LBL-26992

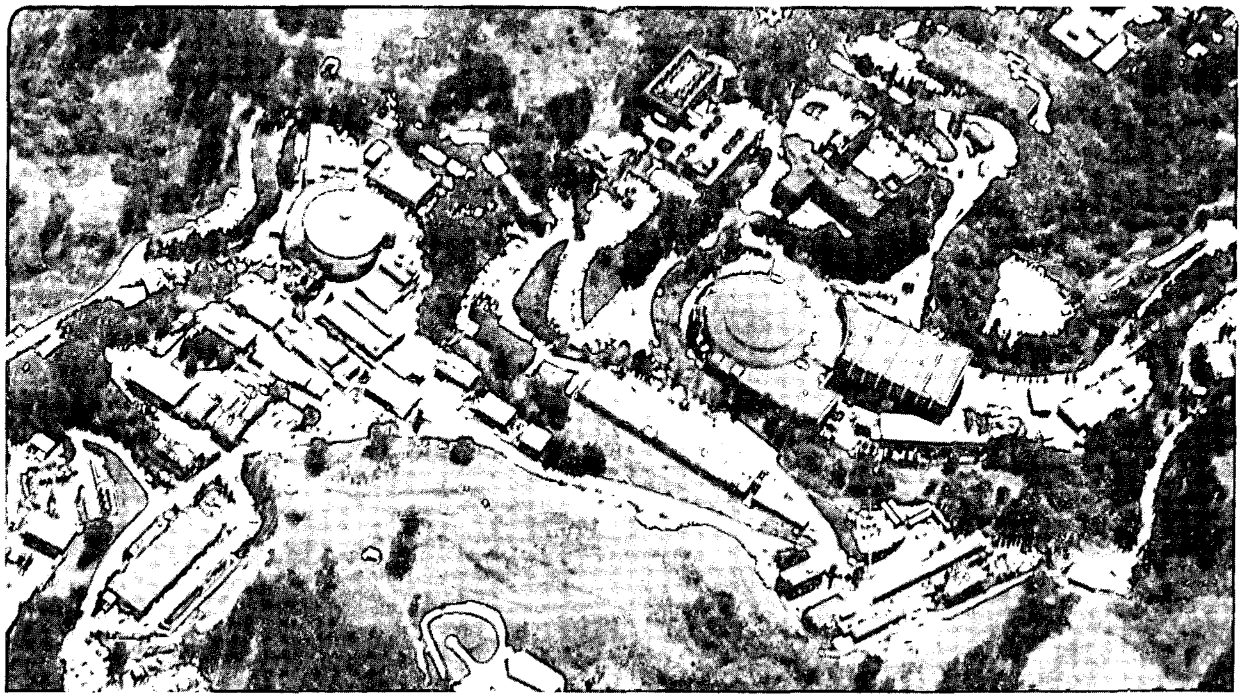
Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA, BERKELEY

Information and Computing Sciences Division

Managing Materialized Views in Distributed Database Systems

A. Segev

July 1989



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

LBL-26992

MANAGING MATERIALIZED VIEWS IN DISTRIBUTED DATABASE SYSTEMS

Arie Segev

School of Business Administration
University of California, Berkeley

and

Computing Science Research & Development
Information & Computing Sciences Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, California 94720

July 1989

This work was supported by the Director, Office of Energy Research, Applied Mathematical Sciences Research Program, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

LBL-26992

MANAGING MATERIALIZED VIEWS IN DISTRIBUTED DATABASE SYSTEMS

Arie Segev

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

Abstract

Materialized database views are a form of derived data. In a distributed environments they constitute a compromise between single copies of data and consistent multiple copies. In this paper we first motivate the support of materialized views by a database management system. We discuss why this capabality is important to the organization, to end-users and to database administrators. Next, we show that the concept can be generalized to inter-organization exchange of data, and address some managerial and technical issues in that context. Finally, we outline an analysis of policies for maintaining distributed materialized views. determinging optimal policies for updating distributed materialized views. We define the concept of materialized view currency and allow a query to specify its currency requirement. We also allow a materialized view to be updated from either a base relation or another materialized view. This flexibility provides an opportunity for further reduction in the cost of maintaining distributed materialized view. We model the problem of optimal update policies to capture currency and policy constraints, replicated data, and various view update policies. The optimization incorporates a minimum-cost objective function as well as user's response time constraints.

MANAGING MATERIALIZED VIEWS IN DISTRIBUTED DATABASE SYSTEMS

ABSTRACT

Materialized database views are a form of derived data. In a distributed environments they constitute a compromise between single copies of data and consistent multiple copies. In this paper we first motivate the support of materialized views by a database management system. We discuss why this capability is important to the organization, to end-users and to database administrators. Next, we show that the concept can be generalized to inter-organization exchange of data, and address some managerial and technical issues in that context. Finally, we outline an analysis of policies for maintaining distributed materialized views.

1. INTRODUCTION

Distributed computing is appropriate for many organization [KING83]. The software to support it is a distributed database management system [CERI84]. Replicating data is attractive in many instances, but guaranteeing consistency is a very expensive proposition in the presence of updates. The decision does not have to be "all" or "nothing". Materialized database views are a form of derived data which is stored explicitly. The strategy of maintaining materialized views in a distributed database system is a compromise between fully synchronized (consistent) replicated data and single copies of files. It is an attractive choice in many environments because it significantly decreases (relative to the case of synchronized replicated data) the cost of processing transactions while increasing (relative to the case of a single copy) the availability of the data and the performance of ad-hoc queries in remote sites.

Materialized views require that an update strategy be devised [BUNE79, ADIB80, BLAK86, LIND86, ROUS86, BLAK88, SEGE89a]. An obvious solution is to rematerialize the view after each update to the base data used to define the view, but normally a differential update procedure (e.g., [LIND86]) is superior. Since it is possible that modified base data is irrelevant to the view, screening 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 [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 versus the cost of updating it. In [SEGE89b], an analytical analysis of view update policies is presented.

The paper is organized as follows. Section 2 introduces the concept of database views in a relational DBMS. In Section 3, we extend the concept to a distributed environment. To illustrate the generality of materialized views, Section 4 presents their application to the case of inter-organization data acquisition. Update policies for distributed materialized views are discussed in Section 5, and the paper is concluded in Section 6.

2. DATABASE VIEWS

A relational database management system (DBMS) stores data in base relations (or tables) such as the one shown in Figure 1.

EMPLOYEE	EMP#	NAME	SALARY	DEPT#
	E1	JIM	35k	D1
	E2	MAKE	30k	D1
	E3	DAVE	35k	D2
	E4	ELLEN	40k	D2
	E5	RON	30k	D1
	E6	RUTH	25k	D2

Fig. 1: A Relational Representation of an Employee Data

The figure presents the rows (or tuples) of the relation EMPLOYEE. We will denote the schema (or attributes) of a relation by a relation name followed by the list of attributes in parentheses, e.g., EMPLOYEE(EMP#, NAME, SALARY, DEPT#). A view is a virtual relation defined by expressing a query on base relation(s). For example, if a user is interested only in employees who earn more than

30k, the following SQL query can be used to defined a view representing the data of these employees (* indicates that all attributes are to be selected):

```
DEFINE VIEW HIGH_EMP (EMP#, NAME, SALARY, DEPT#)
        SELECT *
        FROM EMPLOYEE
        WHERE SALARY > 30000
```

Figure 2 illustrates the data associated with this view.

HIGH_EMP	EMP#	NAME	SALARY	DEPT#
	E1	JIM	35k	D1
	E3	DAVE	35k	D2
	E4	ELLEN	40k	D2

Fig. 2: The HIGH_EMP Data

The HIGH_EMP table does not exist, and is represented in the DBMS by its definition query. From the user's point of view this table exists and can be manipulated by queries. For example, the following query retrieves the name and salary of employees in department D2 from the (virtual) HIGH_EMP table:

```
SELECT (NAME< SALARY)
FROM HIGH_EMP
WHERE DEPT# = 'D2'
```

The data is retrieved by modifying [STON75] the user's query into a query on the base table that was used to defined the view, that is, the actual query to be executed after modification is

```
SELECT (NAME< SALARY)
FROM EMPLOYEE
WHERE DEPT# = 'D2' AND SALARY > 30000
```

A materialized view is a relation that stores the result of executing the view definition query. In the case of the employee data, the relation of Figure 2 will actually be stored. A materialized view, therefore, is a copy of the base data, but rather than replicating relations, an arbitrary subset of the data is replicated; that subset is determined by the view definition query. The advantage of a materialized view is that its definition query does not have to be executed on each reference to the view. If updates are done to a base table used to defined the materialized view, it is possible that the materialized data become inconsistent with the base data, and consequently, an update to the materialized view is required.

The foregoing discussion indicates that a tradeoff is involved in deciding whether to materialize a view. On one hand, queries that reference the view benefit by having the view data materialized, but on the other hand, updates to the base data may effect the materialized data and thus increase the overall cost of updates. A third factor, though less important for most systems, is the additional storage cost incurred by materializing a view. 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 two questions; the first is when to update a materialized view and the second is how to perform the update. In this paper we are primarily concerned with the first question.

3. DISTRIBUTED MATERIALIZED VIEWS

In a centralized DBMS, the decision whether or not to materialized database views is based on the following trade-offs. The benefit from materialized views is a better response time to queries on the view, while the cost is the additional storage requirements and the updates of the materialized view to reflect updates to the base data. An analysis of materialized views vs. query modification in a centralized DBMS can be found in [HANS87].

In a distributed environment, the concept of materialized views is more important because of the added complexity. Ideally, every user in a geographically dispersed organization should have a local copy of the most up-to-date data. In reality, however, there are many situations where users are prevented from accessing a transaction database containing the current data. The reason is simple; in large organizations (e.g., banking, utilities, reservation systems), the operational transactions stretch the capacity of computer systems to the limit. Enabling decision support analysis to be done on the transaction databases will clog the system and deteriorate the response time of customer's transaction to an unacceptable level. Consequently, a common practice is to periodically dump the contents of a transaction database to a decision support database residing on another computer. The data that is used for analysis, then, lags behind the current data. This particular practice is a form of materialized view maintenance.

The notion of distributed materialized views is a compromise between single copies of data and fully synchronized (consistent) replicated data. Figure 3 illustrates the architecture of such a system. Transactions are processed against a transaction database (it is possible that the transaction database is itself distributed). Users in remote sites have their own processors and databases. The data in those databases is extracted from the transaction database(s) (the broken lines indicate logical relationships). After the transactions update the base data, the remote materialized views are updated. The issues of "when" and "how" to preform the view updates are discussed in Section 5. Having a distributed DBMS support materialized views is important to the organization, to end-users, and to database administrators.

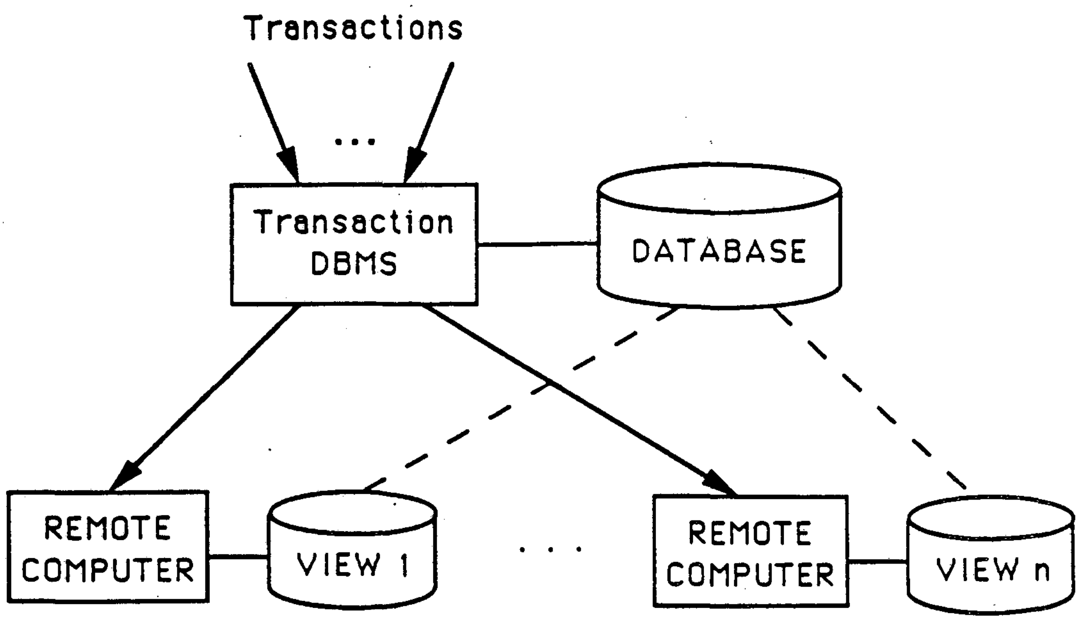


Fig. 3: Distributed Materialized Views

The *organization.* Distributed computing is appropriate for many organization [KING83]. The software to support it is a distributed DBMS [CERI84]. Replicating data is attractive in many instances, but guaranteeing consistency is a very expensive proposition in the presence of updates. The decision does not have to be "all" or "nothing". Materialized views offer the compromise, and provide increased functionality at reduced costs (relative to consistent multiple copies).

The *end-users.* From a user's point of view, the concepts of materialized views, and currency and response time constraints (discussed in Section 5) offer a powerful mechanism to control the currency versus cost trade-off associated with the derivation of data. The view concept is also more powerful than snapshots [ADIB80], because the selection power of view definition equals to that of the query language used.

The Database Administrators. If materialized views are supported by a distributed DBMS, it facilitates the design tasks facing the database administrator. For example, it is possible that a form of a materialized view has to be supported, and the only way to achieve that is by writing special-purpose application programs.

4. THE INFORMATION PROVIDER CASE

Organizational data can be broadly classified into two types -- internal and external. Internal data is captured by the organization and describe organizational entities, attributes and events, e.g., employee data and product data. External data describe entities, attributes and events associated with the environment in which the organization operates. In recent years, we have been witnessing a proliferation of on-line databases storing external data, e.g., economic, financial, and medical data. We refer to a company that collects, maintains, and sells such data as an *information provider*.

These are two basic ways for users to get data from an information provider. The first way is to access a remote on-line database via telecommunication lines (normally dial-up), and the second is to receive, either on a one-time or periodical basis, a tape or an optical disk from the information provider. The first way is beneficial when the data of interest is dynamic and/or requested on an ad-hoc basis. The off-line distribution of data is advantageous when the requested data is static, e.g., economic data related to a given period of time, and the user can afford the delay between the time of data request and the time that it becomes available on the user's computer. It is the first case, of on-line distribution of data, that is of interest to us in this paper.

In the case of on-line information retrieved from a public database, the subset of the information provider's data which is of interest to a user can be considered as a database view. A typical access to the data is through a terminal or a personal computer, where the latter enables the storage of the retrieved data for further manipulation. A common pricing of such a service is based on connect time and the processing cost at the database site. The price of the connect time is also dependent on the transmission speed (the baud rate of the modem). Applying the materialized view concepts to such an environment is dependent on the following factors:

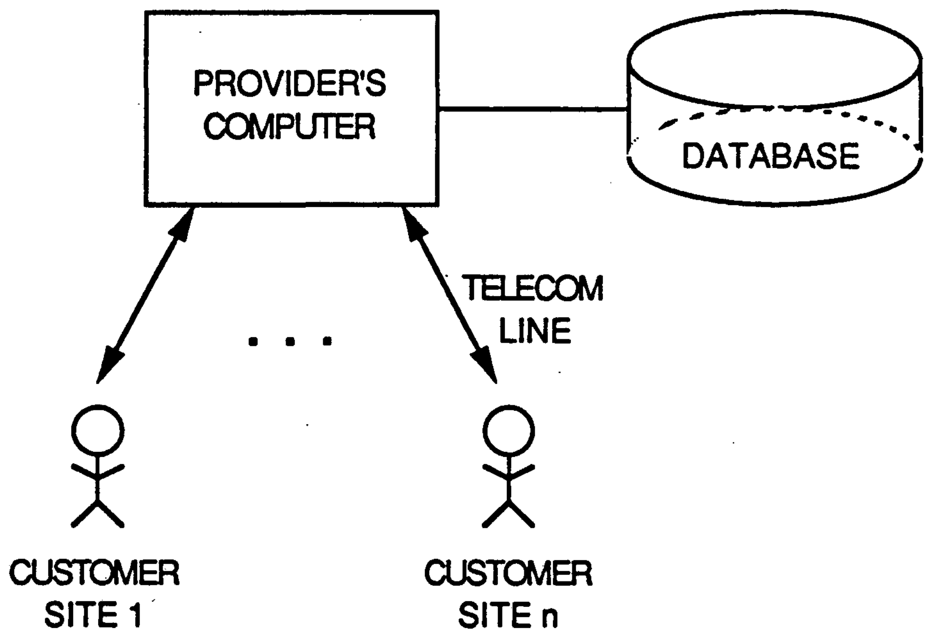
(1) The type of data. In order to benefit from the materialized view approach the data has to be of *state variable* type, e.g., the price of a stock. In this case the materialized view data has to be updated to reflect more *current* states.

(2) The user. The way that the user manipulates the data is an important consideration. A large company with multiple local users can benefit substantially by creating a local materialized view.

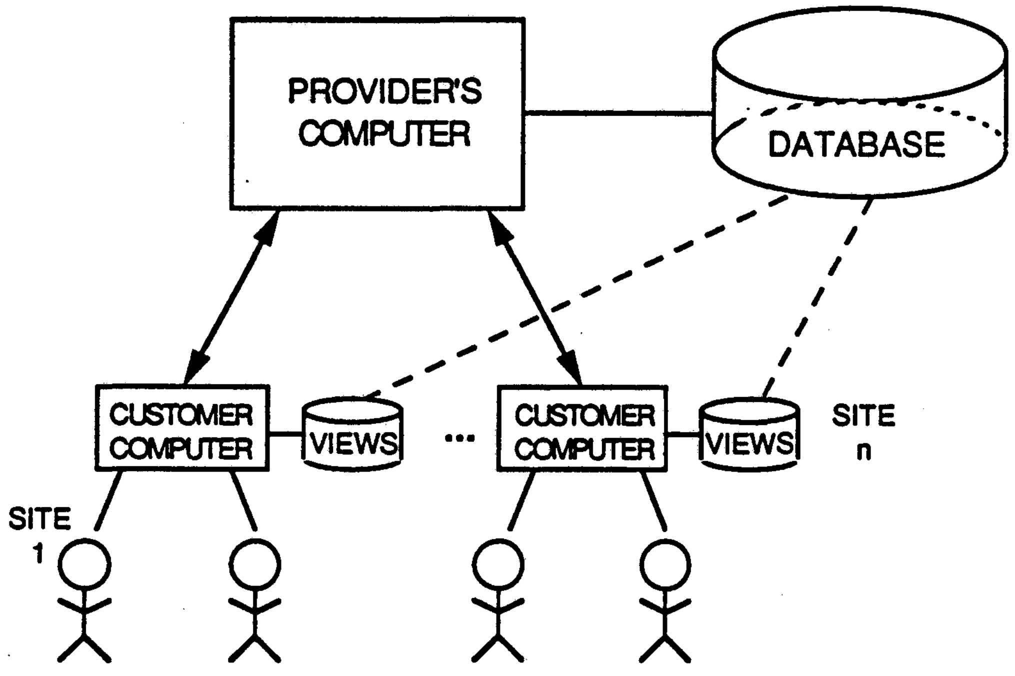
(3) The information provider. The main reason to support materialized views is to gain a competitive edge in terms of customer's satisfaction. In fact the information provider can provide the software to be run on the customer's computer and thus augmenting the data retrieval service by a data management component.

(4) The pricing. The way that the service is priced may affect the procedures used to maintain the materialized views. For example, if the price is a function of the volume of the view data, the information provider has an incentive to maintain it in the most cost-effective way. On the other hand, if the price is a function of the update cost, the provider does not have an incentive to do it in the most efficient way, unless there is a competitive pressure to reduce the price.

In Figure 4, we show four major alternatives of providing data to customers. Figure 4(a) shows the traditional way, where each customer is accessing the provider's database through telecommunication lines. The customer may be using a terminal or a personal computer. What happens at the customer's site is transparent to the information provider and vice versa. In this case the concept of materialized views is not applicable. Figure 4(b) illustrates the case where materialized views are stored at the customer's computers. Those views are defined on the provider's centralized database (cf. the broken lines in the figure). In this case the materialized views have to be updated, and the way that this is done is dependent on the customer's currency and the particular software. In a homogeneous systems it is easier to implement the view maintenance procedures than in a heterogeneous environment. In both cases, it is likely that the information provider will be responsible for the required software. Note that in this case customers do not interact, and the only way to update a materialized view is from the provider's database.

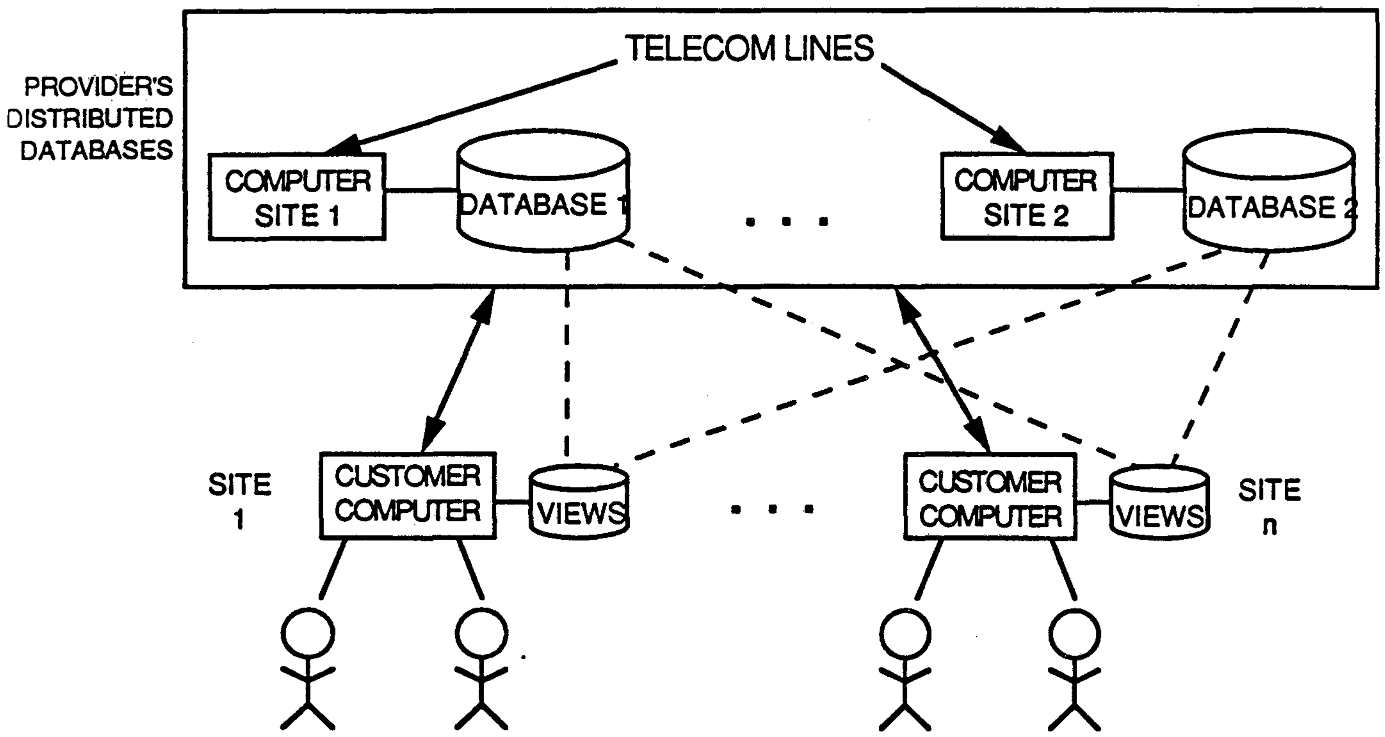


(a)

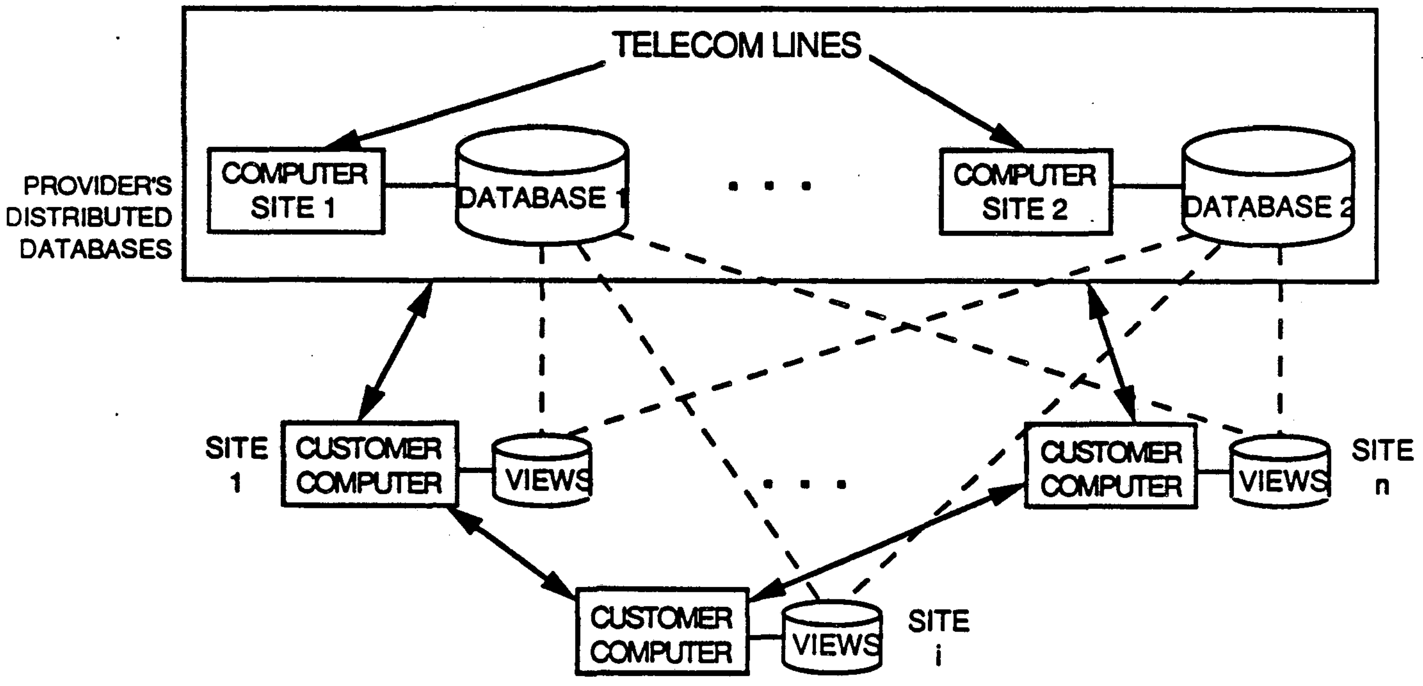


(b)

Fig. 4: The Information Provider Cases



(c)



(d)

Fig. 4: The Information Provider Cases (Continued)

A more complex case is shown in Figure 4(c). Here, the provider's data is stored in a distributed database (data may also be replicated). The main difference from the previous case is that the provider's design and implementation tasks are more difficult. In particular, the maintenance of distributed materialized views become much more complex if views are defined on data that spans multiple sites and if data is replicated. From the customer's point of view, the fact that the data is distributed should be transparent to him.

The most complex case is shown in Figure 4(d). The information provider's data is distributed as well as the materialized views. In this case, it is possible that a customer's materialized view is updated from another customer's data. Since this configuration requires *cooperation* between customers, several intersting questions arise (these questions would also be valid if the provider's database is centralized):

(1) who develops the software?

(2) who pays for a customer's resources which are used to update other customer's data?

(3) who pays for the data communication links?

(4) what objective function is used in updating the distributed materialized views?

(5) how does the provider price the service?

The advantage this configuration offers is reduced overall cost and increased availability. The problem, of course, is that an overall system's objective is not necessarily compatible with the objective of individual users. Consequently, the information provider has to price its service such that users will gain from cooperation, and also satisfy performance constraints imposed by individual users. It should be noted that the connectivity problem is likely to be more severe in this case since it is unlikely that all customers will have compatible systems.

The foregoing discussion described four main cases, but many hybrid cases are also possible. The concept of materialized views can be the basis of a competitive product and reduce users cost in purchasing external information. The questions of "when" and "how" to update distributed materialized views have to be considered by the information provider when deciding how to price the service.

5. VIEW UPDATE POLICIES

Let R be a base relation schema and $R_1, \cdots, R_m$ be fully synchronized stored copies of R. Assume that copy j is stored at base site j (bs_j). If R is not replicated then $m = 1$. Assume that l materialized† views $MV = \{V_i\}, i = 1, \cdots, l$, are defined over R. View V_i is stored at view site i (vs_i). Without loss of generality, we will assume $\{bs_j\} \cap \{vs_i\} = \varnothing$ and $vs_i \neq vs_k$ for $i \neq k$. Occasionally, we will use v to mean V_v; it will be clear from the context. A view update policy is concerned with the timing of the updates. Once the update times are determined, specific update procedures can be used (e.g., [SEGE89a]). We are interested in finding an optimal view update policy for some $V_0 \in MV$. The optimal policy is defined to be the one that minimizes the view update cost subject to a *currency constraint* and possibly a *response time* constraint.

Currency constraints may be value-based or time-based. As an example of a value-based constraint, consider the manipulation of an aggregate view (e.g., averages of base table data). The currency constraint at a given time point may specify that the view averages should be within 2% of the current average, that is, the average that would have resulted if the base data was use to derive it at that time point. Value-based constraints are more difficult to implement than time-based constraints. In many instances, however, value-based constraints can be mapped to time-based constraints if the frequency and pattern of base table transactions are known. In this paper we address time-based currency only.

5.1. 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 $State_v(t_i) \in \{State_B(t_j) \mid t_j \leq t_i\}$, that is, the view state at time t_i was a state of the base table at some time $t_j \leq t_i$. The view currency at time t_i is defined as

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

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

In practice one does not know that $State_v(t_i) = State_B(t)$ except for states that were reflected at view update times. Consequently, if the last update of the view was at time $t_u \le t_i$ and that update reflected the base table state at time $t_j \le t_u$, 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).

5.2. 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 $T_v^{(t_i)} \le T_Q$. 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, Since the view data is not synchronized with the base relations 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. [BLAK86]).

Let $SV \subset MV$ be such that for each $v \in SV$, View_Predicate(V_0) = View_Predicate(v) or View_Predicate(V_0) => View_Predicate(v), and $T_v \le T_Q$. The set SV represents a set of views that can be used to update V_0 such that the new currency of V_0 will satisfy T_Q. There are two advantages to having the option of updating V_0 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.

5.3. 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.

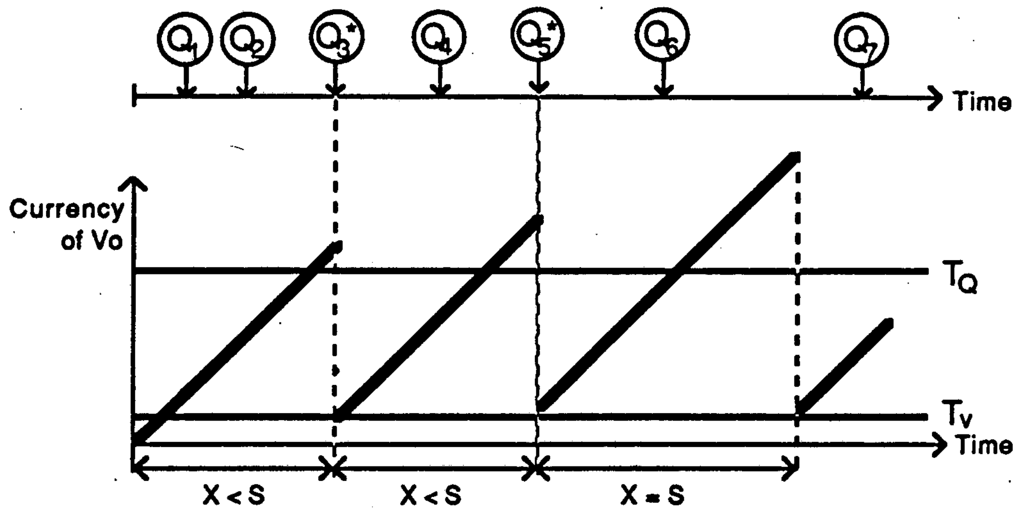
P2: On-Demand - view updates are done only at query processing time.

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

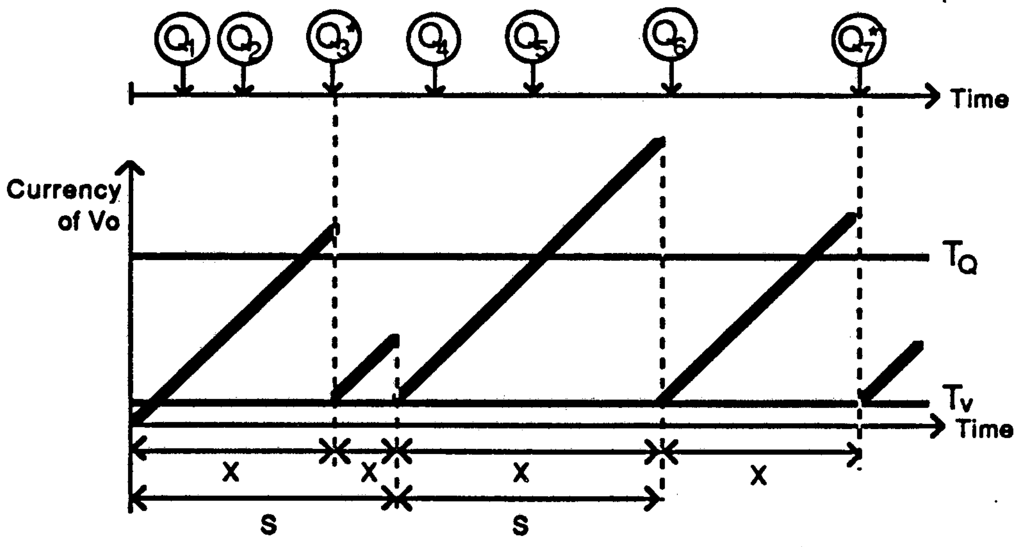
P4: Hybrids - view updates are done according to combinations of the first three policies.

Most previous works have dealt with a single view, $T_Q = 0$, and wither P1 or P2. [SRIV88] deals with a single view, $T_Q = 0$, and a combination of P2 and P3; the random updates are generated from a Poisson process. In our work we are interested in two hybrid policies (P1 and P2 are special cases of these two policies). To explain these policies, we assume that V_0 is to be updated by some $v \in SV$. The first policy is Periodic Or Demand (POD) - an update to V_0 is triggered by either of the following two events: (1) A query arrives and the currency of V_0 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. Note that the cycle time is restarted after each update (either a query-triggered or a clock-triggered). By changing the cycle time s one can control the cost of query-triggered updates.

Figure 5a illustrates the POD policy. Note that the view currency is measured in time units relative to the states of the base table(s); Therefore, a higher currency value means that the view data is older. In the figure, we assumed that initially V_0 is generated from the base table, and subsequently, is updated from v where T_v is a constant. 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 trigger updates.



(a) Periodic Or Demand Policy



(b) Periodic And Demand Policy

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. 5: View Update Policies

The second policy is Periodic And Demand (PAD) - in this case we have two types of updates as in the first policy, but the clock-triggered updates are independent of the query-triggered updates, that is, an update to V_0 is initiated every s time units regardless of the time of the last query-triggered update. This policy models real-life situations where we have slack capacity at a certain time of the day, and by updating the view at that time the cost of subsequent updates is reduced; for example s may be 24 hours from midnight. The effect of this policy on the view currency is shown in Figure 5b for the same query arrivals as in Figure 5a. Note that query 5 does not trigger an update (as it did under the POD policy) because the first clock-triggered update caused the currency value to be below T_Q when query 5 arrived. Query 7, however, triggers an update under the PAD policy because the lack of update at the time of query 5 caused the currency to be above T_Q when query 7 arrived.

In the analytical analysis of the POD and PAD policies (presented in Sections 5 and 6 respectively), our goal is to derive the following results.

(1) Choose a $v \in SV \cup \{R_i\}$, 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 $Pr\{UT_Q^{v,s} > H_1\} \le H_2$, where H_1 and H_2 are user-provided threshold values.

The details of an analytical analysis of both policies can be found in [SEGE89b].

6. CONCLUSION

This paper has addressed issues related to distributed materialized views. Replicating data in a distributed computer system provides higher availability and better response to local users. However, keeping replicated data consistent in the presence of updates is very expensive. Distributed materialized views constitute a compromise between single copies of data and consistent replicated data.

Although introduced in the context of a distributed relational DBMS, the concept of materialized views is much more general and applies to situations where one set of data is derived from another set of data; as an example we have introduced the case of the information provider.

If materialized views are supported, two important problems have to be solved. The first is when to update a materialized view, and the second is how to update it. An intelligent DBMS should allow the user to define currency and response time constraints (it is possible that the database administrator will do that on behalf of the end-users). In this context we have presented two general update policies.

Finally, many managerial and technical issues have to be resolved, such as balancing of users' demands, pricing of services, implementing value-based currency constraints, and devising efficient update procedures.

7. 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.

[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.

[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.

[CERI84] Ceri, S. and G. Pelagatti, in *"Distributed Databases -- Principles and Systems,"* McGraw-Hill, Inc., 1984.

[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.

[KING83] King J.L., "Centralized versus Decentralized Computing: Organizational Considerations and Management Options," in *ACM Computing Surveys,* vol. 15, no. 4, December 1983, pp. 319-349.

[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.

[MADN88] Madnick, S. and Y.R. Wang, ''Evolution Towards Strategic Applications of Databases through Composite Information Systems,'' in *Journal of MIS,* vol. 5, no. 2, Fall 1988, pp. 5-22.

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

[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,'' in *Lawrence Berkeley Laboratory Technical Report LBL-26104.*

[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.