Journal of International Information Management

Volume 5 | Issue 1 Article 6

1996

A differential refresh scheme for remote end-user's views

David Chao
San Francisco State University

George Diehr
California State University, San Marcos

Aditya N. Saharia
University of Illinois Chicago

A differential refresh scheme for remote end-user's views

David Chao
San Francisco State University

George Diehr
California State University, San Marcos

Aditya N. Saharia
University of Illinois at Chicago

ABSTRACT

The growth of end-user computing and recent developments in information technology, such as client/server architecture and data warehouse, promote the use of remote materialized views (RMVs) to support end-users. This article presents a differential scheme to refresh remote end-user's views. The scheme stores the effects of updates relevant to the RMV in a "difference table" which is transmitted to the remote site upon receiving the refresh request to update the RMV. The scheme provides a fast response to a user's refresh request. We discuss the data structures and algorithms of the scheme. Performance measures are developed and compared with the regeneration scheme.

INTRODUCTION

The currency requirements for the information needs of different members of an organization vary considerably. For example, the activities at the operations management level usually require detailed internal data reflecting the current operational state of the organization. On the other hand, at the strategic management level where various planning activities take place, staleness in data may be tolerated. Sprague and Carlson (1982) claimed that, "Few decision support systems have a requirement for real time data" and decision support system users may prefer to work with a static version of the data. Adiba and Lindsay (1980) suggested that users who do not have to or do not wish to work with the operational (and hence constantly changing) database be supported via "snapshots." A snapshot reflects a user's view of the operational data at a fixed point in time. Thus, snapshots correspond to "materialized views" except that they are "refreshed" (i.e., brought to a state consistent with the operational database) only when the user explicitly issues a refresh request.

With the common availability of personal computers and work stations, end-user computing has grown tremendously. Snapshots, usually in the form of downloaded data, have been commonly used to satisfy the end-user's needs for data and as a way to maintain the security of the operational database. Current developments in information system technology, such as client/server network architecture, also promote the use of snapshots as a cost effective alternative to achieve distributed computing. Typically, a client/server system maintains a centralized operational database at the server site, and client site applications are supported using snapshots. Recently, widespread interest has developed in the use of so-called "data warehouses" to store information which is extracted from operational and external databases and used primarily in decision support applications (Radding, 1995; White, 1995). With these developments, snapshots have taken the form of "remote materialized views" (RMVs): a user who is interested in working with a snapshot requests a copy of the appropriate subset of the operational database at the client site. The RMV is then used without further interaction with the operational database until the user perceives that the RMV has become stale due to updates to the operational database. Note that in a distributed environment, even in situations where it is desirable to make the current data available to the users, the overhead associated with concurrency control protocols may make the use of current data prohibitively expensive. Another factor that discourages a tightly coupled distributed database is the lack of homogeneity of software across the network and various types of DBMSs such as Access and dBASE. In such cases, the use of RMVs with frequent refreshes provides an attractive alternative.

There are a number of ways to refresh RMVs. An obvious and easy solution is to regenerate the materialized view each time it is referenced. However, the cost of the refresh can be substantially reduced by following a differential refresh strategy which is even more attractive because of reduced communication costs. The current view is obtained by posting the additions and deletions to the view as a result of updates to the base table:

New view = Old view - Deletions + Additions

For example, Hanson (1987) suggested that updates to a materialized view be deferred until it is actually referenced by a user. To facilitate identification of changes in the base tables since the last refresh, Hanson suggests the use of a hypothetical relation for each permanent relation, which is made up of the base table itself, a table which keeps additions, and a table which keeps deletions. Blakeley et al. proposed a differential view maintenance scheme in which each update tuple is pre-screened to determine whether the update is relevant for a view or not before applying the update to the view (Blakely, Larson & Tompa, 1986; Blakeley, Coburn & Larson, 1989). Lindsay et al. proposed timestamping database records to identify changes since the last refresh (Lindsay, Hass, Moham, Pirahesh & Wilms, 1986). The refresh message for an RMV is generated by sequentially scanning the base table and identifying the tuples relevant to the snapshot which have been inserted or deleted since the last snapshot. Also to allow for efficient identification of the deleted records, they use pointers embedded within base table records which identify the boundaries of the empty regions. Since the only purpose of the timestamp and pointer system is to facilitate RMV refresh, the timestamp and the pointer are fixed at the time the refresh message is generated. The scheme is easily applied to multiple RMVs but is not easily extended to the RMVs based on multiple tables (i.e., defined by SPJ views).

Segev and Park (1989) and Roussopoulos and Kang (1986) provided schemes in which the history of updates to the base tables is kept in log files. To determine the entries in a log file which have come up since the last refresh, Segev and Park suggested timestamping the log file entries whereas Roussopoulos and Kang suggested using a pointer maintained by the remote site. To limit the growth of the log file, entries which have been used to refresh all the RMVs are deleted from the top of the log file. Both the schemes can be applied to the case of multiple RMVs. The scheme reported in Roussopoulos and Kang (1986) has been extended to RMVs based on multiple base tables. Segev and Park do not provide extension to the RMVs based on multiple tables. In case the communication lines between the central site and the remote sites are available throughout, the RMV refresh can be initiated by the central site. The central site forced refresh scheme has been recently suggested by Alonso et al. where the user at a remote site specifies the maximum divergence allowed between an object at the central site and its copy (called "quasi-copy") at the remote site (Alonso, Barbara & Garcia-Molina, 1990). The refresh is postponed to the last moment possible allowed by the divergent conditions specified by the user.

The differential refresh methods reviewed above require the processing of the base tables or the update logs to generate the differential refresh messages after receiving the refresh request from the user, thus prolonging the response time to the user. In this work we describe the difference table scheme for maintaining RMVs. In this scheme, the effect of all the updates to the base tables on an RMV is maintained in a difference table. The difference table keeps the tuples to be deleted from and to be added to the RMV, i.e., the difference table contains the net refresh messages to the view. On receiving the refresh request, the contents of the difference table are sent to the remote site and the difference table is initialized to empty. This scheme provides a fast response to the user's refresh request. It supports RMVs based on the joins of multiple tables as well as multiple RMVs. The added costs of supporting the difference table scheme are the cost of performing relevance checks on the updates to determine their relevance to the view and the cost of maintaining the difference tables. At low updates rates and low qualification rates (proportion of base table records that satisfy the view definition) the cost of performing the relevance checks and maintaining the difference table will be relatively small compared to the cost of scanning the base tables for the regeneration scheme. Section 2 gives an overview of the scheme, and descriptions of additional procedures needed in the scheme. Section 3 compares the performance of the scheme with the regeneration scheme, and section 4 gives the conclusions.

THE DIFFERENCE TABLE SCHEME

In this scheme, each RMV is supported by a difference table (called DT in the sequel) which maintains the net changes to be posted against the RMV since the previous refresh. The query defining the RMV is stored at the central site. When an update arrives it is checked against the RMV definition to determine if it is relevant for the RMV. If so, the effect of the update on the RMV is posted in the DT. When a refresh request arrives from the remote site, the current entries in the DT are transmitted to the remote site and the DT is initialized to status empty. The remote site then runs an updating algorithm to incorporate the changes in its RMV.

Multiple updates to the same tuple in a base table result in at most two entries in the DT. For example, consider an existing base table tuple which receives multiple updates. If both the original (before update) tuple and the final updated tuple are relevant, the DT will contain the effect of deleting the original version of the tuple and the effect of inserting the latest version. If the original version was not relevant, but the latest is relevant, the DT will contain only the effect of the latest version. If the original version was relevant, but the latest was not relevant, the DT will contain only the effect of the original version. The fourth possibility, neither original nor final versions are relevant, results in no entries in the DT.

Algorithms and Data Structures

The additional data structures required at the central site are the RMV defining query and the difference table. Algorithms at the central site are needed to: 1a) decide whether an update to a base table is relevant for the RMV and (if so) 1b) make proper updates to the DT. Additional algorithms are executed upon receipt of a refresh command from the remote site to: 2a) transmit the contents of the difference table to the remote site and 2b) initialize the difference table to empty status. A remote site algorithm is needed to 3) update the RMV table using the difference table received from the central site.

Descriptions of these algorithms are facilitated by the introduction of the following standard relational algebra terminology: The RMV is defined on the scheme $R = \{R_1 \cup R_2 \cup . . . \cup R_p\}$ by the query Q: $\pi_x \sigma_{C(Y)}(r)$, where $r = r_1 \bowtie r_2 \bowtie . . . \bowtie r_p$; r_k is a base table on the scheme R_k for k=1,2, ... p; $\bowtie$ represents the natural join; X and Y are each sets of attributes included in R; C(Y) is a Boolean expression specifying the selection condition. The RMV table r_v is defined by the scheme V={X} if X includes a key for the relation r, otherwise by the scheme {X, COUNT}. The field COUNT in r_v takes on positive integer values to indicate the number of occurrences of each tuple. This allows us to eliminate duplicates of a tuple from the RMV which may come up because of projection operation on r while still allowing us to retain the tuple unless all the matching tuples from r have been deleted. The difference table r_s, is defined by the scheme S={X, COUNT} where a positive value of COUNT indicates the number of additions and a negative value indicates the number of deletions. Thus, for an RMV which is defined by a select operation on r (i.e., X = R), a tuple is inserted into the difference table with COUNT = +1 for an add, COUNT = -1 for a delete. A value modifying update is treated as a pair of entries--a deletion and an addition. A tuple in a table r_k for k=1,2,. . .,p, v, s, is denoted by t_k.

The impact of an update to the table r on the RMV depends on the type of update (add, delete, or value modifying update). This is now formalized:

If X includes a key for the relation r, then

(i) an addition to r, denoted t^a, will cause an addition $t_v{}^a = \pi_X(t^a)$, if t^a is relevant for r_v;

(ii) a deletion t^d will cause a deletion $t_v{}^d = \pi_X(t^d)$ if t^d is relevant for r_v;

(iii) a value modifying update, represented by the pair of tuples $t^d t^a$, will cause a deletion $t_v{}^d = \pi_X(t^d)$, if t^d is relevant for r_v and an addition $t_v{}^a = \pi_X(t^a)$, if t^a is relevant for r_v;

(iv) a sequence of value modifying updates for one entry in r will create the pairs $t^d(1), t^a(1), t^d(2), t^a(2) \ldots t^d(m), t^a(m)$, where $t^d(2)=t^a(1)$, $t^d(3) = t^a(2)$ etc. The impact on the RMV table depends only on the relevance/non-relevance of the original tuple value, t(1) and final updated value, $t^a(m)$. If t(1) is relevant, a deletion $t_v^d = \pi_X(t^d(1))$ results. If $T^a(m)$ is relevant, an addition $t_v^a = \pi_X(t^a(m))$ results.

If X does not include a key for the relation r, then
(v) t_v^a is expanded with COUNT = 1, and t_v^d is expanded with COUNT = -1.

Central Site Algorithms for Relevance Check and Difference Table Maintenance

The relevance check is straightforward when the RMV scheme is limited to selections and projections (i.e., when p = 1). In this case all the attributes of a tuple t being inserted into or being deleted from the base table are known and hence determining if C(Y) is True or False can be done in time linear in number of conditions.

For each t generated by the update:
1. Perform a relevance check of t, i.e., determine if $t_t = \pi_x \sigma_{C(Y)}(t)$ is non-null.
2. If t_t is non-null update the difference table as follows:
   a. If no tuple in r_s matches t_t (i.e., is identical ignoring the COUNT field), add the tuple $t_s = t_t$ to r_s with t_s. COUNT = +1 if t is an addition, t_s. COUNT = -1 if t is a deletion.
   b. If a tuple in r_s matches t_t, add or subtract one to the COUNT field of the matching tuple in r_s depending on whether t is an add or a delete. (Note that there can be at most one such matching tuple in r_s.) If the resulting COUNT field value is zero, delete the tuple.

RMVs defined by joins will, in general, complicate relevance checking. This is because deciding that an update is relevant will always require performing the joins. However, a non-relevance determination may be possible without performing the joins. For example, a simple non-relevance check is possible if the RMV defining query has a special form. First, define Y = {Y1, Y2} where $Y1 = Y \cap R_k$ where r_k is the base table receiving the update t_k, and $Y_2 = Y - Y_1$. Then if the following conditions hold the update is not relevant:

1. C(Y) is a conjunction of terms,
2. one or more of the terms involves only attributes contained in Y_1,
3. the defining query does not involve a self join of r_k,
4. replacing attributes Y_1 by $t_1 = t_k.Y_1$ makes at least one term false.

This preliminary test for re.evance (more properly, test for non-relevance) is clearly very inexpensive. It is essentially the same type of test performed for the relevance check for simple projection/selection RMVs. No file access operations are required.

In a more general case, the preliminary relevance check amounts to determining if a substitution for uninstantiated variables exist which will make the restricting condition true, a problem well known to be NP-complete. Rosenkrantz and Hunt (1980) have shown that for the class of expressions which are conjunctions of atomic formulas which do not include a "not equal to" operator, satisfiability can be decided in polynomial time in the number of uninstantiated variables. The algorithm proceeds via constructing a directed graph where the nodes are variables $y \in Y_2$ (i.e., the uninstantiated variables of Y) and a special node called null node corresponding to constant zero. Each atomic formula of the type $x \leq y + c$, where x and y are either variables or the constant zero and c is a constant, is replaced by a directed arc of weight c from node x to node y. The restricting condition is false as a result of substitution if the corresponding graph contains a cycle of negative length which can be verified in polynomial time using Floyd's algorithm. Note that the substitution approach of the R/H algorithm allows us to determine if an update is not relevant or if it is potentially relevant. If the relevance check determines potential relevance, the joins must be performed. At this point, the final relevance check and the difference table update are performed. These steps are identical to those described above for select/project queries. Clearly the substitution approach is practical only when the restricting condition has a few clauses or falls into the special class such that it can be transformed into a conjunction of atomic terms which do not involve the "not equal to" operator.

Central and Remote Site Algorithms for RMV Update

Upon receipt of a message from the remote site for refresh, the central site must transmit the contents of the difference table to the remote site and initialize the difference table to empty status. Upon receipt of the difference table, the remote site algorithm updates the RMV table. The central site algorithms are straightforward. An outline of the remote site RMV maintenance algorithm follows.

For each tuple, t_s, in the difference table, the following steps occur:

1. If t_s matches a tuple t_v in r_v (i.e., is identical ignoring the COUNT field), then: Add t_s.COUNT to t_v.COUNT. If t_v.COUNT = 0, delete this tuple from r_v.
2. If t_s does not match a tuple t_v in r_v, insert t_s into r_v. (Note, t_s. COUNT must be positive in this case.)

PERFORMANCE CONSIDERATIONS

We assume that the objective is to minimize the overall system costs due to refresh activities. The two major components which contribute to overall cost are:

(i) Communication costs and
(ii) Delay costs imposed on the operational activities.

The delay costs on operational activities are, in turn, determined by the amount of CPU time spent and the number of I/O operations at the central site due to the refresh activity. Here we give the cost estimates for RMVs based on a single table. The analysis can be easily generalized to the case of RMVs based on multiple tables. To obtain an estimate of the overall systems costs, we assume that:

(i) An RMV is defined by applying SELECT operation to a single base table.
(ii) The updates to the base table are distributed randomly over the entire base table, the probability of a tuple receiving the next update is independent of how many updates it has received in the past.
(iii) The size of the base table is not changing (or at the most changing slowly).
(iv) The probability that a base table tuple is qualified to be included in the RMV remains constant irrespective of when the tuple was updated. This assumption, together with the previous assumption of (essentially) constant base table size, results in an essentially constant RMV table size.

The following parameters are used in determining systems costs:

N:	Number of tuples in the base table
Λ:	Frequency of refresh requests
λ:	Frequency of updates to the base table
q:	Fraction of tuples in r which satisfy the snapshot condition

Communication Costs

In one refresh cycle, the number of updates posted against the base table is λ/Λ. Making exponential approximation to the binomial distribution, the expected number of tuples receiving at least one update is given by:

$$Nf = N(1 - e^{-\lambda/\Lambda N}),$$

where f is the fraction of the base table modified. In case the updates to a single record are identified by sending a delete (if the tuple was qualified before the updates) followed by an insertion (if the tuple is qualified after the updates), the average number of tuples sent to the remote site per updated tuple in the base table is given by 2q. The communication costs per refresh is given by

$$2qN(1-e^{-\lambda/\Lambda N}). \qquad (1)$$

The number of messages in the limit of $\lambda/\Lambda N$ is $2q\lambda/\Lambda N$and asymptotically approaches 2qN for large $\lambda/\Lambda N$. By comparison, the number of messages transmitted in a regeneration scheme is always qN, independent of the frequency of update or the frequency of refresh. The comparison of the two schemes is given in Figure 1. The difference table scheme does better than the regeneration scheme for $\lambda/\Lambda < \log_e 2$ si· ce in this case the expected size of the difference table at each refresh is less than qN. In developing expression (1), we have assumed that both of the before and after versions of a tuple being updated are relevant to the RMV and, therefore, each generates one message. In case the key fields of the base table are included in the attributes of the RMV, we can preprocess the difference table to combine the two messages into a single message and send it with COUNT set to 0 to indicate that the tuple in the RMV is to be replaced by the new tuple. In this case the number of messages is given by:

$$(2q\text{-}q2)N(1\text{-}e^{-\lambda/\Lambda N})). \qquad (1a)$$

In this case the difference table scheme will provide savings over the regeneration scheme over a larger range. Also if keys are available to identify tuples, we can further reduce the communication costs by just sending the keys of the tuples which are to be deleted from the RMV.

Delay Costs

The refresh activity imposes delay costs on the operational activity becuase of the following two processes:

(a) Relevance check. Each update must be checked for relevance against the RMV definition. Assume that the cost per tuple for a relevance check is given by $C_{R.}$. Since for each update to the base table, both the versions before the update and the version after the update have to be checked for relevance, the cost of relevance checking per refresh is given by $2C_R\lambda/\Lambda$.

(b) Difference table maintenance. Update of the DT involves a search for a matching tuple followed by addition, deletion, or modification. The search will involve both CPU time and I/O processes. Assume that the cost of searching and updating the DT is constant, C_M (e.g., the DT is a hashed file). Thus, the cost of difference table maintenance is given by $2qC_M\lambda/\Lambda$. The overall delay costs is thus given by

$$2(C_R + q\,C_M)\lambda/\Lambda \qquad (2)$$

To compare these costs to the corresponding costs for the full regeneration scheme, note that for the latter scheme when a refresh request is issued, the base table is scanned and each tuple in the base table is checked for relevance. Thus, the costs given by (2) have to be compared against the cost of scans, which will involve the I/O costs, comparison costs, and the delay costs imposed on other users. We denote such scanning costs by C_SN. In Figure 2 we give a comparison of the relevance check and DT maintenance costs with the scanning costs. Again we note that for low values of λ/Λ, our scheme is expected to work much better than a regeneration scheme or any other schemes which will require scanning the base table to determine qualified tuples.

(Figures 1 and 2 are on next page.)

Figure 1. Fraction of base table transmitted as function of number of base table updates per refresh as a fraction of the base table size.

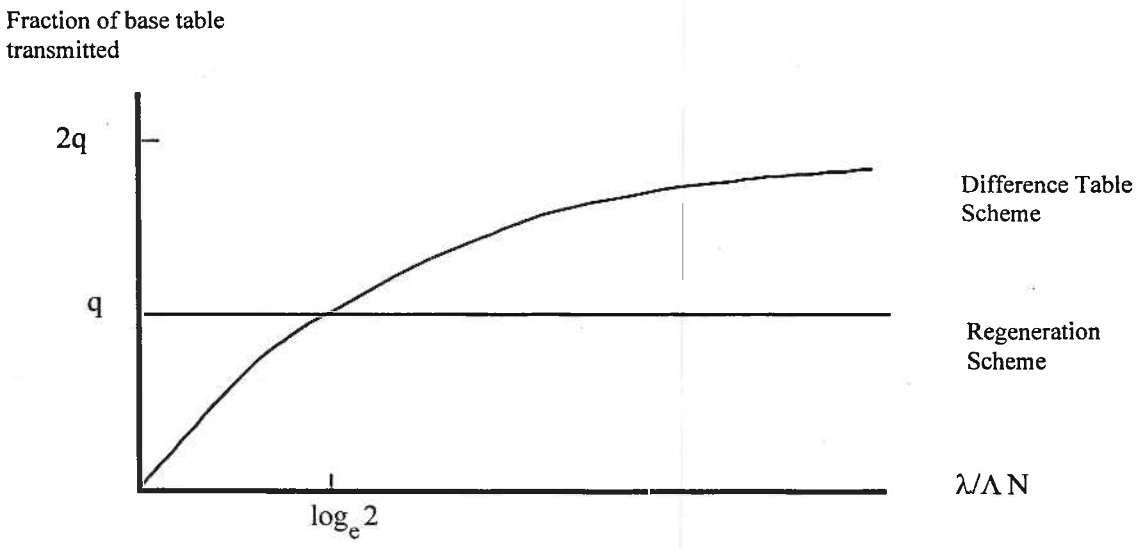
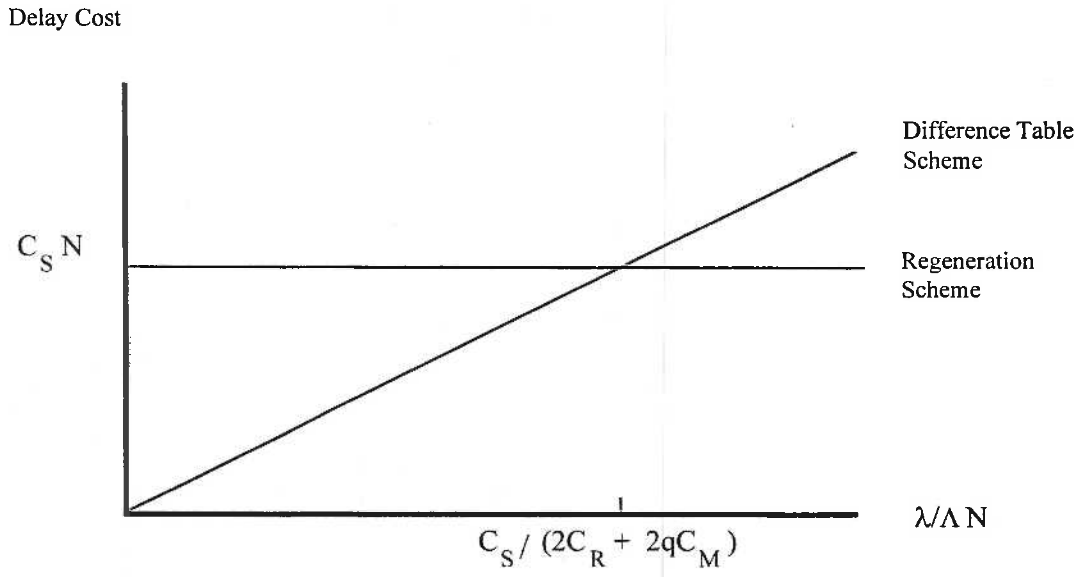


Figure 2. Delay costs per refresh as a function of number of base table updates.



DISCUSSION AND CONCLUSIONS

The difference table scheme compares very well with the regeneration scheme in terms of the amount of data communication. Implementation of the difference table scheme, however, requires large overhead in terms of CPU time and delay imposed on operational activities. In the regeneration scheme this delay cost is paid in terms of base tables being unavailable during the scans, whereas in our scheme the delay cost is paid at each update as the update has to be checked for relevance against the query definition of the RMV. From Figures 1 and 2 we observe that the difference table scheme should be implemented if

$$\lambda/\Lambda N<\log_e 2$$

and $$\lambda/\Lambda N<C_S/(2C_R+ 2qC_M).$$

The parameters $C_S, C_R, C_M, \lambda, \Lambda$ and q will be determined by the central database administrator when the request to support the RMV is specified. If at that time it is determined that the above two conditions are met, the differential refresh scheme should be implemented. If neither condition is met, clearly regeneration (or some other scheme) should be used. If only one of the conditions is met, the tradeoff between communication and delay costs must be considered. Also in an environment where a quick response to a user's refresh request is critical, DT should be considered because other schemes pay higher delay cost after receiving the refresh request. There are clearly other potential approaches to RMV maintenance. In particular, there are alternative ways to handle a transaction which applies to multiple tuples (e.g., increase the salary of all employees by 10%). To the extent that such an update causes no additions or deletions from the RMV, it is possible to transmit the update request itself rather than the modified table entries. Such an extension may be implemented by closing the current segment of the difference table, attaching the range modification entry, and starting a new segment.

REFERENCES

Adiba, M. & Lindsay, B. (1980). Database snapshots. *Proceedings of the 6th International Conference on Very Large Data Bases*, 86-91.

Alonso, R., Barbara, D. & Garcia-Molina, H. (1990, Sept.). Data caching issues in information retrieval systems. *ACM Transactions on Database Systems, 15*(3), 359-384.

Blakeley, J. A., Larson, P. & Tompa, F. W. (1986). Efficiently updating materialized views. *Proceedings of the 1986 ACM-SIGMOD Conference*, 61-71.

Blakeley, J. A., Coburn, N. & Larson, P. (1989, Sept.). Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems, 14*(3), 369-400.

Hanson, E. N. (1987). A performance analysis of view materialization strategies. *Proceedings of the 1987 ACM-SIGMOD Conference*, 440-453.

Lindsay, B. Hass, L., Mohan, C., Pirahesh, H. & Wilms, P. (1986). A snapshot differential refresh algorithm. *Proceedings of the 1986 ACM-SIGMOD Conference,* 53-60.

Radding, A. (1955). Support decision makers with a data warehouse. *Datamation, 41*(5), 53-58.

Rosenkrantz, D. J. & Hunt III, H. B. (1980). Processing conjunctive predicates and queries. *Proceedings of the 6th International Conference on Very Large Databases,* 64-72.

Roussopoulos, N. & Kang, H. (1986, Dec.). Principles and techniques in the design of ADMS+/-. *Computer,* 19-25.

Saharia, A. & Diehr, G. (1990, Sept.). A refresh scheme for remote snapshots. *Information Systems Research, 1*(3), 277-307.

Segev, A. & Park, J. (1989, June). Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering, 1*(2), 173-184.

White, C. (1995). The key to a data warehouse. *Database Programming & Design, 8*(1), 23-25.