Title

PRECISION-TIME TRADEOFFS: A PARADIGM FOR PROCESSING STATISTICAL QUERIES ON DATABASES

Authors

Srivastava, J.
Rotem, D.

Publication Date

1988-05-01

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA, BERKELEY

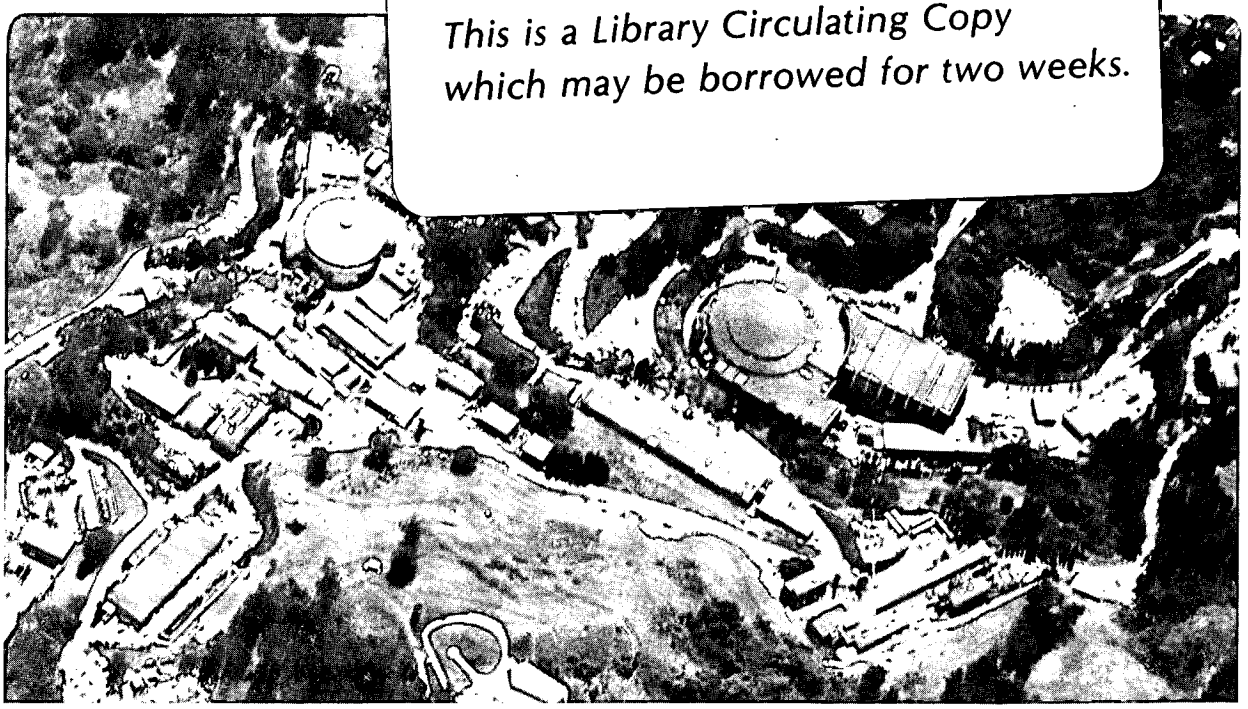
Information and Computing Sciences Division

Presented at the 4th International Working Conference on Statistical and Scientific Database Management, Rome, Italy, June 21–23, 1988

Precision-Time Tradeoffs: A Paradigm for Processing Statistical Queries on Databases

J. Srivastava and D. Rotem

May 1988



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

DISCLAIMER

LBL-24767

Precision-Time Tradeoffs: A Paradigm for Processing Statistical Queries on Databases

Jaideep Srivastava
Computer Science Division
University of California
Berkeley, CA 94720

Doron Rotem
Computer Science Research Department
Information & Computing Sciences Division
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

May 1988

Presented at the 4th International Working Conference on Statistical & Scientific Database Management, June 21-23, 1988

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

Precision-Time Tradeoffs: A Paradigm for Processing Statistical Queries on Databases

Jaideep Srivastava †
Doron Rotem

Computer Science Research
Lawrence Berkeley Laboratories
University of California
Berkeley, CA 94720

ABSTRACT

Conventional query processing techniques are aimed at queries which access small amounts of data, and require each data item for the answer. In case the database is used for statistical analysis as well as operational purposes, for some types of queries a large part of the database may be required to compute the answer. This may lead to a data access bottleneck, caused by the excessive number of disk accesses needed to get the data into primary memory. An example is computation of statistical parameters, such as count, average, median, and standard deviation, which are useful for statistical analysis of the database. Yet another example that faces this bottleneck is the verification of the truth of a set of predicates (goals), based on the current database state, for the purposes of intelligent decision making. A solution to this problem is to maintain a set of precomputed information about the database in a view or a snapshot. Statistical queries can be processed using the view rather than the real database. A crucial issue is that the *precision* of the precomputed information in the view deteriorates with time, because of the dynamic nature of the underlying database. Thus the answer provided is approximate, which is acceptable under many circumstances, especially when the error is bounded. The tradeoff is that the processing of queries is made faster at the expense of the precision in the answer. The concept of precision in the context of database queries is formalized, and a data model to incorporate it is developed. Algorithms are designed to maintain materialized views of data to specified degrees of precision.

1. Introduction

Conventional databases have focused mainly on the efficient execution of *transaction queries*, found typically in banking and airlines applications. Such queries have following properties: *(i)* they access small amounts of data, *(ii)* their answer is sensitive to each individual data item, and most importantly, *(iii)* they can be answered from the the basic

† This work was done while the first author was on leave from the C.S. Division, U.C. Berkeley.

database, i.e. the stored relations. An example is,

Does there exist any account number in the bank
with a negative balance ?

There are many applications where the overall characteristics of an entire dataset are required rather than individual data items. The required information does not exist in the database, and therefore has to be computed. An example is the need to obtain some statistical parameter, say average age of Caucasian males in California, from a demographic database. A typical query to derive this would be,

retrieve mean(*PERSON.age*)
where *PERSON.state* = *California*
and *PERSON.sex* = *male* and *PERSON.race* = *Caucasian*

Yet another example is the real-time controller of a manufacturing process, producing machine parts of specified dimensions. A typical rule for controlling the process would be,

If error(mean*(part_size)*) > mean-tolerance
or error(variance*(part_size)*) > variance-tolerance
then take some corrective action.

Applying such a rule requires computing the mean and the variance of *part_size*. Queries which carry out such computations are called *aggregate queries*. Answering an aggregate query requires extracting some *feature* from a number of data items, which in turn leads to accessing large volumes of data. Features, aggregates, etc., measure some characteristics of a set of data items, and thus are relatively insensitive to the value of individual data items. Hence, estimates of such characteristics are acceptable instead of their exact values. Conventional database systems incur prohibitive costs in providing accurate answers which are not essential. We believe that the cost of processing a query can be reduced significantly if an approximate answer is acceptable. We therefore conclude that efficient processing of aggregate queries is feasible if data management and query processing techniques are designed to accept *degree of precision* as a query parameter.

Researchers in the area of Statistical Databases [OLKE 86], [GHOS 85], [SHOS 82], [VITT 84], have faced the problem of data access bottleneck, since they had to deal with queries that calculated statistical aggregates of a set of data items. They realized the power of estimation, and introduced the statistical technique of *sampling* [COCH 53] as a

database operator. This has been very successful for statistical applications, dealing with numeric data. We propose an alternative approach to this problem. As we indicate in a later section, the usefulness of our approach lies in the fact that it can also be applied to the framework of *approximate reasoning*.

We propose the idea of *materialized view maintenance* as a technique to provide efficient support for the applications mentioned above. The stored database, for example the tuples of a relational database, is the *basic* database. All knowledge that can be obtained from it is called its *derived information*. If the entire derived information is stored at all times, processing queries that require it will be extremely fast. However, all possible derived information for any database is extremely large, making its storage infeasible. Our idea is to store the derived information that is interesting to a user, and process his/her queries on it. On top of the basic database, *views* [ULLM 82] are defined. A copy of each view, called its materialization (or *MatView*), is stored. Views are defined by the user and are specific to his application. They are defined by predicates which can be both logical and arithmetic.

Maintaining MatViews is a critical issue. Since the basic database is constantly updated the derived information changes as well, and the MatView, which is a part of the latter, has to be updated. A simple approach is to consider each individual change to the basic database and propagate its effects to all the MatViews that depend on it. However, this is prohibitively expensive due to the large number of updates to the basic database, and a correspondingly larger number of updates to the MatViews. Our approach to this problem is to maintain *approximate* MatViews. The definition of each MatView specifies not only the predicate defining it, but also a *degree of precision*, which determines its accuracy. The main idea is that updates to MatView need not be made as long as it stays within the bounds of the degree of precision attached to it. This approach reduces the total number of updates to the MatView drastically and brings it in a manageable range.

We present the tradeoff between precision and time as a paradigm for database management and query processing. MatViews are maintained at their specified degree of precision and are used to answer queries. The principal idea is that it costs less to maintain data that is less precise, and hence queries which do not require a very high degree of precision will not be forced to pay the price for it. This paradigm affects all components of a database system, namely the query language, data storage and access methods, query processing and optimization, and data distribution and replication.

In this paper we focus on the problem of maintaining derived information to its desired degree of precision. Specifically, we discuss the following: *(i)* a definition for precision of data and queries, *(ii)* materialized views as a mechanism for providing data with a specified degree of precision, and *(iii)* efficient automatic maintenance of views to their specified degree of precision. For clarity in exposition the examples in this paper define precision only in terms of the number of tuples of the database. However, our methods are applicable to other statistical parameters as well. Yet another application of these techniques is in the domain of approximate reasoning and deduction, which is based on *fuzzy logic* [ZADE 65]. This area has been rapidly gaining interest in the recent past.

A Real-Life Example: The idea of automatic maintenance of materialized views, defined in an application dependent manner, is especially attractive in a distributed environment. Each site stores only the views defined by the applications residing on it. The degree of precision of each view depends on the nature of the application.

We now present a real-life example *Fig. 1.1* to introduce the problem that we shall be formalizing and addressing in subsequent sections.

Consider the population data collected by the Census Bureau every time a census is carried out. Data about individuals is collected at the district level and is precise since it contains the individuals' exact age, height, weight, income, etc. Districts are grouped together to form cities, cities to form counties, counties to form states, and states to form the country. The lowest level of the hierarchy in *Fig. 1.1* may consist of a standard database, which is used for all sorts of local applications. However, the applications at the higher levels seek aggregate features of the data rather than the raw data itself. These various levels of geographic aggregation are logically equivalent to levels of information aggregation. From an information content viewpoint, it is not necessary to store any data at any level other than the lowermost. Higher levels of information can be derived from it. However, as we go upwards, the multiplicative increase in the volume of data makes it impossible to process raw data to obtain the required information on demand.

Our solution is to store, at each level of the hierarchy, the information required. Information at all levels but the lowermost, consists of *views*, which are defined in terms of the raw data, or other views, by ussing aggregation operations. Thus, we model the various levels of geographical aggregations, i.e. district, city, county, state and country, seen as a hierarchy of views defined on top of each other. The information that flows

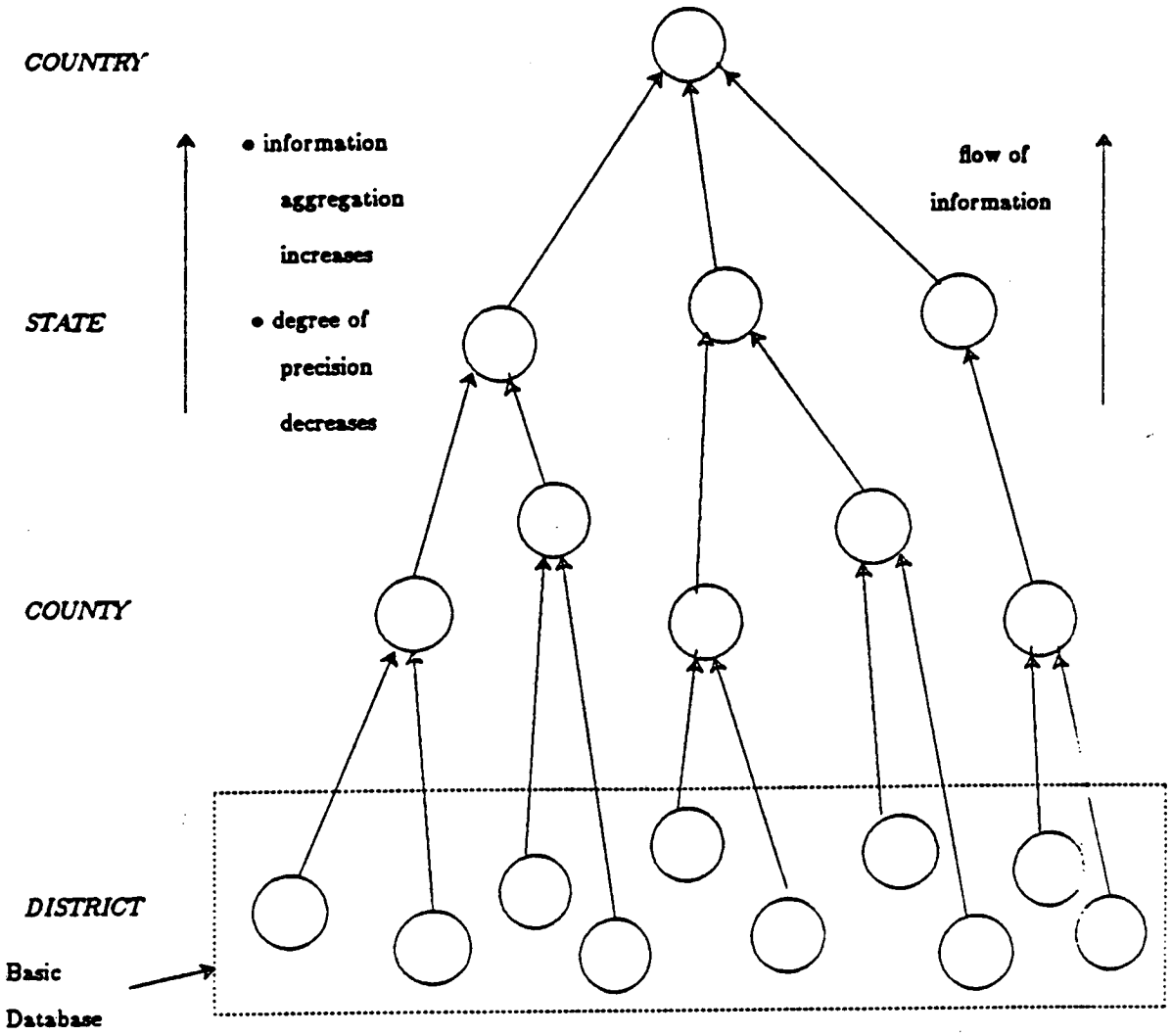


Fig. 1.1 *Hierarchy of Views in Census Data Collection.*

upwards is only the aggregate characteristics of the original data.

An important feature of the propagation of aggregates in such a manner is that the degree of precision required at higher levels is less than that at lower levels. The error introduced in the information in going up each level is modeled by the notion of *degree of precision* in our model. The advantage of the model presented is that it enables us to come up with efficient algorithms to maintain the various views in the hierarchy to their desired degrees of precision, at a minimum possible expense. Thus, updates need not propagate upwards, and flood the system, unless they are absolutely necessary.

This paper is organized as follows. Sections 3 and 3 discuss our definition of precision and the logical data model. Section 4 introduces a stochastic model for the problem of materialized view maintenance, and derives policies that maintain a MatView to its

specified degree of precision. Section 5 provides a summary of the issues that future research efforts need to address, and section 6 lists our conclusions.

2. Definition of Precision

This section discusses the definition of precision which shall be used as a running example to illustrates our view maintenance algorithms. However, the approach presented here does not depend on the definition of precision chosen. To make the discussion more concrete, we illustrate our algorithms by considering the higher level abstraction to be a counter. As an example we assume the counter under consideration is the size of a view. The theory developed, however, is applicable to any counter.

We use the following notation:

N_0: Size of the view after last refresh.

N: Size of the view at some later instant.

p: Degree of precision.

q: Degree of confidence.

Degree of precision of data: N has (p,q)-degree of precision, with respect to N_0, if the probability of the difference between N and N_0, expressed as a fraction of N_0, being less than $(1-p)$ is at least q.

Formally,

$$P\{\frac{|N - N_0|}{N_0} \leq 1-p\} \geq q$$

Note: Intuitively, p is the precision of N (w.r.t. N_0)and$q$ is the *degree of confidence* in the statement,

$$\frac{|N - N_0|}{N_0} \leq 1-p.$$

Usually, q will be fixed at some high value, say 0.99, so as to have a high degree of confidence in the answer obtained. The precision requirement of a query is defined as follows.

Precision Requirement of a Query: A query with precision (p,q), requires that it be processed using copies of data having degree of precision $\geq (p,q)$. †

† $(p',q') \geq (p,q) \equiv (p' \geq p)$ and $(q' \geq q)$.

Given a query and the set of data items it accesses, there are many ways to process it. The decisions that have to be made are, *(i)* in what order should the subtasks of the query be performed, *(ii)* what algorithm should be used to perform each subtask, *(iii)* how should a particular data item be accessed, and *(iv)* how should the data be buffered. If data items are replicated, the additional decision of which copy to access has to be made. A set of decisions, one along each dimension, called a *query plan*, is a unique way of executing the query.

Given the above definitions, we can identify two kinds of queries.

Let,

Q: A query.

(p,q): Precision requirement of Q.

qp_i: i^{th} query plan for Q.

$QP=\{qp_1, qp_2, ..., qp_n\}$: Set of possible query plans to execute Q.

$cost(Q, qp_i)$: cost of executing query Q using query plan qp_i.

Precision Query: Answer query Q, in the shortest possible time, such that each copy of data used to process it has degree of precision $\geq (p,q)$,

i.e.,

$$\min_{qp_i \in QP} cost(Q, qp)$$

$$subject\ to\ \text{for each data copy } D_i \text{ used, } (p_i,q_i) \geq (p,q)$$

Deadline Query: Answer query Q, using maximum precision data, such that it takes no more than T time units to process it.

i.e.,

$$choose\ qp \in QP\ such\ that,\ data\ copies\ with\ maximum\ possible\ precision\ are\ chosen$$

$$subject\ to\ \ 0 \leq cost(Q, qp) \leq T$$

3. Logical Data Model

The paradigm of tradeoff between precision and time is independent of the data model chosen. However, exemplifying our techniques requires us to make a specific choice. Further discussions assume the underlying data model to be the relational model with some extensions. Our choice of the model is motivated by the fact that it is both well accepted and easy to understand.

Any particular application finds only a subset of the database *interesting*. We further assume that such an interesting subset can be specified by a *view* definition.

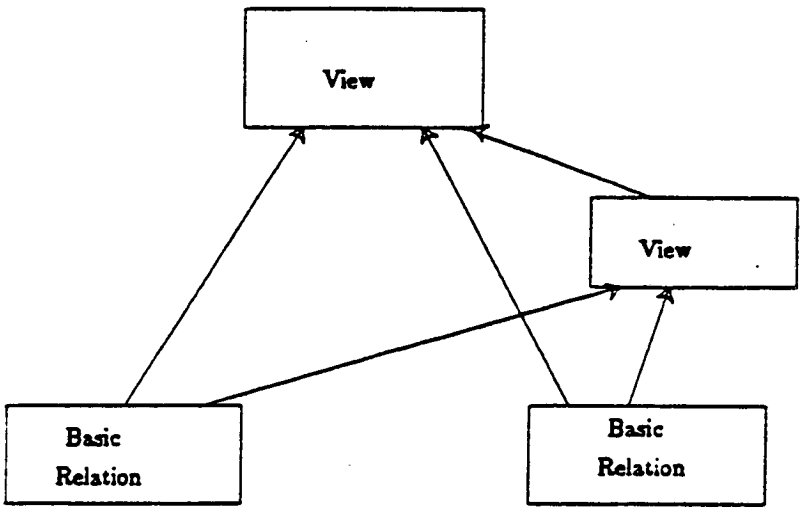


Fig. 3.1 *View Hierarchy.*

A view can be defined in terms of the base relations, in terms of other views, or both. This creates an entire hierarchy of views as shown in *Fig. 3.1*. We also assume that views are limited to those definable by the following operators.†

Relational Algebra: Relational algebra operations like *selection, projection, join, union, difference*, etc.

Example: *(A view definition using relational algebra):* Let,

$$EMP = (E.Id\#, E.name, E.age, E.sex, E.salary, E.med\_hist)$$

be a relation. A view defined using relational algebra is,

$$EMP\_HEALTH = \pi_{E.Id\#, E.age, E.med\_hist}(\sigma_{20K \leq E.salary \leq 40K}(EMP)).$$

Aggregation: Aggregation Operations like *count, sum, square-sum, data-grouping* etc.

Example: *(A view definition using aggregate functions):* Considering the same *EMP relation as above,*

$$SUM\_EMP = (sum(E.age), sum(E.salary))$$

Each view has a degree of precision attached to it, which is is one of the parameters specified during view-definition. The degree of precision depends on the requirements of the application defining the particular view. At all times, the degree of precision of the

† Our choice of the set of operators allowed for defining views is more powerful than the relational model, since it allows aggregation operations. Thus our assumption is not simplistic or unrealistic

view has to be maintained within the limits specified during view-definition.

4. Creation and Maintenance of Materialized Views

This section describes the maintenance of materialized views and develops a stochastic model for it. It also describes periodic policies for the maintenance of a view to its specified degree of precision.

4.1. Maintenance of Materialized Views

A materialized view (*MatView*) is a stored copy of the view which is created at the time of view definition. Any changes made to the base relations have to be reflected in the MatView. This is done by means of a periodic maintenance process or *refresh process*. The mechanics of MatView maintenance can be described by the files that exist in the system and the processes that manipulate them.

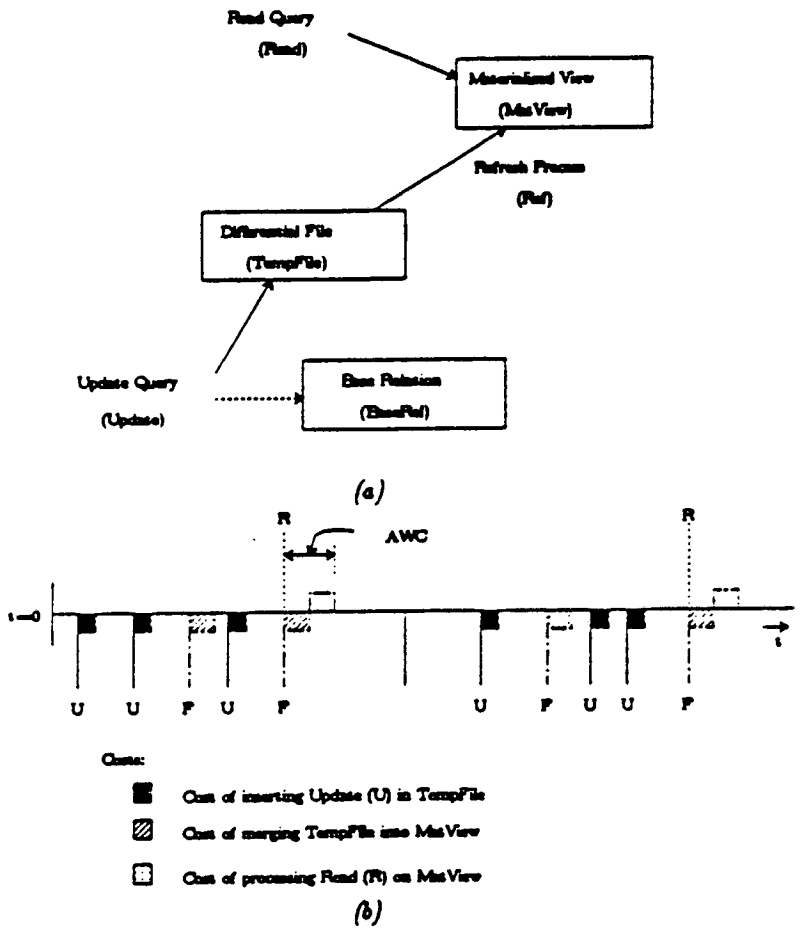


Fig. 4.1 *Mechanics of Materialized View Maintenance.*

Files: *Fig. 4.1(a)* shows the files existing in the system. The base relations from which the view is derived are stored in the file *BaseRel* and the materialized view in the file *MatView*. If MatView is refreshed as soon as a change is made to BaseRel, then these are the only files required. However, deferring the refreshing of MatView has advantages of being more efficient under certain conditions, [ROUS 86, HANS 87]. Thus another file, the *TempFile*, is required to store the changes between successive refreshes to the MatView.

Processes: There are three kinds of processes in the system which are of interest to us, as shown in *Fig. 4.1(a)*. First are Read Queries (R), that are directed to the MatView and processed using it. Second are the Update Queries (U), that can be directed either to the MatView or to the BaseRel. These are handled by making appropriate changes to the BaseRel † and also recording it in the TempFile. If the Update is an insert, delete, or change on the MatView, the appropriate action is taken on the BaseRel and the change is recorded in the TempFile. Handling Updates directed to BaseRel requires more care. Only some of these Updates affect MatView and are the only ones that have to be accounted for. This is done by *screening* each Update to the BaseRel against a *filter* (i.e. the logical predicate defining the view), to determine if it does indeed change the view too, [BLAK 86]. The ones that do so are recorded in the TempFile in addition to being changed in the BaseRel. From now on we will only consider those updates that affect the MatView. The third kind of processes are the Refreshes (F), which are executed periodically and whose function is to refresh the MatView using the contents of TempFile and bring it up-to-date with respect to BaseRel. This involves changing all tuples that appear in TempFile. No changes have to be made to BaseRel when a Refresh occurs, since the former is always up-to-date. Also, every occurrence of Refresh updates MatView with the contents of TempFile, and the latter is deleted.

4.2. A Stochastic Model of View Maintenance

Operation of the materialized view maintenance mechanism is modeled by the arrival of stochastic processes. The arrival of a Read or an Update, or the creation of a Refresh is the arrival of a job for service, and the times required for their execution are their service times, respectively. *Fig. 2.1(b)* gives a pictorial representation of the model.

† Changes to BaseRel are done anyway and *not* because of maintaining materialized views.

Nature of the Processes: The Reads and Updates are both assumed to be stochastic processes. The Updates are assumed to have the Poisson † distribution [ROSS 85] while the Reads can have any distribution. The arrival rate of Updates is λ_U. Thus,

Read $\approx$ G (general distribution).

Update $\approx$ Poisson (λ_U).

Costs associated with processes: The cost associated with a process is the number of disk accesses made during its execution. For a fair comparison of materialized view maintenance algorithms we have developed a cost model that measures precisely the *extra overhead* that a database system incurs in supporting the materialized view, i.e. maintaining MatView and TempFile. Consider the effort required to handle insertions/deletions to BaseRel. This would be required even if there were no materialized views, and thus we do not include its cost. There is a cost associated with *screening* each tuple to decide if it affects the view; and if it does, there is the additional cost of inserting it in the TempFile. Given below are the costs associated with the three processes in our model.

Cost of Read: The cost of accessing a single record † from MatView.

Cost of Update: If the update is to MatView, the only relevant cost is of recording it in TempFile. When the update is to BaseRel, it also has to be screened to see if it affects MatView. From our point of view only the *relevant updates* [BLAK 86], i.e. the ones that affect the view, are of interest. Since screening is done in main memory, its cost is negligible and is henceforth ignored. Thus, the cost associated with an Update is that of recording it in TempFile regardless of whether it is directed to MatView or BaseRel.

Cost of Refresh: The cost of updating MatView with the contents of TempFile to bring the former up-to-date.

The work done in maintaining the files MatView and TempFile is exclusively for the purpose of view materialization. Thus the extra cost incurred by the database system should be borne by the queries made to the view. Hence, the cost of a Read is augmented by the expected cost of handling all the Updates since the previous Read. Table 4.1 lists the various parameters of the model.

---

† The assumption of arrival processes having a Poisson distribution is widely used by almost all researchers in the analysis of stochastic models.

† For simplicity we assume exactly one record is accessed by a Read.

Symbol	Meaning
$Read(R)$	Read Process (Query)
$Update(U)$	Update Process (Query)
$Refresh(F)$	Refresh Process
$\lambda_R, \lambda_U, \lambda_F,$	arrival rates of processes R, U, F respectively
p_U, p_F	probabilities of arrival of U, F respectively
N_0	value of counter (MatView size) at last refresh
N	value of counter (MatView size) at a later instant
p	degree of precision
q	degree of confidence

Table 4.1 *Summary of Notation.*

4.3. Materialized View Creation

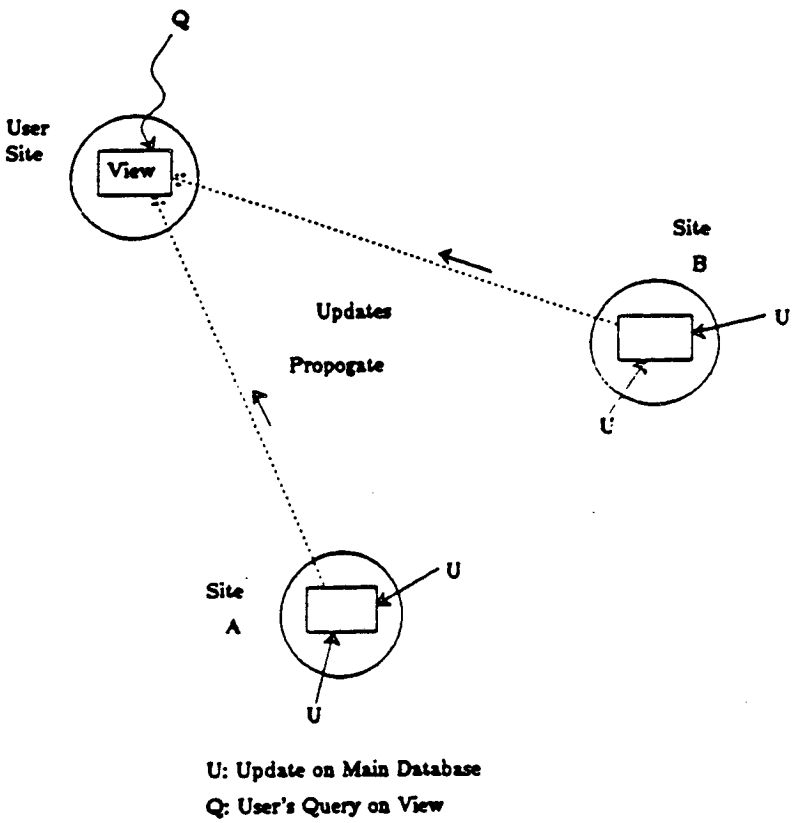


Fig. 4.2 *Propagation of Updates Between Copies.*

When a user defines a view, its materialized copy (MatView), is created using the basic database. It consists of records (if relational algebra operators are used in its definition), or aggregate information (if aggregation operators are used). It is then stored

at the site of the user creating it. As shown in *Fig. 4.2*, updates made to the main database have to be propagated to the view. If the view was required to be completely precise, i.e. reflect all changes in the main database, there would be no choice but to propagate each update on the main database immediately to the view. This is called the *immediate refresh policy*. However, for the applications we are looking at, some imprecision in data is tolerated. This enables us to design periodic refresh policies which propagate updates from the main database to the view periodically. *Table 4.2* compares periodic refresh policies with immediate ones.

Cost Factor	Immediate Policies	Periodic Policies
Transmission Cost of Data	High	Low
Contention on Data	High	Low

Table 4.2 *Comparison of Refresh Policies.*

4.4. Nature of View Updates

In this paper we have assumed that all the updates to the MatView are insertions. Since the precision of a materialized view has been defined as the fraction of tuples that it is out-of-sync with the main database, it suffices to consider each update as an arrival of a random variable, U_i, with a constant value 1. Formally the Update process can be specified as,

$$P(U_i = 1) = 1;\ i = 1, 2, 3, \cdots$$

If T_i is the time of arrival of U_i, then

$$(T_{i+1} - T_i) \approx Exponential(\lambda_U),\ i = 1, 2, 3, \cdots$$

Thus, U_i's are *Poisson* arrivals.

One way to model a database with both additions and deletions is to consider them separately, and then combine the effect by adding up the errors. However, addition of errors leads to bounds that are weak and do not adequately capture the quality of the data. A better model for such a database is to consider the updates U_i as both increasing or decreasing the counter (database size). We are currently working with the following model for U_i's.

$$P(U_i = 1) = r,\ P(U_i = -1) = 1-r;\ i = 1, 2, 3, \cdots$$

$$(T_{i+1} - T_i) \approx Exponential(\lambda_U),\ i = 1, 2, 3, \cdots$$

4.5. Periodic Time Refresh Policy

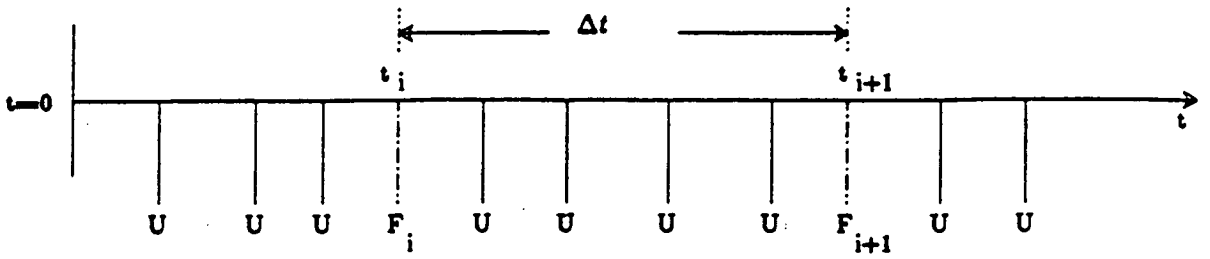


Fig. 4.3 *Updates Collecting Over Time.*

In this section we consider the design of a periodic time refresh policy which maintains the view MatView to a *desired degree of precision*. *Fig. 4.3* illustrates the arrival of various processes. From the definition of *degree of precision* we have,

$$P\left(\frac{N - N_0}{N_0} \leq (1-p)\right) \geq q$$

$$<=> P(N - N_0 \leq (1-p)N_0) \geq q$$

$$<=> \sum_{i=0}^{(1-p)N_0-1} \frac{e^{-\lambda_U \Delta t}(\lambda_U \Delta t)^i}{i!} \geq q$$

The expression on the left hand side can be approximated by the *Normal distribution* for large values of $\lambda_U \Delta t$ [FELL 68]. Formally,

$$\frac{(N - N_0) - \lambda_U \Delta t}{(\lambda_U \Delta t)^{1/2}} \approx Normal(0,1)$$

$$<=> P\left\{\frac{(N - N_0) - \lambda_U \Delta t}{(\lambda_U \Delta t)^{1/2}} \leq \frac{(1-p)N_0 - \lambda_U \Delta t}{(\lambda_U \Delta t)^{1/2}}\right\} \geq q$$

$$<=> \frac{(1-p)N_0 - \lambda_U \Delta t}{(\lambda_U \Delta t)^{1/2}} \geq F^{-1}(q)^{\dagger}$$

$$<=> (\Delta t)^2 - \frac{(F^{-1}(q))^2 + 2(1-p)N_0}{\lambda_U}(\Delta t) + \left\{\frac{(1-p)N_0}{\lambda_U}\right\}^2 \geq 0 \quad (4.1)$$

$$<=> \Delta t = \frac{1}{2\lambda_U}\left[\, 2(1-p)N_0 + (F^{-1}(q))^2 \pm [(F^{-1}(q))^4 + 4(1-p)N_0(F^{-1}(q))^2\,]^{1/2}\,\right]$$

As shown in *Fig. 4.4*, the quadratic form of *Eqn. 4.1* is non-negative in the regions $(-\infty, \Delta T_1]$ and $[\Delta t_2, \infty)$. The value of Δt (call it T) required is such that the quadratic form of *Eqn. 4.1* is non-negative for all $\Delta t \leq T$. This limits the choice to be,

† $F^{-1}(q)$ is the inverse function of the Normal distribution that gives the value of the *standard normal random variable*, to the left of which the probability is q.

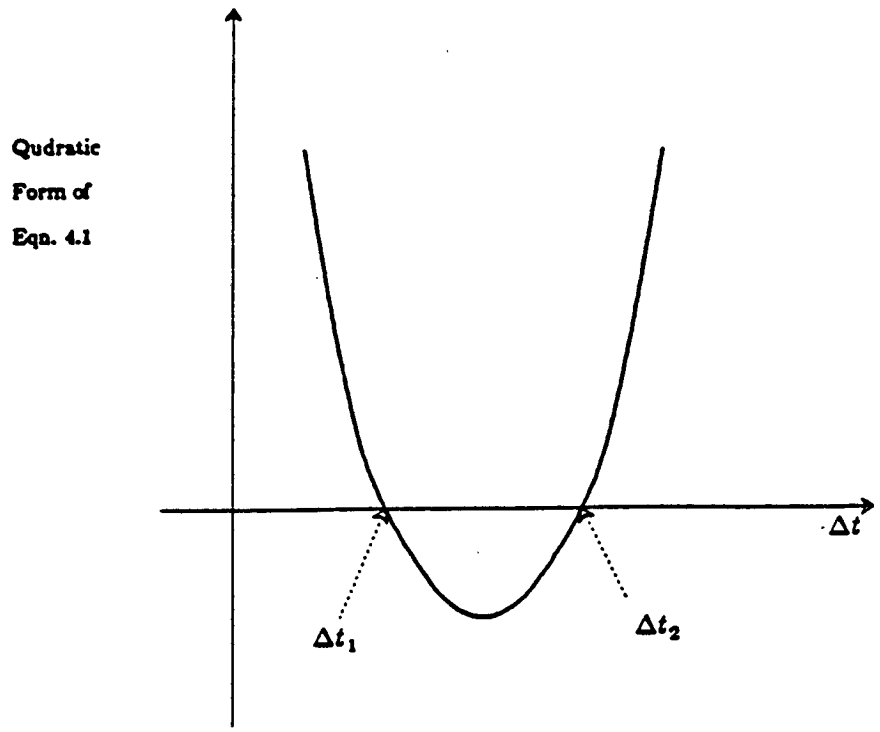


Fig. 4.4 *Roots of Eqn. 4.1*

$$\Delta t \in (-\infty, \Delta t_1]$$

The minimum cost refresh policy is to choose the largest possible value for Δt, i.e.

$$\Delta t = \Delta t_1$$

$$<=>$$

$$\Delta t = \frac{1}{2\lambda_U}\{\, 2(1-p)N_0 + (F^{-1}(q))^2 - [(F^{-1}(q))^4 + 4(1-p)N_0(F^{-1}(q))^2\,]^{1/2}\,]$$

Therefore, the optimal refresh algorithm is a periodic time algorithm. Every Δt_1 time unit it is executed to refresh MatView by installing all the Updates that have arrived since the last refresh.

Example: Time periods of the minimal policies were calculated for some values of N_0, p, and q. The results are presented in *Table 4.3(a)* and *Table 4.3(b)*.

As seen in *Table 4.3(a)*, Δt, i.e. the frequency of refreshes, decreases with increase in p, and also with increase in N_0. As observed in *Table 4.3(b)*, Δt decreases with increase in N_0, and with increase in q. For

$$N_0 = 1000;\ p = 0.90 \text{ and } q = 0.98,$$

$$\Delta t = 8.1453s$$

N_0 \ p	0.90	0.95
100	0.5279	0.2054
1000	8.1453	3.7287
10000	93.7093	45.6112

$q = 0.98$, $\lambda_U = 10/s$

Table 4.3(a) *Change in Δt with N_0 and p, fixed q.*

N_0 \ q	0.95	0.98
100	0.2434	0.2054
1000	3.9643	3.7287
10000	46.4545	45.6112

$p = 0.95$, $\lambda_U = 10/s$

Table 4.3(b) *Change in Δt with N_0 and q, fixed p.*

$\lambda_U = 10/s$ means that on the average, 81.453 updates are allowed between successive refreshes. Since the precision required is only 0.90 it would seem that $(1-p)N_0 = 100$ updates should be allowed between refreshes. However, the refresh rate is higher (Δt is smaller) because $\lambda_U \Delta t$ represents only the *average* number of updates in the interval Δt. In fact there is a non-negligible probability of having more arrivals. Since our aim is to provide 0.90 precision in the worst case (actually not quite since it is being done only with 0.98 confidence), the refresh rate is slightly higher.

4.6. A Stochastic Count Refresh Policy

The idea behind a stochastic count policy is to design a refresh process, *Refresh*, based on the number of the updates (count) between successive refreshes. The refresh process will be designed as a Poisson process. The important criterion here is that the probability that the count of updates between successive refreshes exceeds a certain threshold is extremely small. From the definition of *degree of precision* we have,

$$P(\frac{N - N_0}{N_0} \leq (1-p)) \geq q$$

Considering the mixture of Update and Refresh processes we have a process in which the probability of any arrival being Update or Refresh is fixed, and is given by

$$p_U = \frac{\lambda_U}{\lambda_U+\lambda_F};\ p_F = \frac{\lambda_F}{\lambda_U+\lambda_F}$$

The criterion mentioned above is equivalent to saying that the probability of the number of Update arrivals between successive Refreshes being more than $(1-p)N_0$ is smaller than

$1-q$. Thus,

$$P(\#\ Updates\ between\ successive\ Refreshes > (1-p)N_0) \leq 1-q$$

$$<=> P(\#\ Updates\ between\ successive\ Refreshes \leq (1-p)N_0) \geq q$$

$$<=> p_F + p_F p_U + p_F p_U^2 + \cdots + p_F p_U^{(1-p)N_0} \geq q$$

$$<=> 1 - p_U^{(1-p)N_0+1} \geq q$$

Now, $p_U = \dfrac{\lambda_U}{\lambda_U + \lambda_F}$

Thus,

$$\lambda_F \geq \lambda_U((1-q)^{\frac{-1}{(1-p)N_0+1}} - 1) \quad (4.2)$$

Thus, the optimal refresh process is a Poisson process that has the rate λ_F given in *Eqn. 4.2* above. Example: Update rates for the minimal refresh policies were calculated for some values of N_0, p, and q. The results are presented in *Table 4.4(a)* and *Table 4.4(b)*.

p / N_0	0.90	0.95
100	4.271	9.194
1000	0.395	0.797
10000	0.0392	0.0784

$q = 0.98$, $\lambda_U = 10/s$

Table 4.4(a) *Change in λ_F with N_0 and p, fixed q.*

q / N_0	0.95	0.98
100	6.475	9.194
1000	0.605	0.797
10000	0.060	0.0784

$p = 0.95$, $\lambda_U = 10/s$

Table 4.4(b) *Change in λ_F with N_0 and q, fixed p.*

As observed in *Table 4.4(a)*, the rate of the refresh process, λ_F, decreases on lowering the precision requirement, p, while moving along a row. Also, the rate decreases along a column as N_0 increases. As observed in *Table 4.4(b)*, λ_F *increases with* q. *For*

$$N_0 = 1000;\ p = 0.90\ and\ q = 0.98,$$

$$\frac{\lambda_U}{\lambda_F} = \frac{10}{0.395} = 25.317$$

This rate is higher than one would expect, reasons for which being similar to that for the Periodic Time Refresh Policy. With the solution derived above there is a non-zero probability of successive refreshes with no updates in between. A design which makes it

mandatory for some updates to occur between successive refreshes decreases λ_F, thus making the refresh policy cheaper. However, the analysis of such a policy is quite involved, and is not presented here.

5. Other Considerations in Precision-Time Tradeoff

Many challenging problems remain, and our future efforts will be directed towards them. Specifically the following problems are important:

Logical Model: Extensions to the query language have to be provided to *(i)* enable the user to define views, i.e. data copies, with desired degrees of precision, and *(ii)* express the precision requirement of a query.

Query Processing & Optimization: Query processing and optimization strategies have to take into account the existence of data copies with differing degrees of precision. Processing queries from a copy with less precision will be cheaper due to lesser amount of data contention on it. The query optimizer should generate an access path such that the data accessed has no more degree of precision than is necessary.

Physical Model: The techniques, i.e. data structures and algorithms, developed have to be implemented to see how well they perform on real data. The queueing model we have developed for view maintenance has to be extended to include replicated copies of views. New data storage techniques have to be devised, since conventional ones are oriented towards small and simple queries.

Data Distribution: The problems of data fragmentation, i.e. does a database need to be broken up into parts, and data placement, i.e. where should the units be placed, have to be revisited. Data of different degree of precision may be located at different nodes. A query arriving at a node is checked to determine its precision requirements, and sent to the appropriate node to be processed.

Data Replication: How many data copies of each degree of precision are required? It will depend on the requirements of the environment.

Fault Tolerance: If multiple copies of data exist in a distributed conventional database, the failure of sites holding more than half the copies can cause the whole system to stop. The reason for this is the need for complete precision of all copies. This approach is called the pessimistic or the careful approach. Since we allow the precision of some of the data copies to be less than complete, such a system can still continue operation. A

challenging problem is to model the tradeoff between the availability of the system (i.e. how long after the failures the system is still operational), to the degree of imprecision accumulated in the inactive data copies.

Extension to Knowledgebases: Our emphasis so far has been on views defined in terms of relational and aggregate operations. Recent years have seen the rapid development of *fuzzy logic* [ZADE 65] and *possibility theory* [ZADE 78], and their applications to *approximate reasoning*. In this logical framework the *truth* or *falsehood* of a proposition is not a binary value. Instead it is a real number, α, between 0 and 1, where α and $1-\alpha$ are the *degree of truth* and *degree of falsehood* of the proposition, respectively. The value *α depends on the evidence collected so far, both for and against the proposition. Each new piece of information increases or decreases α depending on whether it provides evidence for or against the proposition.*

Our model of abstraction maintenance is very well suited to support such a framework of logic. Consider a set of propositions, representing fuzzy facts about some domain, which are defined in terms of some premises, i.e. existing pieces of evidence. The former correspond to the abstractions in our model while the latter correspond to the base data. As more knowledge about the domain is gathered, additional evidence about the facts is available and the corresponding α values have to be revised. For a specified degree of precision in the α's what is the most efficient way of maintaining them, is precisely the problem that our model provides an answer to.

6. Conclusions

Conventional databases are often not suitable for statistical applications, since they do not fulfill many of the important needs of the latter. Specifically, during calculating statistical parameters, they provide results that are completely precise, and spend prohibitive amount of resources in doing do. Almost all analysis in statistics can be done in the presence of imprecise data, as long as the error is bounded. Bearing this in mind we present the idea of *precision-time tradeoff* as a new paradigm for processing queries that evaluate statistical parameters from a database. The notions of *precision of data* and *precision-requirement of query* were defined. The main idea is that the cost of processing a query should be proportional to its precision-requirement, i.e. highest if full precision is required. We described a data model which allows data with varying degrees of precision. We next described *materialized views* as a mechanism (i.e. the physical model) to support

data with a specified degree of precision. We used a stochastic model to analyze the problem of view maintenance, and derived two efficient maintenance policies. Finally, we outlined the various issues that the new paradigm raises, and that need to be addressed.

7. References

[ASTR 77] Astrahan, M.M., "System R: A Relational Database Management System", IBM Research Report.

[BLAK 86] Blakeley, J.A., P.Larson and F.W.Tompa, "Efficiently Updating Materialized Views", Proc. of the 1986 ACM-SIGMOD Conf. on Management of Data, Washington DC, May 1986, 61-71.

[COCH 53] Cochran, W.G., "Sampling Techniques", John Wiley Sons, New York, USA, 1953.

[FELL 68] Feller, William, "An Introduction to Probability Theory and Its Applications", John Wiley & Sons, Inc., New York 1968.

[GHOS 85] Ghosh, S.P., "SIAM: Statistics Information Access Method," IBM RJ 4865 (51295).

[HANS 87] Hanson, Eric N. "A Performance Analysis of View Materialization Strategies," Proc. of the 1987 ACM-SIGMOD Intl. Conf. on the Management of Data, San Francisco, CA, May 1987.

[HOEL 71] Hoel, P.G., S.C.Port and C.J.Stone, "Introduction to Probability Theory," Houghton Mifflin Company, Boston, 1971.

[HOU 87] Hou, Wen-Chi, G. Ozsoyoglu, B.K. Taneja, "Statistical Estimators for Relational Algebra Expressions," Deptt. of Comp. Sc., Case Western Reserve University, 1987.

[OLKE 86] Olken, F. and D.Rotem, "Simple Random Sampling from Relational Databases," Proc. of the Conf. on VLDB, Kyoto, Japan, August, 1986.

[ROSS 85] Ross, Sheldon M., "Introduction to Probability Models", Academic Press, Inc., Orlando, Florida, 1985.

[ROUS 86] Roussopoulos, N. and H.Kang, "Principles and Techniques in the Design of ADMS+/-", Computer, December 1986.

[SHOS 82] Shoshani, A., "Statistical Databases: Characteristics, Problems, and Some Solutions." Proc. 8th Intl. Conf. on VLDB, 1982, pp 208-222.

[SRIV 87] Srivastava, J. and Doron Rotem, "Analytical Modeling of Materialized View Maintenance," Lawrence Berkeley Laboratories Tech. Rep., 1987.

[ULLM 82] Ullman, J.D., "Principles of Database Systems," Computer Science Press, 1982.

[VITT 84] Vitter, Jefferey S., "Faster methods of Random Sampling," CACM 27(7):703-718, July 1984.

[ZADE 65] Zadeh, L.A., "Fuzzy Sets", Information and Control 8, 1965, pp. 338-353.

[ZADE 78] Zadeh, L.A., "Fuzzy Sets as a basis for a theory of possibility." Fuzzy Sets and Systems, 1, pp. 3-28, 1978.

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720