AD-A111 157 RAND CORP SANTA MONICA CA F/G 5/2
DEFERRING UPDATES IN A RELATIONAL DATA BASE SYSTEM,(U)
JUN 81 S CAMMARATA
UNCLASSIFIED RAND/P-6634 NL

END
DATE FILMED
DTIC

DEFERRING UPDATES IN A RELATIONAL DATA BASE SYSTEM

Stephanie Cammarata

June 1981

P-6634

The Rand Corporation
Santa Monica, California 90406

DEFERRING UPDATES IN A RELATIONAL DATA BASE SYSTEM

Stephanie Cammarata

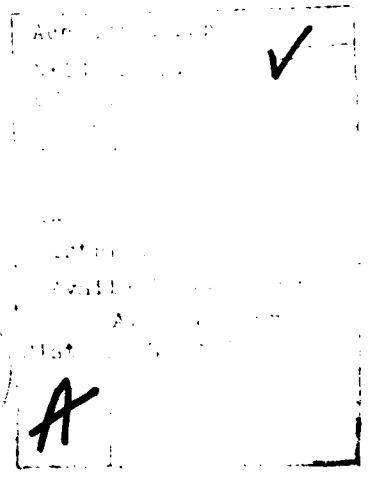
The Rand Corporation
1700 Main Street
Santa Monica, CA 90406

This paper is to be presented at the Seventh International Conference on Very Large Data Bases, September 1981, sponsored by the Institut National de Recherche en Informatique et en Automatique.

ABSTRACT

Deferred updating in a relational data base is a technique which delays the application of updates until a request is made for the value. In general we do not know, a priori, which of the updated values will be retrieved; therefore; it is beneficial to defer the access and computation. This technique promotes an "update-by-need" policy.

The method uses generalization, property inheritance, and procedural attachment for dynamically maintaining the update and query processes. The use of these representational concepts aids in maintaining the structure of the relational model, yet provides opportunities to extend the semantic power of the model.



ACKNOWLEDGEMENTS

I would like to acknowledge the support and helpful advice provided by Professor Stott Parker of UCLA and the insightful discussions with my colleagues at The Rand Corporation.

I. INTRODUCTION

In the relational model, data base updates are performed by accessing tuples with selected property values, computing the new value for some specified property, and storing the resulting value in the relation. For updates which select a large set of tuples to be modified, this process could involve many accesses and computations. For instance, computing interest and account balances in a data base maintaining bank accounts means accessing every account record. The update is invoked by an update command which specifies the set to be selected and the new value; however, the execution of the update is transparent to the user and the effect of the update is covert until an actual value is requested.

A deferred update strategy obviates the immediate modification of the selected tuples by storing the update as a procedure. The update procedure may be retained in high speed main memory. When a query is received for one of the presumably updated values, control first passes to the in-core update procedure which subsequently accesses the tuple, computes the new value, stores the new value in the original relation, and returns it as the result. Retained with the new updated value is a time stamp to indicate the time of its most recent update.

By deferring updates until the values are requested, we have eliminated tuple accesses and the corresponding computations for those values which are never requested. Nevertheless we have preserved the intension of the update by storing it in a procedural fashion; and upon demand in the form of a query, its extension can be generated [5,6].

In this paper we characterize the types of updates for which this strategy is most successful and describe the mechanisms required to operationalize this process. Although we do not explicitly introduce this method in the context of hierarchical or network models, it should be clear that this approach can be extended to any data base model. For discussion purposes, the relational model is convenient and will be posited for the rest of this paper. Section 2 presents the motivation for this work and discusses similar capabilities in other relational systems. In Section 3 we outline the underlying foundations, including application of principles developed in other computer science disciplines. Section 4 discusses a conceptual design and analyzes an example of the update and query processes within the framework of a deferred update. In Section 5 we compare performance characteristics for conventional updating schemes and deferred updating. Section 6 is a conclusion.

II. MOTIVATION

The concept of a deferred update is a synthesis of a variety of updating facilities found in relational query languages. Three of these facilities are 1) the "set" update, 2) the procedural update, and 3) delayed query evaluation.

The "set" update refers to an update over a set of tuples. This type of update can be invoked in most query languages using a form of the select operator. An example of the set update is the following:

Change the interest rate for time deposit accounts to 5.75.

Executing this update requires the selection of all tuples where the value of the "account-type" property is "time-deposit"; and then changing the "interest" property value for that tuple to 5.75.

The procedural update is one where an expression or a procedural specification is provided as the update value. Square and QBE are languages which allow arithmetic expressions as updates [14]. When the expressions are executed, they are instantiated and evaluated to produce the new value. Below is an example of a procedural update:

Update Mary Brown's total balance to be the sum of her demand deposit balance, time deposit balance, and her year-to-date accrued interest.

Execution of this update necessitates performing the summation which perhaps will require accessing additional tuples.

Delayed query evaluation is a feature which is implemented in the relational algebra language ISBL [14]. Its purpose in ISBL is to allow

expressions to be constructed and saved. It is also typically used to define user views of relations [13]. Our notion takes the concept one step further by delaying evaluation until the explicit result of an update is requested.

Let's analyze the following update in light of our previous discussion:

Set the current-balance of all accounts to
current-balance + (interest-rate * current-balance)).

Languages which accept such an update would effect a retrieval for each tuple of the account relation; a calculation of the new balance using values for interest-rate and current-balance; and a store of the new value. Furthermore, this process would be performed in real time as the update was issued. Under the deferred update scheme, a generalization of the account relation is dynamically generated and used to store the update in procedural form. A bookkeeping data structure is also generated which provides the link between the update procedure retained in main memory and the full account relation stored in secondary memory. Tuple accesses, procedure instantiation, and value assignment are delayed until a subsequent query requests the current-balance value for any account. When the new value is requested, the update procedure is retrieved and evaluated with the appropriate values. The new value is returned as the result or passed on to subsequent queries, and is also stored in the full relation with a time stamp indicating when it was most recently updated. In this fashion, it is easy to identify which values in the relation are current and which are outdated.

The merit of this technique lies in the delayed updating. Since the majority of data base maintenance costs are incurred through disk accesses, minimizing the number of tuple modifications is extremely desirable. And since we generally cannot predict how many, if any, of the updated values will be retrieved; why expend resources accessing and updating every tuple instance at the time the update is issued? Instead, defer the retrieval and computation until it is required. This technique would have particular utility in a dynamic data base where "set" updates are likely to occur. We can also view this scheme as a temporary updating mechanism. A complete conventional update of the full data base could then be performed off-line or at off-peak periods.

III. DISCUSSION OF RELATED WORK

The underlying mechanism of the deferred update capability is derived from variations of abstraction, property inheritance, and procedural attachment. There currently is a thrust in relational data base research toward augmenting the relational model to represent additional semantic information [1,3,10,11]. Data base abstractions have played a major role in extending the semantics of the relational model. Property inheritance and procedural attachment are concepts extracted from AI research on knowledge representation; in particular, frame-based representation and semantic networks [16].

Some significant work has been conducted for two types of data base abstractions: aggregation and generalization. Aggregation refers to the "part-of" relationship, and generalization refers to the "is-a" relationship. "House" is an aggregation of the set of objects {living room, dining room, bath, kitchen, bedroom, den}; but "house" is a generalization of the set of objects {ranch, split-level, townhouse, duplex, condominium}. Codd [1] points out that there are two aspects to the notion of generalization: subtype and instantiation. Subtype is denoted by set inclusion, while instantiation is denoted by set membership. Our previous example of generalization corresponds to the notion of subtype. The set {my house, the White House, Buckingham Palace} is an example of instantiation. Walker [15] discusses querying from abstracted data bases versus abstracting from the results of querying a full data base, and Smith and Smith [10] have proposed data structures of the type "aggregate" and "generalization" to represent the

corresponding abstractions and the relationship between these two hierarchical structures. Also, Spyratos and Bancilhon [12] presented a scheme for translating the relational model to the entity/relationship model via an abstraction process.

The abstraction process we have utilized is generalization with instantiation as the inverse. When an update is issued, the relation under scrutiny is generalized over the property or properties which will participate in the selection of update tuples. For example, let's consider a relation containing the property "account-type" where "account-type" can take on values from the set: {demand-deposit, time-deposit, certificate-of-deposit, money-market-certificate}. If an update selects on one of those four elements, the generalization would contain four instances for the four values of "account-type". In this fashion, all tuples of the same "account-type" are represented as a single instance. This exemplifies the "is-a" relationship underlying a generalization. Inversely, the individual tuples represent instantiations of the generalization. In Section 4 we will discuss the other properties and values for the generalization structure.

Property inheritance is defined as the use of a hierarchical data structure based on generalization for storing property values which are common to all descendents. This capability obviates the need to store the values explicitly for each descendent. We view the generalization as a hierarchical structure with respect to the full relation. Generalization enables a class of tuples to be viewed generically as a single object. Property inheritance encourages us to store characteristics of the class of tuples with the generic ancestor. In

our application, the characteristic which all tuples share is the update procedure. Smith and Smith [10] point out that explicit naming of generic objects facilitates application of operators to these objects. This is precisely the capability we have employed for storing and applying the deferred update.

Procedural attachment is the process of storing a procedure to be evaluated, instead of a constant or literal expression, as the value of an attribute. This delays the computation of values until they are requested. When the attribute is referenced, the procedure is instantiated and evaluated. Procedural attachment has been discussed most often in the context of property inheritance [1,7]. It is naturally regarded as a piece of baggage hanging from an ancestor node. Its evaluation is triggered when a reference is made to a relevant characteristic of a descendent. Likewise, for our purposes the procedural update is attached to the generic ancestor and activated when the value, represented by the procedure, is requested.

IV. A CONCEPTUAL DESIGN

The design for a deferred update capability requires the introduction of one new conceptual module for dynamically producing a generalization data structure. It also requires modification to the update and retrieval processes. In addition, a bookkeeping data structure for each relation is maintained for associating full relations with generalizations and retaining information such as the time that the update was issued. For simplicity and uniformity, we have represented the generalization itself in the form of a relation. In this way, a tuple in the generalized relation stands for a class of tuples from the full relation. Figure 1 illustrates the components and control involved in the update process. Figure 2 illustrates the query execution process under a deferred update regime.

We will discuss the various components by use of an example update and query similar to one cited earlier. Suppose we have the relation R1 and update U1 given below:

R1:

Acct#	Acct-Type	Interest Rate	Current-Balance
8501234	time-deposit	.05	1150.00
8502345	time-deposit	.05	100.00
8503456	demand-deposit	.00	500.00
8504567	time-deposit	.05	3000.00
8505678	demand-deposit	.00	750.00
8506789	money-market	.12	1000.00

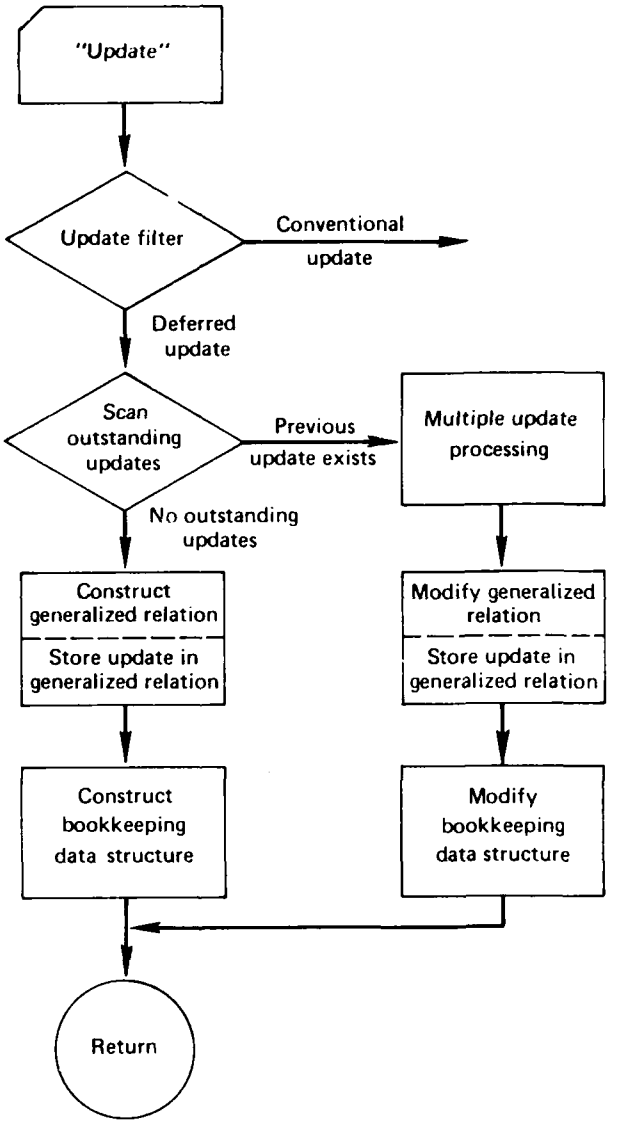


Fig. 1—Update processing

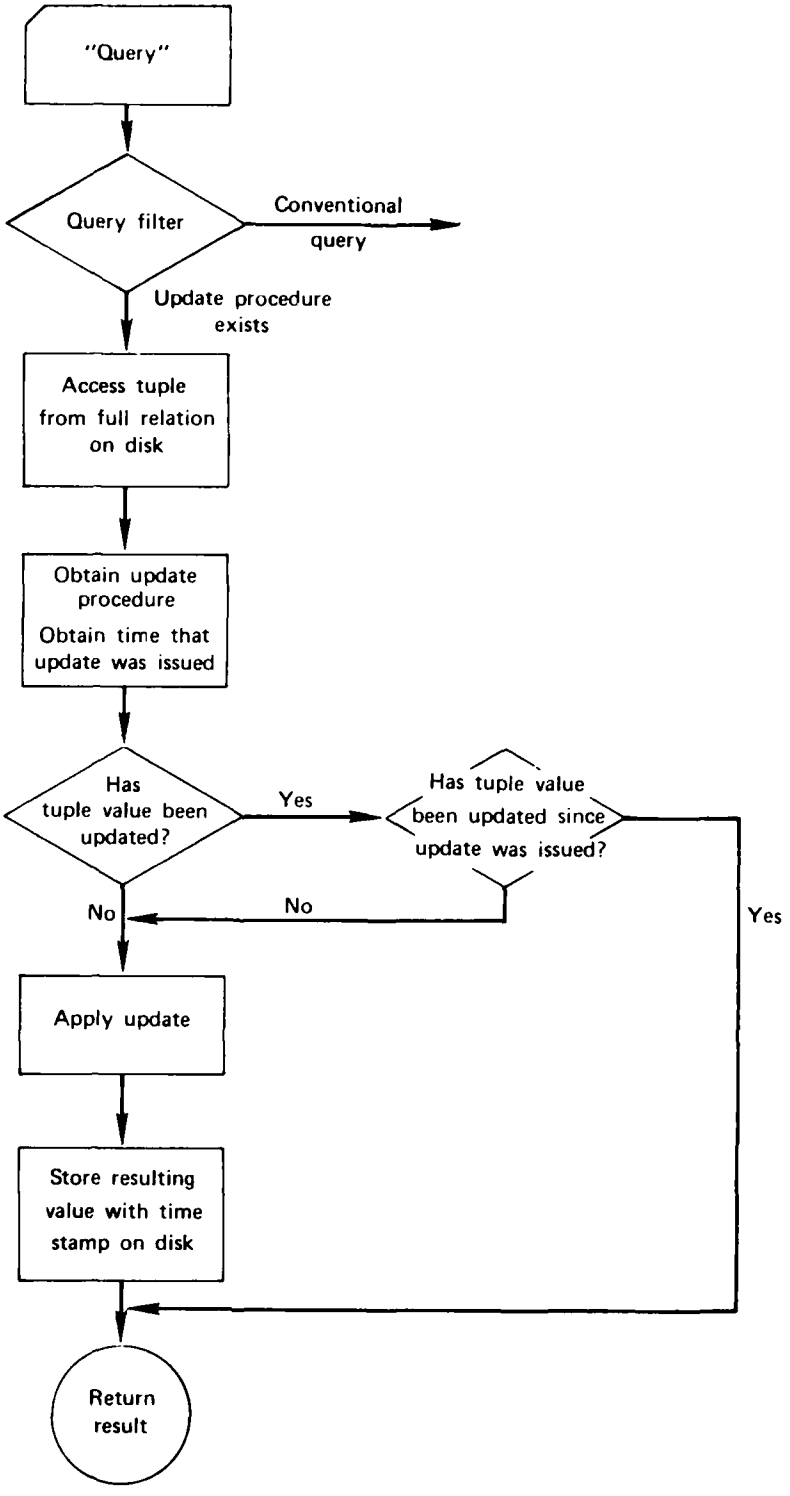


Fig. 2—Query processing

U1: Set the current-balance of all time-deposit accounts to (current balance + (interest-rate * current-balance)).

An update filtering process is used to determine if the update is a potential candidate for being deferred. The main constraint for insuring success using a deferred update scheme, i.e., improved performance, is to reject updates which select on the keys of a relation. This constraint prevents the number of instances in the generalized relation from approaching the number of instances in the full relation. The type of updates which benefit most from this method are the "set" updates which we discussed earlier. The inclusion of an update filter with the above constraint is one example where this approach diverges from the concept of differential files for maintaining data bases as presented by Severance and Lohman [9]. A differential file makes no discrimination among updates, thereby permitting the number of modifications stored in the file to reach its maximum operational limit in a shorter period of time. The update filter can be tailored to suit the needs of particular applications by adding or relaxing constraints. As the number of constraints increases, the system approaches a strictly conventional updating method.

Assuming that our example update passed through the filter, the system must now determine if there are outstanding updates of "current-balance" for the set of accounts whose type is "time-deposit". If one exists, say, from a previous day, then the new update procedure must be "piggybacked" on the previous update. Also, the bookkeeping data structure must be adjusted to reflect that situation. This is to insure that all values in the original full relation will be updated properly

when they are requested, since some of them may have been explicitly updated in the interim.

Let's assume there are no outstanding updates and continue with the process of building the generalization. In the processing of a "set" update we are conceptually partitioning a relation into two or more disjoint sets of tuples. These disjoint sets are the basis upon which the generalized relation is built. In our example, by selecting on the property "account-type," we partition the relation R1 into three disjoint sets corresponding to the three acceptable values for "account-type". Each set will be represented by a single tuple in the generalized relation. The generalized relation is a binary relation whose key is the selection property specified by the update and whose other attribute is the property to be updated. The value of the second property, however, is not a literal or constant but instead is the procedural specification of the value as dictated by the update. Below we show the representation for the generalized relation R2 of our example.

R2:

Account-Type	Current-Balance
time-deposit	current-balance + (current-balance * interest)
demand-deposit	current-balance
money-market	current-balance

In practice this relation can be generated by projecting on the property "account-type", joining the result with a constant unary relation on "current-balance", and substituting the appropriate procedures for the "current-balance" values. Alternatively, it can be built simply by retrieving allowable domain values for the "account-type" property, and joining these with "current-balance" values. It may not be obvious why the generalization is constructed in lieu of simply storing the update in its original form. One reason is because we feel intuitively that there are some underlying semantics behind these "set" updates and that it is very likely that updates which follow may also focus on the same tuple divisions. For example, in the bank scenario we are addressing, it is not unlikely that a subsequent update would be the following:

Set the current-balance of all demand-deposit accounts to (current-balance - service-charges + (current-balance * interest-rate)).

As an effect of the first update we have the generalization structure built and need only to modify the "current-balance" value for the demand-deposit accounts. A second reason for building the ancestor structure in the form of a relation is to maintain the consistency of the relational model, and thirdly, the overhead of this scheme over storing the original update is negligible.

The final step in the update process is to build the bookkeeping data structure for associating the full relation with the generalized version. For our example the structure would be represented as follows:

FILE#0013

ORIGINAL-RELATION: R1
GENERALIZATION: R2
DOMAINS: account-type, current-balance
KEY: account-type
CREATION-DATE: "2-16-81/9:13:57"

This data structure provides the link between the update procedure stored in main memory and the full relation stored on disk. It is also necessary for maintaining information about multiple updates to the same set via its "CREATION-DATE" time slot.

To step through the operation of the query process, let's use the following query, Q1, as our example:

Q1: What is the current balance of the account with account number 8501234?

The query filter is analogous to the update filter in that it determines if the tuples involved in the query have associated update procedures. Notice at this point that although we are processing a query for a deferred update, we must now access the required tuples for one or more of the following reasons: 1) to decide if the value has been explicitly updated, which is reflected by the time stamp and current value in the original tuple, 2) to retrieve the old value if it is outdated, since it will probably be used for computing the new value, 3) to store the new value with its time stamp. From the bookkeeping structure FILE#0013 we know when this update was issued and from relation R2 we know the procedure for updating the current-balance. Since the value has no time stamp, we must calculate the new value, append the current time, and save the new value. The process terminates

by returning the correct result to the user. After the example query is processed, the state of the full data base is given below:

R1 (after applying Q1):

Acct#	Account-Type	Interest-Rate	Current-Balance
8501234	time-deposit	.05	1207.50 -"2-16-81 /12:03:41
8502345	time-deposit	.05	100.00
8503456	demand-deposit	.00	500.00
8504567	time-deposit	.05	3000.00
8505678	demand-deposit	.00	750.00
8506789	money-market	.12	1000.00

Since this discussion only consitutes a conceptual design, there remains a number of interesting issues which warrant further investigation. For example, different specifications for the update filter could substantially alter the behavior of this technique. Also, various representations for the update procedures and ancestor structures could be developed to emphasize criteria such as efficiency, generality, or robustness. Another area which we haven't yet addressed is integrity maintenance. This involves the garbage collection of exhausted generalizations as well as recovery procedures after system failures.

V. PERFORMANCE EVALUATION

An underlying assumption which supports the feasibility of deferred updating is the caching of update procedures. This is a reasonable assumption, since the update filter is selective about which updates are deferred and which are applied immediately. By using a filtering process, we can control the number of generalizations which are constructed. The update filter allows the data base administrator to place constraints upon those updates which are to be deferred. It is interesting to note that as the number of constraints decreases to zero, the number of tuples stored in a generalization relation approaches the number in the full disk relation. This situation occurs when we choose to defer the update of a tuple selected on the key field. Inversely, as the number of filter constraints increase, the behavior approaches that of a conventional updating system. Therefore, as noted in the previous section, performance is greatly affected by the data base application, the types of updates, and the corresponding update filter. This contrasts with the approach taken by Severance and Lohman [9] in their presentation of differential files. Their method confines data base modifications to a relatively small area of secondary storage. They suggest that a differential file is analogous to an errata list for a book and contend that this is an efficient method for storing a large and changing data base.

Deferred updating exhibits some of the flavor of a differential file in the sense that the original relation is perceived as a preestablished point of reference and the updates represent a form of

differential encoding. The technique differs, however, in the following respects: 1) As previously mentioned, the update filter can be fine-tuned so that only distinguished updates are deferred. Under the method of differential files, all modifications resulting from updates are added to the auxiliary file. 2) In deferred updating, the modification procedures are retained in a data structure, whereas for differential file processing, the updates are applied immediately and the modified values are stored. 3) For a given tuple on disk, the tuple is explicitly updated after the first query. In this way, computation of the update is performed only once. This diverges from the "static base" file principle of differential files. 4) Deferred update procedures are stored in main memory in a condensed representation. Severance and Lohman admit that by assigning a differential file to a secondary storage device, data base update costs may be reduced at the expense of increased access time. These increased times can also result from situations when a double access to the differential and main file is necessary.

The next discussion compares the costs and benefits of deferred updating over conventional updating. If updates are executed in real time as in the latter method, we foresee many tuples being updated but never retrieved. If subsequent updates are performed before the tuple is retrieved, there is an additional waste of resources. Also, one update command applied to a large relation can be time consuming. Alternatively, if updates are deferred, then each tuple is updated when it is first requested. This means increased fetching and preparation time during query processing. Subsequent queries only require indirect

access to the tuple through the generalizations. Executing the update command is relatively fast and in most cases only requires accessing the data directory.

Performance analysis becomes more complex when we are dealing with multiple updates over attributes and tuples of the same relation. This necessitates the "piggybacking" of update procedures. The generality of the method affords the capability to create multiple generalization structures and, during query processing, to recreate the proper sequence of updates by using the associated time stamp. The question becomes: Will the additional overhead during this chaining process negate any potential improvement which results from delaying the update? Future experimentation is planned to simulate various sequences of update and query transactions and to monitor performance for various kernel operations. We hope to demonstrate that even worse-case performance will still be an improvement over current transaction optimizations. Throughout the previous discussion we have assumed that disk accesses are expensive and computation almost negligible. Although in practice this isn't wholly true, we think it's reasonable to evaluate this method by the reduction in disk accesses which it promotes.

VI. CONCLUSION

Deferring updates in a relational data base is a technique which stores the intension of an update as a procedure. When the updated value is referenced, the procedure is evaluated and applied. For data base systems where updates are frequently performed over large sets of tuples, this technique offers a substantial reduction in the total number of disk accesses. The work of two disk accesses (one for tuple update and one for query request) is essentially combined into one. In general, we do not know which of the updated values will be requested; therefore, we are eliminating unnecessary real time accesses and modifications by delaying the updates. Some additional overhead costs are incurred by creation of a generalized ancestor structure for storing the update procedure and processing during a query which requests an updated value; however, in most cases we may discount the added computation, since disk I/O is the main bottleneck for processing large data bases.

The foundations for this work are based on knowledge representation issues. The design we presented begins to lay the groundwork for recent theoretical advances in extending the relational model. This work integrates generalization, property inheritance, and procedural attachment with common data base operations in a fashion which streamlines the conventional operations and at the same time establishes a structured representational framework which could be extended for additional semantic encoding.

VII. REFERENCES

1. Codd, E.F., "Extending the Database Relational Model To Capture More Meaning," ACM Transactions on Database Systems, Vol. 4, No. 4, December 1979, pp. 397-434.

2. Date, C.J., An Introduction to Database Systems, Addison-Wesley Publishing Company, Reading, Massachusetts, January 1976 (second printing).

3. Kent, William, "Limitations of Record-Based Information Models," ACM Transactions on Database Systems, Vol. 4, No. 1, March 1979, pp. 107-131.

4. Kim, Won, "Relational Database Systems," Computing Surveys, Vol. 11, No. 3, September 1979, pp. 185-211.

5. Maier, David, and David S. Warren, A Theory of Computed Relations, Technical Report #80-012, State University of New York at Stony Brook, Department of Computer Science, November 1980.

6. Maier, David, and David S. Warren, Incorporating Computed Relations in Relational Databases, Technical Report #80/017, State University of New York at Stony Brook, Department of Computer Science, December 1980.

7. Nilsson, Nils J., Principles of Artificial Intelligence, Tioga Publishing Company, Palo Alto, California, 1980.

8. Rowe, Lawrence A., and Kurt A. Shoens, "Data Abstraction, Views and Updates in Rigel," Memorandum No. UCB/ERL M79/5, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, January 26, 1979.

9. Severance, Dennis G., and Guy M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976, pp. 256-267.

10. Smith, John Miles, and Diane C.P. Smith, "Database Abstractions: Aggregation and Generalization," ACM Transactions on Database Systems, Vol. 2, No. 2, June 1977, pp. 105-133.

11. Smith, John Miles, and Diane C.P. Smith, "Database Abstractions: Aggregation," Communications of the ACM, Vol. 20, No. 6, June 1977, pp. 405-413.

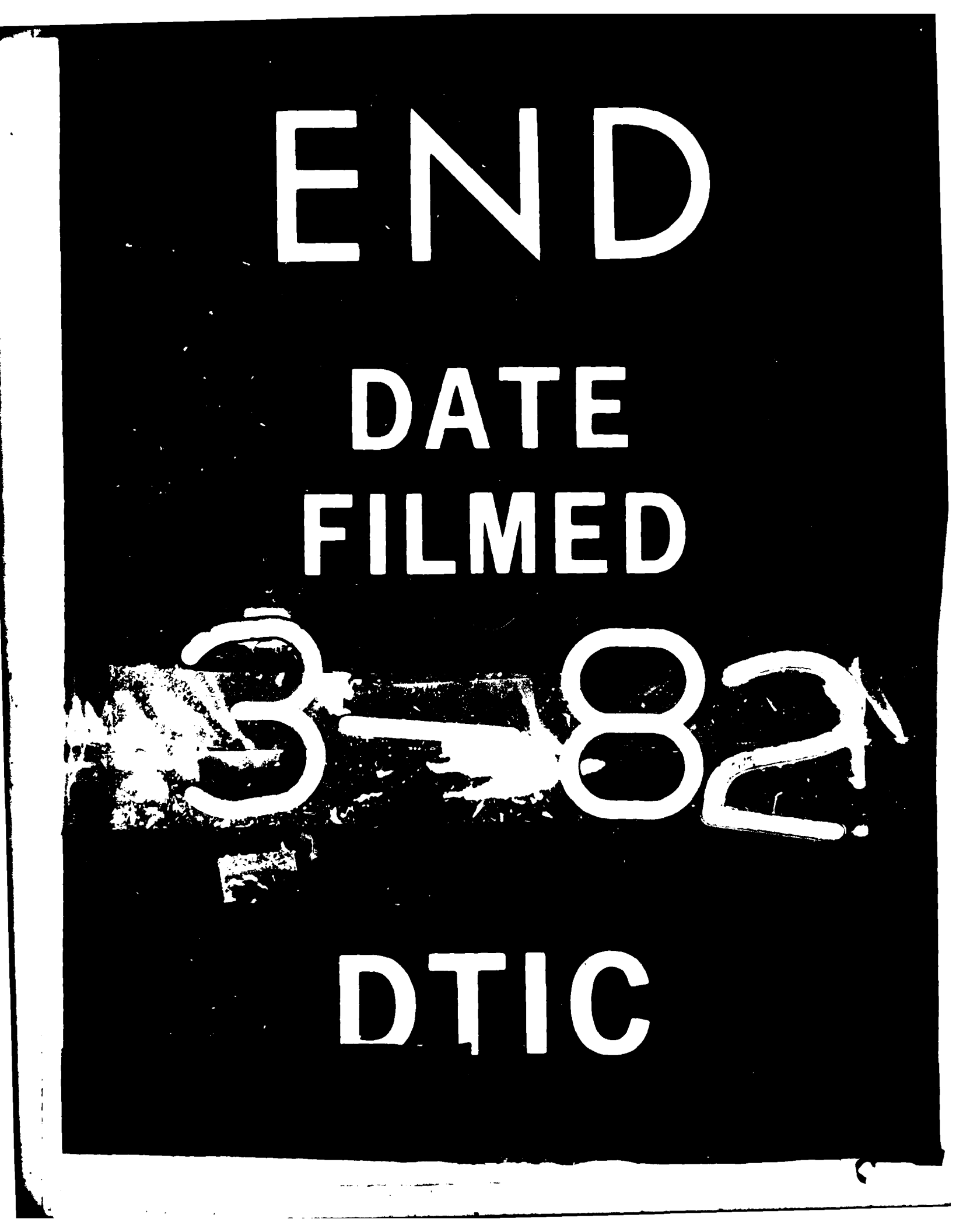
12. Spyratos, Nicolas, and Francois Bancilhon, The Abstraction Process in the Relational Model, Rapport de Recherche No. 342, IRIA Laboria, Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, Janvier 1979.

13. Stonebraker, Michael, "Implementation of Integrity Constraints and Views by Query Modification," Memorandum No. ERL-M514, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, March 17, 1975.

14. Ullman, Jeffrey D., Principles of Database Systems, Computer Science Press, Potomac, Maryland, 1980.

15. Walker, Adrian, "On Retrieval from a Small Version of a Large Data Base," Proceedings of the 6th International Conference on Very Large Data Bases, Montreal, Canada, 1980, pp.47-54.

16. Winograd, Terry, "Frame Representations and the Declarative/Procedural Controversy," in Bobrow, Daniel G., and Allan Collins (eds.), Representation and Understanding: Studies in Cognitive Science, Academic Press, Inc., New York, 1975, pp. 185-210.


END
DATE
FILMED
3-82
DTIC