v \in SV \cup \{R\}$ is used to update $\bar{v}$, and the average post-update currency of $\bar{v}$ is the same after each update from $v$. In this case, we use the time between updates as the value of $x$ in the cost formula. Note that previous studies are a special case, where the view currency after each update is a constant and equal to 0. In section 4, we show how to derive the coefficients $a_{ij}$ and $b_{ij}$.

## 3. A Model and Algorithms for Multiple View Updates

In this section, we formalize the optimization problem of determining optimal POD update policies for a collection of views. In order to simplify the subscripting we refer to the base relation as $V_0$. A collection of views $V_i$, $i = 1, \cdots, n$, are defined over $V_0$. Each $V_i$ is *derivable* from $V_i$ if the selection predicate of $V_i$, and all attributes of $V_j$ appear in $V_i$. View $V_j$ can be updated from $V_i$ if it is derivable from it; the update is *satisfactory* if the currency requirements of queries to $V_j$ are satisfied by the updated data; in that case we denote $V_i$ and $V_j$ as the source and target of the update respectively. For a particular target a source is satisfactory if the resulting update is satisfactory.

The following input date is used by the algorithms described below. The update cost coefficients $a_{ij}$ and $b_{ij}$ are used to calculate $c_{ij}(x) = a_{ij}x + b_{ij}$ which is the cost of updating $V_j$ from $V_i$, where $x$ is the time from the last update of $V_j$; if $V_j$ is not derivable from $V_i$, then $a_{ij}$ and $b_{ij}$ are $\infty$. A coefficient $d_i$ describes the cost rate of creating a differential file at view site $i$; this file is required if $V_i$ is used to differentially update at least one $V_k$, $k \neq i$; if $x$ is the time between two consecutive updates of $V_i$, then $d_i x$ is the differential file writing cost associated with an update of $V_i$. A directed weighted graph is given as $G = (V, A, C, D)$ where $V = \{i \mid i = 0, \cdots, n\}$, $A = \{(i, j) \mid i, j \in V$ & $V_j$ is derivable from $V_i\}$, $C = \{(a_{ij}, b_{ij}) \mid (i, j) \in A\}$, and $D = \{d_i \mid i \in V\}$. The node numbers in $G$ correspond to view numbers. Each element in $C$ is associated with the corresponding arc, and each element of $D$ is the weight of the corresponding node. Queries to $V_i$ arrive according to a Poisson process with a rate $\lambda_i$, and have a currency requirement $T_Q^i$.

In the algorithmic analysis, we distinguish between two cases -- acyclic $G$ and cyclic $G$. An acyclic graph represents the case where no two views are identical. This case results in the simpler algorithm (Algorithm 1 in Section 3.1). Section 3.2 extends Algorithm 1 to handle the case of identical views.

### 3.1. The Case of an Acyclic Graph

Algorithm 1 is of a greedy type. At each iteration an update source and timing is determined for the next view (the one for which the incremental cost is minimized); once this is done, the average currency of that view is calculated. Consequently, at each iteration there are two sets of views; the first set contains the views for which an update source and optimal update times have been determined, and the second set contains the rest ("unfixed") of the views. Note that given a set of fixed views the calculation for each of the unfixed views (based on the formulae in Appendix 1) is optimal, the overall procedure, though, is heuristic. The previous steps assume that the differential files at view sites are free; the final step of the algorithm evaluates this additional cost for each view that is used as an update source, and if this cost is greater than the savings realized by the target views (compared to being updated from the base table), the base table is fixed as the update source for those target views. If the differential file cost incurred by other applications at the view site, this final step should be omitted. The outcome from the above steps is dependent on the order in which the views are examined. A topological order [KNUT73] is used by the algorithm. The reason for a topological order is that all views from which view $V_j$ is derivable, are by definition, topologically before $V_j$ in the graph $G$. Therefore, when $V_j$ is being evaluated to determine its source of update, all views that can be used to update it have already been evaluated (including their average currencies $T_i$'s). A formal statement of the algorithm follows.

**Algorithm 1**

Step 1. Initialization: Relabels $V_j$'s in topological order; i.e., if $(i, j) \in A$, then $i < j$ in $V$. For each $j \in V$, find the set of immediate predecessors of $V_j$, $I_j = \{i \mid (i, j) \in A\}$. Set $T_0 = 0$.

Step 2. For $j \in V - \{0\}$ in topological order, do the following:

2.1 Find $i^*$ such that

$$\frac{a_{i^*j}}{\lambda_j} + \frac{b_{i^*j}}{\lambda_j(T_Q^j - T_{i^*}) + 1 - e^{-\lambda_j(s_j - (T_Q^j - T_{i^*}))}}$$

$$= \min_{\substack{i \in I_j \\ T_i < T_Q^j}} \left\{ \frac{a_{ij}}{\lambda_j} + \frac{b_{ij}}{\lambda_j(T_Q^j - T_i) + 1 - e^{-\lambda_j(s_j - (T_Q^j - T_i))}} \right\}.$$

2.2 SOURCE[$j$] = $i^*$.

2.3 $T_j = \dfrac{E(X_j(s_j))}{2} = \dfrac{1}{2}\left[T_Q^j - T_{i^*} + \dfrac{1 - e^{-\lambda_j(s_j - (T_Q^j - T_{i^*}))}}{\lambda_j}\right].$

Step 3. For $i \in V - \{0\}$ in topological order, do the following:

3.1 Find $J = \{j \mid \text{SOURCE}[j] = i\}$.

3.2 If $J \neq \varnothing$, find $J' \subset J$ such that for $j \in J'$

$$\frac{a_{ij} + \frac{d_i}{|J|}}{\lambda_j} + \frac{b_{ij}}{\lambda_j(T_Q^j - T_i) + 1 - e^{-\lambda_j(s_j - (T_Q^j - T_i))}}$$

$$> \frac{a_{0j}}{\lambda_j} + \frac{b_{0j}}{\lambda_j(T_Q^j - T_0) + 1 - e^{-\lambda_j(s_j - (T_Q^j - T_0))}}.$$

3.3 If $J' \neq \varnothing$, for $j \in J'$ set SOURCE[$j$] = 0 and $T_j = \dfrac{1}{2}\left[T_Q^j - T_0 + \dfrac{1 - e^{-\lambda_j(s_j - (T_Q^j - T_0))}}{\lambda_j}\right]$, $J \leftarrow J - J'$, and go to 3.2.

In the above algorithm, Step 1 topologically sorts the vertex set $V$ and for each vertex (view) $j$ in $V$ finds the set of its immediate predecessors, which, in other words, can be used to update $V_j$ in terms of their view expressions. Step 2 finds an update source that minimizes based on Appendix 1 the average update cost per query for the target view, and calculates its mean currency to be used in

case this view is used as an update source. Step 3 considers the cost of creating a differential file at each update source and may change the update source of views to the base table. The cost of the differential files is consedered after the update sources are fixed in Step 2, in order to account for cost sharing by multiple target views. When the algorithm terminates the update source of each $V_j$ is given by $V_{\mathrm{SOURCE}[j]}$.

As a final note, the algorithm assumes linear update cost functions. However, if the cost functions are piecewise linear as in the example in Section 4, the cost coefficients are different in different intervals. We then, for each possible update source (in $I_j$), calculate the mean update cycle length ($E(X_j(s_j))$) and use it to determine those cost coefficients ($a$'s and $b$'s) before the minimization in step 2.

### 3.2. The Case of a Cyclic Graph

The presence of identical views at different sites introduces cycles into the graph $G$. Each group of identical views leads to a maximal complete subgraph of $G$. Algorithm 2 is an augmentation of Algorithm 1 to handle this case. It first (Step 1) transform $G$ into $G'$ by shrinking every maximal complete subgraph of $G$ into a single node of $G'$ to make it acyclic. Then (Step 2) the nodes of $G'$ are evaluated in a topological order. If a node corresponds to a single node of $G$, Algorithm 2 does exactly the same as algorithm 1; Otherwise, it acts like a greedy algorithm for a minimum spanning tree, and repeatedly removes a node from the corresponding complete subgraph to the set of predecessors (see Step 2.3 below). The following is a formal statment of the algorithm.

**Algorithm 2**

Step 1. Initialization:

1.1 By shrinking every cycle into a single node, transform $G$ into $G'(V', A')$, where $V'$ is actually a partition of $V$; i.e., each element in $V'$ is a subset of $V$, and these subsets covering $V$ are disjoint. Since $A$ is transitive, every element in $V'$ constitutes a maximal (not maximum) complete subgraph of $G$. $A' =$
$\{(u', v') \mid u', v' \in V' \ \&\ (u, v) \in A \text{ for } u \in u', v \in v'\}$.

1.2 Sort $V'$ topologically according to $A'$. Let $V' = \{V_1', \cdots, V_m'\}$.

Step 2. For $l = 1, \cdots, m$, do the following:

2.1 Set $J \leftarrow V_l'$. $J$ is the set of views for which their source views are to be chosen from $I$.

2.2 Find $I = \{i \mid i \in V_k' \ \& \ (V_k', V_l') \in A'\}$. $I$ is the set of views that can be used to update views in $J$.

2.3 While $J \neq \emptyset$, repeat the following:

2.3.1 Find $i^* \in I$, $j^* \in J$, and $T_{i^*} < T_Q^{j^*}$ such that the following expression is minimized

$$\frac{a_{i^*j^*}}{\lambda_{j^*}} + \frac{b_{i^*j^*}}{\lambda_{j^*}(T_Q^{j^*} - T_{i^*}) + 1 - e^{\lambda_{j^*}(s_{j^*} - (T_Q^{j^*} - T_{i^*}))}}.$$

2.3.2 $J \leftarrow J - \{j^*\}, I \leftarrow I \cup \{j^*\}$.

2.3.3 SOURCE[$j^*$] = $i^*$.

2.3.4 $T_{j^*} = \frac{1}{2}\left[T_Q^{j^*} - T_{i^*} + \frac{1 - e^{-\lambda_{j^*}(s_{j^*} - (T_Q^{j^*} - T_{i^*}))}}{\lambda_{j^*}}\right]$.

Step 3. Same as Algorithm 1.

## 4. An Example

In this section, we illustrate the algorithm for the acyclic case using a simple example given below. The syntax of the view definition follows INGRES/SQL [RTI86].

*/* $V_0$: employee base relation containing 100,000 92-byte tuples stored at site 0. */*

*create table V0 (*

```
        EMP#        integer,
        NAME        char(20),
        ADDRESS     char(40),
        SALARY      money,
        JOB_CODE    char(4),
        DEPT        char(12),
        PROJ#       integer

);


/* V1: set of employees whose salary is greater than $50000. */
/* We assume 10,000 qualifying tuples stored at site 1; tuple width = 52 */

create view V1 as

        select EMP#, NAME, SALARY, JOB_CODE, DEPT, PROJ#
        from EMPLOYEE where SALARY > 50000;


/* V2: set of employees whose salary is greater than $40000. */
/* Wu assume 20,000 qualifying tuples stored at site 2; tuple width = 52 */

create view V2 as

        select EMP#, NAME, SALARY, JOB_CODE, DEPT, PROJ#
        from EMPLOYEE where SALARY > 40000;


/* V3: set of employees in Engineering Department with salary greater than $50000. */
/* We assume 1,000 qualifying tuples stored at site 3; tuple width = 52 */

create view V3 as

        select EMP#, NAME, SALARY, JOB_CODE, DEPT, PROJ#
        from EMPLOYEE
        where SALARY > 50000 and DEPT = 'ENGINEERING';
```

### 4.1. Deriving the Update Cost Coefficients

It is clear in the example that $V_0 \supset V_2 \supset V_1 \supset V_3$; therefore, $V = \{0, 1, 2, 3\}$ and $A = \{(0, 1), (0, 2), (0, 3), (2, 1), (1, 3), (2, 3)\}$.

To find out view updating costs ($C$ in our algorithm), we use the formulae of [SEGE89a] which are given in Appendix 2. The appendix also contains the linearization of the nonlinear expressions. The following parameter values are common to all updates:

$C_{I/O}$ = 0.025 sec; $B$ = 1024 bytes; $W_B$ = 8 bytes; $W_v$ = 52 bytes; $W_i$ = 56 bytes; $W_d$ = 4 bytes; $C_{comm}$ = 100000 bps.

**Costs for updating $V_1$ from $V_0$**

Further assumptions specific to this case:

$U = 20t/60 = t/3$; ($t$: time in seconds); $W_R = 92$; $N_R = 100{,}000$; $\alpha_s = 0.1$.

Based on those assumptions, we then have

$$DIO\,1 = 0.025 \times \frac{t}{3} \times \frac{92}{1024} = 0.7487 \times 10^{-3} t$$

$$DIO\,2 = 0.025 \times 2 \times \frac{t}{3} \frac{10000}{100000} \times \frac{92}{1024} = 0.1497 \times 10^{-3} t$$

$$DIO\,3 = 0.025 + 0.025 \times \begin{cases} t/30 & t/30 \le 39.06 \\ (t/30 + 78.125)/3 & 39.06 < t/30 \le 156.25 \\ 78.125 & 156.25 < t/30 \end{cases}$$

$$DIO\,4 = 0.05 \times \begin{cases} t/30 & t/30 \le 253.91 \\ (t/30 + 507.81)/3 & 253.91 < t/30 \le 1015.625 \\ 507.81 & 1015.625 < t/30 \end{cases}$$

$$DCOM = 8\left(\frac{t}{60} \times 56 + \frac{t}{60} \times 4\right)/100{,}000 = 0.08 \times 10^{-3} t$$

Adding the above cost components we get

$$DIO\,1 + DIO\,2 + DIO\,3 + DIO\,4 + DCOM$$

$$= \begin{cases} 3.478 \times 10^{-3}t + 0.025 & t \le 1171.875 \\ 2.923 \times 10^{-3}t + 0.676 & 1171.875 < t \le 4687.5 \\ 2.645 \times 10^{-3}t + 1.978 & 4687.5 < t \le 7617.19 \\ 1.534 \times 10^{-3}t + 10.44 & 7617.19 < t \le 30468.75 \\ 0.978 \times 10^{-3}t + 27.37 & 30468.75 < t \end{cases}$$

$$DIO\,5 = 0.7487 \times 10^{-3}t$$

Therefore,

$$(a_{0,1}, b_{0,1}) = \begin{cases} (3.478 \times 10^{-3}, 0.025) & t \le 1171.875 \\ (2.923 \times 10^{-3}, 0.676) & 1171.875 < t \le 4687.5 \\ (2.645 \times 10^{-3}, 1.978) & 4687.5 < t \le 7617.19 \\ (1.534 \times 10^{-3}, 10.44) & 7617.19 < t \le 30468.75 \\ (0.978 \times 10^{-3}, 27.37) & 30468.75 < t \end{cases}$$

and

$$d_0 = 0.7487 \times 10^{-3}.$$

Similarly, we get the following cost coefficients:

Updating $V_2$ from $V_0$ under assumptions $U = t/3$, $W_R = 92$, $N_R = 100{,}000$, $\alpha_s = 0.2$:

$$(a_{0,2}, b_{0,2}) = \begin{cases} (6.208 \times 10^{-3}, 0.05) & t \le 1171.875 \\ (5.097 \times 10^{-3}, 1.352) & 1171.875 < t \le 4687.5 \\ (4.542 \times 10^{-3}, 3.956) & 4687.5 < t \le 7617.19 \\ (2.319 \times 10^{-3}, 20.883) & 7617.19 < t \le 30468.75 \\ (1.208 \times 10^{-3}, 54.737) & 30468.75 < t \end{cases}$$

Updating $V_1$ from $V_2$ under assumptions $U = t/15$, $W_R = 52$, $N_R = 20{,}000$, $\alpha_s = 0.5$:

$$(a_{2,1}, b_{2,1}) = \begin{cases} (2.749 \times 10^{-3}, 0.025) & t \le 1171.875 \\ (2.194 \times 10^{-3}, 0.676) & 1171.875 < t \le 4687.5 \\ (1.916 \times 10^{-3}, 1.978) & 4687.5 < t \le 7617.19 \\ (0.805 \times 10^{-3}, 10.44) & 7617.19 < t \le 30468.75 \\ (0.249 \times 10^{-3}, 27.37) & 30468.75 < t \end{cases}$$

Updating $V_3$ from $V_0$ given that $U = t/3$, $W_R = 92$, $N_R = 100{,}000$, $\alpha_s = 0.01$:

$$(a_{0,3}, b_{0,3}) = \begin{cases} (1.022 \times 10^{-3}, 0.025) & t \le 1171.875 \\ (0.966 \times 10^{-3}, 0.09) & 1171.875 < t \le 4687.5 \\ (0.938 \times 10^{-3}, 0.22) & 4687.5 < t \le 7617.19 \\ (0.827 \times 10^{-3}, 1.067) & 7617.19 < t \le 30468.75 \\ (0.772 \times 10^{-3}, 2.759) & 30468.75 < t \end{cases}$$

Updating $V_3$ from $V_1$ given that $U = t/30$, $W_R = 52$, $N_R = 10{,}000$, $\alpha_s = 0.1$:

$$(a_{1,3}, b_{1,3}) = \begin{cases} (0.309 \times 10^{-3}, 0.025) & t \le 1171.875 \\ (0.254 \times 10^{-3}, 0.09) & 1171.875 < t \le 4687.5 \\ (0.225 \times 10^{-3}, 0.22) & 4687.5 < t \le 7617.19 \\ (0.114 \times 10^{-3}, 1.067) & 7617.19 < t \le 30468.75 \\ (0.059 \times 10^{-3}, 2.759) & 30468.75 < t \end{cases}$$

Updating $V_3$ from $V_2$ given that $U = t/15$, $W_R = 52$, $N_R = 20{,}000$, $\alpha_s = 0.05$:

$$(a_{2,3}, b_{2,3}) = \begin{cases} (0.351 \times 10^{-3}, 0.025) & t \le 1171.875 \\ (0.296 \times 10^{-3}, 0.09) & 1171.875 < t \le 4687.5 \\ (0.267 \times 10^{-3}, 0.22) & 4687.5 < t \le 7617.19 \\ (0.156 \times 10^{-3}, 1.067) & 7617.19 < t \le 30468.75 \\ (0.101 \times 10^{-3}, 2.759) & 30468.75 < t \end{cases}$$

Cost of differential files:

$$d_1 = 0.0423 \times 10^{-3}$$

$$d_2 = 0.0846 \times 10^{-3}$$

### 4.2. Applying Algorithm 1

We now illustrate how to use Algorithm 1 with the cost coefficients derived in Section 4.1. Assume that the query arrival rates are $\lambda_1 = 1/20$, $\lambda_2 = 1/10$, $\lambda_3 = 1/200$; also let $T_Q^1 = T_Q^2 = T_Q^3 = T_Q = 60$, and $s_1 = s_2 = s_3 = \infty$. Since $E(X_j(s_j))$ are less then 1171.875 for all $j$,

and thus fall into the first interval ([0, 1171.876]), we take $a$'s and $b$'s for that interval and omit the calculations of $E(X_j(s_j))$; otherwise we would pick $a$'s and $b$'s for the time interval containing $E(X_j(s_j))$.

After the initialization in Step 1 we get $V = \{0, 2, 1, 3\}$; and $I_2 = \{0\}$, $I_1 = \{0, 2\}$, $I_3 = \{0, 1, 2\}$.

In Step 2, for $j = 2$, the only possible source is $V_0$, and therefore SOURCE[2] = 0 and $T_2 = 35$. For $j = 1$, $\frac{a_{2,1}}{\lambda_1} + \frac{b_{2,1}}{\lambda_1(T_Q^1 - T_2) + 1} = 0.066$ is less than $\frac{a_{0,1}}{\lambda_1} + \frac{b_{0,1}}{\lambda_1(T_Q^1 - T_0) + 1} = 0.076$, and therefore SOURCE[1] = 2 and $T_1 = 22.5$. Similarly, for $j = 3$, we get SOURCE[3] = 1 and $T_3 = 118.75$.

In Step 3, for $i = 2$, $J = \{1\}$, and $J' = \varnothing$ because $\frac{a_{2,1} + d_2}{\lambda_1} + \frac{b_{2,1}}{\lambda_1(T_Q^1 - T_2) + 1} = 0.068$ is still less than 0.076. Thus the update source of $V_1$ remains unchanged. Similarly the source of $V_3$ also remains unchanged.

## 5. Simulation Experiments

The purpose of the simulation experiments reported in this section was two-fold. First, the view update decisions generated by the algorithms presented in this paper are based on a linear approximation of the cost function and an assumption that each view is updated from a single source. The latter assumption may significantly affect the error of the estimated cost because, in an actual system, it is possible that a view designated as an update source will not have a satisfactory currency at all update points (even though its average currency is satisfactory). In that case, either the source view is updated to a satisfactory currency (this may have to be done recursively for more than one view) or an alternative source is used (by the definition of currency the base table is always an acceptable source). In the simulation experiments, if an update source is a view, its currency is checked and if not satisfactory the update is done from the base table instead. We are interested in finding the ratio of the estimated update cost and the simulated update cost.

The second (and the more important) objective of the simulations was to evaluate the benefit of using views as update sources. The best measure of the benefit is derived from the simulation experiments; we ran simulations for cases where view update are done only with the base table as a source (the update times are determined based on the POD policy), and for cases (with the same parameter values) where the view update sources are determined by Algorithm 1. The ratio of the resulting costs indicates the benefit of updates from views. Next, we describe the specific details of the simulation experiments and their results.

The simulation model was built using DeNet [LIVN88] for a 4-node network representing the example of Section 4. The parameter values that were changed from one run to another were the Poisson arrival rates of queries and the currency requirements. All other parameters were fixed at the values given in Section 4. Each run lasted 100,000 seconds of simulation time. For each set of parameter values, Algorithm 1 generates the primary update sources for the case where views can be used as an update source (we refer to this case as "view-to-view updates"). The case of using only the base table for updates (referred to as "base-to-view updates") was handled according to Appendix 1. We report here the main results of the simulations. Figure 2 shows the ratio of the simulated total update cost to the estimated total update cost as a function of the query currency (the same query currency requirement was associated with all views). The inter-arrival times of queries were drawn from an Exponential distribution with a mean ($1/\lambda$) of 20, 10, and 200 seconds for views 1, 2, and 3 respectively. The graph of Figure 2 shows that the degree of underestimation is decreasing with the required currency.

Figure 3, demonstrates the more important result that it pays to allow view-to-view updates (even if the cost estimates used to determine the sources are not accurate). For all the simulations represented in Figure 3, the source for updating view 2 was the base table (this is the only choice because of the view predicates and the reason why its cost ratio is 1). View 2 was chosen as the update source for View 1 which in turn was chosen as the update source for View 3. The query arrival rates where the same as for Figure 2. The figure illustrates that substantial cost savings can be realized from using views as update sources (in this particular example up to 60% for View 3). In

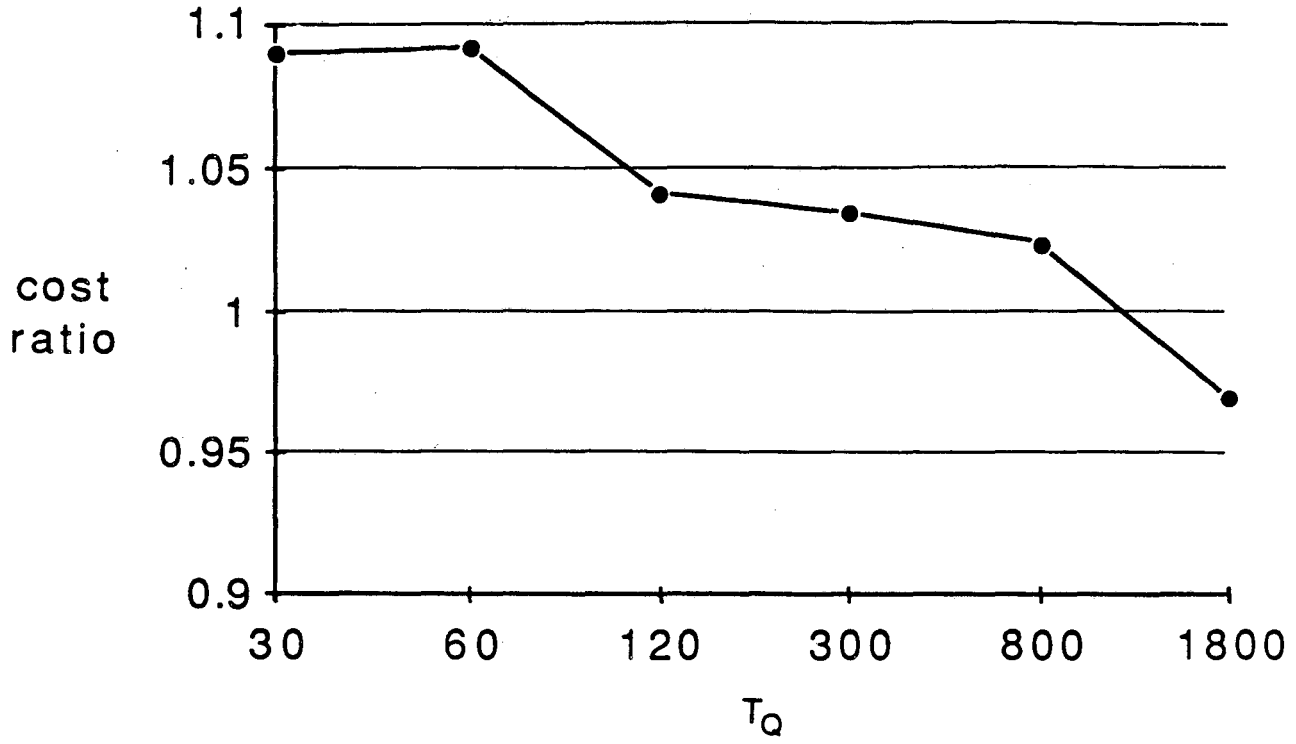



Fig. 2: Ratio of Simulated Cost to Estimated Cost.

general the cost savings increase with the query currency; this can be explained by the fact that the higher the value of $T_Q$, the higher the probability that a non-base table update source will have a satisfactory currency, and thus, the savings opportunity is realized by the target more frequently.

## 6. Summary and Future Research

Distributed materialized views can be a cost effective alternative to synchronized replicated data in many environments. In order for a DBMS to support materialized views, three problems have to be solved; the first is "when to update the view," the second is "where to update it from," and the third is "how to update the view." In this paper we have been primarily concerned with the first two problems, and introduced the concept of view currency.

By allowing queries on a materialized view to specify a currency requirement, a more powerful and flexible update policy results. If the currency does not imply immediate updates, it may be possible to update one materialized view from another rather than from a base table. This can reduce the

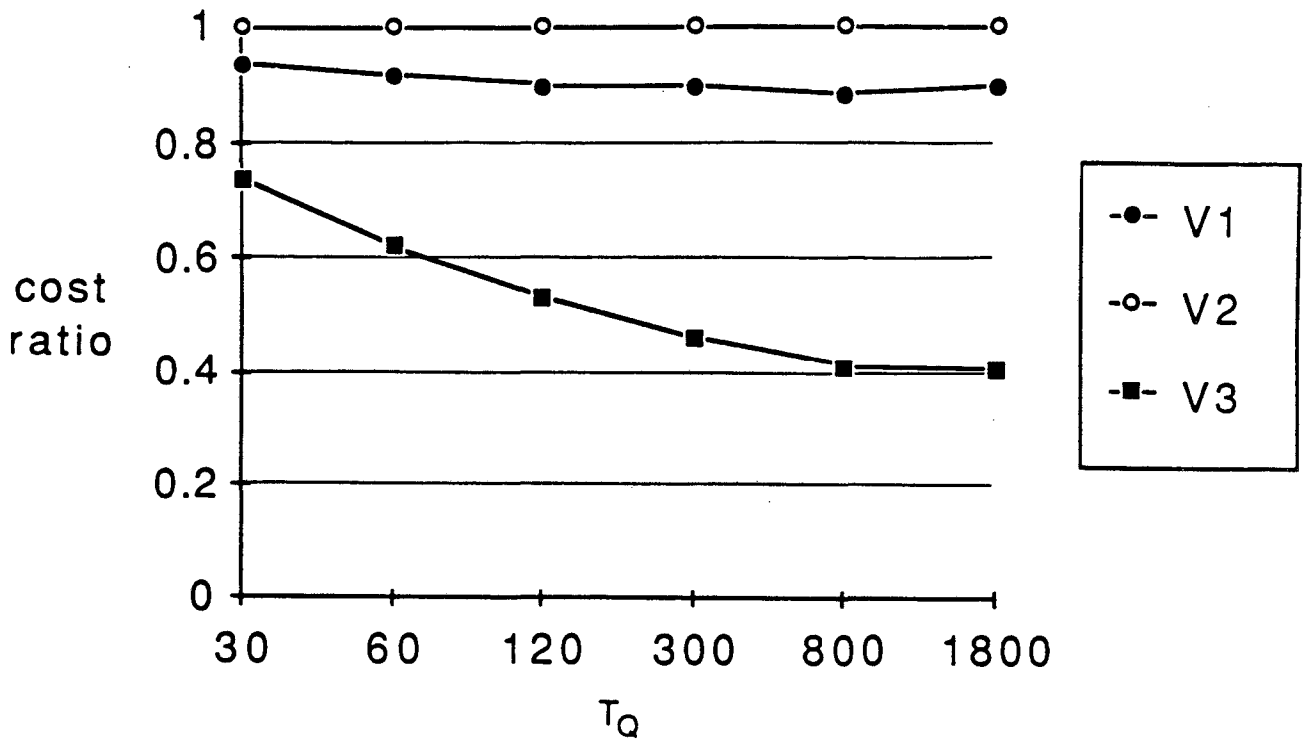



Fig. 3: A Comparison of View-to-View and Base-to-View Update Costs.

cost of maintaining distributed materialized views significantly, as well as lead to a further reduction in the interference with base table transactions. We have introduced an optimization model and algorithms to determine the optimal update sources and timing for a collection of views defined on common base data. A detailed example has been presented showing how to get the cost coefficients needed at the abstraction level of the algorithms; the example also demonstrated how the algorithms are applied. A DeNet simulation model was constructed in order to capture the cost of the algorithm's decision in more detail. In particular, the algorithm's cost estimation is based on a fixed update cost; however, in an actual system, if a currency is not satisfied for a particular update, then the source has to be changed to the case table; this 're-routing' is captured in the simulation experiments. The results of these experiments have demonstrated the potential cost savings from our approach and algorithms; in current work are extending the simulations to cover additional cases. Finally, it should be noted that at the algorithm level the base data can be a set of relations, and the views can be general. However, the details of deriving the linear cost coefficients (and their quality) will be different for general

Select-Project-Join views.

## ACKNOLEDGEMENT

We would like to thank Miron Livny for providing us with the discrete event network simulation language DeNet.

## APPENDIX 1

The following results were derived in [SEGE89b], for the POD policy.

**Summary of the Basic Model**

A base relation: $R$.

View to be updated: $\bar{v}$.

Views implied by or identical to $\bar{v}$: $SV$, $\bar{v} \notin SV$.

Cost of updating $\bar{v}$ from $v \in SV \cup \{R_i\}$: $c_v(x) = a_v x + b_v$, where $x$ is the time between view updates. ($a_v$ and $b_v$ are an abbreviated notation for $a_{v\bar{v}}$ and $b_{v\bar{v}}$ respectively.)

Currency of $v \in SV$: Uniformly distributed over the interval $[0, t_v]$.

Query (to $\bar{v}$) arrival: Poisson process with arrival rate of $\lambda$.

=> inter-arrival time: Exponential with mean $\frac{1}{\lambda}$.

Required currency for query $Q$: $T_Q$.

Objective: find a $v \in SV \cup \{R\}$ to be used in updating $\bar{v}$ with minimum cost subject to currency, policy, and response time constraints.

Note that the currency of the base table is 0.

**Theorem 1:**

The values of $s$ and $v$ that minimize the average cost per query are given by $s^* = \infty$, and $v^*$ is such that

$$\frac{a_{v^*}}{\lambda} + \frac{b_{v^*}}{\lambda\left[T_Q - \frac{t_{v^*}}{2}\right] + 1} = \min_{v \in SV \cup \{R\}} \frac{a_v}{\lambda} + \frac{b_v}{\lambda\left[T_Q - \frac{t_v}{2}\right] + 1}.$$

**Theorem 2:**

For any view $v$, the optimal $s$ value subject to update time constraints $H_1$ and $H_2$ is given by

$$s^* = \frac{1}{\lambda}\ln\frac{1 - e^{-\lambda t_v}}{(1 - e^{-\lambda t_v})e^{-\lambda\frac{H_1 - b_v}{a_v}} - \lambda t_v H_2 e^{-\lambda T_Q}}.$$

## APPENDIX 2

To calculate the cost of updating materialized views we use the following notation and cost expressions from [SEGE89a]:

DIO1: cost of reading tuples from the differential file

DIO2: cost of sorting the tuples after the screen test

DIO3: cost of accessing the $B^+$ tree

DIO4: cost of updating the data in the view table

DCOM: cost of transmitting over the network

DIO5: cost of creating a differential file at the update source

$B$: block size (bytes)

$C_{I/O}$: I/O cost (second/block)

$U$: number of tuples in the differential file

$W_R$: width (bytes) of each tuple for the base table

$U_s$: number of tuples that pass the screen test

$N_v$: number of tuples in the view table

$H_B$: height of a $B^+$ tree record at the view site ($\lceil \log_{B/W_B} N_v \rceil$)

$f(N, P, K)$: expected number of blocks fetched when accessing $K$ out of $N$ tuples in $P$ blocks [YAO77]

$W_B$: width of a $B^+$ tree record at the view site

$U_t$: number of tuples to be transmitted to the view site

$W_v$: width of of a view tuple

$W_i$: width of an insertion tuple

$W_d$: width of a deletion tuple

$C_{comm}$: transmission rate (bps)

The resulting cost expressions are:

$$DIO\,1 = C_{I/O}\,UW_R/B$$

$$DIO\,2 = C_{I/O}\,2U^s W_R/B$$

$$DIO\,3 = C_{I/O}(H_B - 1 + f(N_v, N_v W_B/B, U^t))$$

$$DIO\,4 = C_{I/O}\,2f(N_v, N_v W_v/B, U^t)$$

$$DCOM = 8(U_i^t W_i + U_d^t W_d)/C_{comm}$$

$$DIO\,5 = DIO\,1$$

Using the linearization of the Yao function from [BERN81],

$$f(N, P, K) \approx \begin{cases} K, & K \le \tfrac{1}{2}P \\ (K + P)/3, & \tfrac{1}{2}P < K \le 2P \\ P, & 2P < K \end{cases}$$

we get

$$DIO\,3 = C_{I/O}(H_B - 1) + C_{I/O} \begin{cases} U^t & U^t \le \tfrac{1}{2}N_v W_B/B \\ (U^t + N_v W_B/B)/3 & \tfrac{1}{2}N_v W_B/B < U^t \le 2N_v W_B/B \\ N_v W_B/B & 2N_v W_B/B < U^t \end{cases}$$

$$DIO\,4 = C_{I/O} \times 2 \begin{cases} U^t & U^t \le \tfrac{1}{2}N_v W_v/B \\ (U^t + N_v W_v/B)/3 & \tfrac{1}{2}N_v W_v/B < U^t \le 2N_v W_v/B \\ N_v W_v/B & 2N_v W_v/B < U^t \end{cases}$$

## REFERENCES


[ADIB80] Adiba, M. E. and B. G. Lindsay, "Database Snapshots," in *Proceedings of the International Conference on Very Large Data Bases*, October 1980, pp. 86-91.

[ALON88] Alonso R., Barbara D., H. Garcia-Molina, and S. Abad, "Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems," in *Lecture Notes in Computer Science*, vol. 303, edited by J.W. Schmidt, S. Ceri, and M. Missikoff, Springer-Verlag, 1988, pp. 443-468.

[BERN81] Bernstein, P. A., N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. "Query Processing in a System for Distributed Databases," in *ACM Transactions on Database Systems*, vol. 6, no. 4, December 1981.

[BLAK86] Blakeley, J. A., P. Larson and F. W. Tompa, "Efficiently Updating Materialized Views," in *Proc. of the ACM-SIGMOD Conf. on Management of Data*, Washington DC, May 1986, pp. 61-71.

[BLAK88] Blakeley, J. A. and F. W. Tompa, "Maintaining Materialized Views Without Accessing Base Data," in *Information Systems*, vol. 13, no. 4, 1988, pp. 393-406.

[BUNE79] Buneman, O. P. and E. K. Clemons, "Efficiently Monitoring Relational Databases," in *ACM Transactions on Database Systems*, vol. 4, no. 3, September 1979, pp. 368-382.

[FINK82] Finkelstein, S., "Common Expression Analysis in Database Applications," in *Proc. of the ACM-SIGMOD Intern. Conf. on Management of Data*, Orlando, FL, June 1982, pp. 235-245.

[HANS87] Hanson, E. R., "A Performance Analysis of View Materialization Strategies," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, May 1987, pp. 440-453.

[KAHL87] Kahler, B. and O. Risnes, "Extending Logging for Database Snapshot Refresh," in *Proceedings of the International Conference on Very Large Data bases*, Brighton, September 1987, pp. 389-398.

[KNUT73] Knuth, D. E., *The Art of Computer Programming, Vol. I: Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading Mass, 1973.

[LIND86] Lindsay, B. G., L. Haas, C. Mohan, H. Pirahesh, and P. Wilms, "A Snapshot Differential Refresh Algorithm," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, June 1986, pp. 53-60.

[LIVN88] M. Livny, "DeNet User's Guide," Version 1.0, Comp. Sci. Dept., Univer. of Wisconsin, Madison, 1988.

[ROUS86] Roussopoulos, N. and H. Kang, "Principles and Techniques in the Design of ADMS+/-," in *COMPUTER*, December, 1986, pp 19-25.

[RTI86] "INGRES/SQL Reference Manual," Relational Technology Inc., CA Release 5.0, UNIX, August, 1986.

[SEGE89a] Segev, A. and J. Park, "Updating Distributed Materialized Views," in *IEEE Trans. on Knowledge and Data Engineering*, (forthcoming).

[SEGE89b] Segev, A. and W. Fang, "Optimal Update Policies for Distributed Materialized Views," *Technical Report LBL-26104*, Lawrence Berkeley Laboratory, CA, 1989.

[SHMU84] Shmueli, O. and A. Itai, "Maintenance of Views," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Boston, 1984, pp. 240-255.

[SRIV88] Srivastava, J. and D. Rotem, "Analytical Modeling of Materialized View Maintenance Algorithms," in *Proc. of the 7th Annual Symposium on Principles of Database Systems*, Austin, Texas, 1988.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, San Jose, May 1975, pp. 65-78.

[YAO77] Yao, S. B., "Approximating Block Accesses in Database Organizations," in *Communications of the ACM*, vol. 20, no. 4, April 1977.

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
1 CYCLOTRON ROAD
BERKELEY, CALIFORNIA 94720