Incremental Implementation Model for Relational Databases with Transaction Time

Christian S. Jensen, Leo Mark, and Nick Roussopoulos, *Member, IEEE*

Abstract— **The database literature contains numerous contributions to the understanding of time in relational database systems. In the past, the focus has been on data model issues and only recently has efficient implementation been addressed. We present an implementation model for the standard relational data model extended with transaction time. The implementation model integrates techniques of view materialization, differential computation, and deferred update into a coherent whole. It is capable of storing any view—reflecting past or present states—and subsequently use stored views as outsets for incremental and decremental computations of requested views, making it more flexible than previously proposed partitioned storage models. The workings and the expressiveness of the model are demonstrated by sample queries that show how historical data are retrieved.**

Index Terms— **Data and pointer caching, incremental and decremental computation, implementation model, query processing, relational model, transaction time.**

I. Introduction

THERE seems to be general agreement in the database community that efficient and user-friendly time support is needed.

Two orthogonal concepts of time, transaction time and logical time, have been described, see [32]. Transaction time is time in the input subsystem of the database system and is therefore application independent. In contrast, logical time is time in the part of reality modeled in the database. The orthogonality of these domains allows us to consider only transaction time. A database supporting only transaction time is termed a *static rollback database*. Once entered, data are logically never deleted, and it is possible to rollback or *time-slice* the database in order to see a previous state. The main objective behind the design of the static rollback database with tuple time-stamping is to create a small yet powerful extension of the standard relational model. Thus, the model is a first normal form transparent extension of the standard relational model. By giving access through the query language to so-called backlogs recording the change history of relations, the model makes it easy to retrieve detailed rollback data. For surveys and further references to work on time extended relational data models, see [38], [32], [3]; for a comparison of some of the best documented temporal data models, see [37].

As a contrast to the extensive amount of work on data models, there has been little work done on the efficient implementation of temporal databases [33]. Substantial efforts have been put into the investigation of partitioned storage strategies where the fundamental assumption is that fast retrieval is needed for current data while a gracefully degrading performance is allowed for the retrieval of older data [33], [36]. Thus, only current states are efficiently maintained. Our strategy is more flexible because it allows for the storage of old states when they are first computed and *subsequently* provides efficient support for access to arbitrary time-slices of arbitrary relations. The overhead paid for this flexibility and efficiency is increased space consumption. Another approach [35] suggests grid files as a means of implementation. They assume an ordering on surrogate domains, and indexes on other attribute domains are not allowed—both restrictions are inappropriate. For an overview and further reference to efforts relevant to storage of temporal databases, see [24], [12], [22], [31].

In our implementation model, updates to relations are entered as time-stamped change requests into backlogs [19]. Propagation to the affected relations can be done according to protocols ranging from eager to lazy. When a query is computed, an access path consisting of intermediate views and a final view is generated. An access path can be stored as both a collection of pointers and as data, but it might not be stored at all—the model allows for dynamic control of redundancy. Finally, the computation and recomputation of views are done differentially (the term "differentially" is employed to mean "incrementally or decrementally") from already computed and stored views.

The view concept is central in the implementation model. The idea of storing access paths as pointer structures is presented in [28] and [29] where the standard relational data model is the framework, and the ADMS± system [27] utilizes lazy evaluation and incremental update based on access paths stored as pointers [30]. The implementation model of the present paper can be seen as a natural extension of this work. Work on how to efficiently maintain views stored as materialized data can be found in [39], [5], and [4]. This work focuses on how to detect irrelevant and (conditionally) autonomously computable updates. Related works can be found in [16], [6], [11], [2], and [26].

The remaining sections have the following contents: Section II, "Data Structures of the Implementation Model," initially defines the transaction time concept supported. Next,

Manuscript received May 3, 1990. This work was carried out at the University of Maryland and was supported by NSF, IRI-8719458, and AFOSR-89-0303. C. S. Jensen was in addition supported by Aalborg University.

C. S. Jensen is with the Department of Computer Science, University of Arizona, Tucson, AZ 85721.

L. Mark and N. Roussopoulos are with the Department of Computer Science, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742.

IEEE Log Numbe 9101539.

we introduce the concepts of *backlog* and *base relation*. Then, we discuss storage of fixed and time dependent time-slices of base relations and generalize the discussion to include *views*. Lastly, we present the concept of *differential file* and summarize the section. Section III, "Query Evaluation and Redundancy Control," consists of two parts. The subject of the first is the function of a query evaluation subsystem. We briefly outline the decisions to be made by the system in order to efficiently evaluate queries. In the second part, we describe how the system differentially evaluates queries under a number of simplifying assumptions. In Section IV, "Querying a Database," we show how sample queries are evaluated in the model, and we put special focus on the helpfulness of backlogs in answering queries on the change history of relations. The last section contains the "Conclusion and Future Research."

II. Data Structures of the Implementation Model

In this section, we present the basic concepts of the implementation model: the transaction time concept and the different kinds of relations, i.e., *backlog*, *base relation*, *view*, *backlog view*, *differential file*. We characterize these relation types according to traditional understanding of base relations and views, and according to persistence and time dependence.

A. Transaction Time Concept

In Fig. 1 we characterize the time concept supported, and it is discussed below.

First, the time concept is a transaction time concept—thus it models time in the part of reality that surrounds the database, the input subsystem. This means that the time stamps of tuples reflect the times when they were entered into the database. This implies that the model supports queries on the update activity on the database—it only supports queries on temporal aspects of the modeled reality if a well-defined mapping exists between when events took place and when the events were registered in the database. In many applications—e.g., satellite surveillance of crops and weather, and the monitoring of power plants—this is a very realistic assumption, and we have chosen to adopt it in the examples of this paper. Other candidate applications for transaction time support are econometrics, banking, inventory control, medical records, and airline reservations [25]. Second, a domain is regular if the distances between consecutive values of the active domain are identical; otherwise, the domain is irregular. We support an irregular time domain. Third, a time domain can be discrete or stepwise continuous. Facts with discrete time stamps are only valid at the exact times of their time stamps. In contrast, the facts in a stepwise continuous domain have an interval of validity. Our time concept has this property (also termed *stability*), because the values of a relation remain the same until the relation is changed by a new transaction. Fourth, we support true time as opposed to arbitrary time. True time reflects the actual time while an arbitrary time domain needs only to have a metric and a total order defined on it (e.g., the natural numbers). Fifth, we support automatic time stamping—the natural choice for transaction time while manual, user-supplied time-stamp values are natural for logical time.

We have chosen tuple stamping as opposed to attribute value stamping. The major motivation has been to provide a first normal form model that is a simple, yet powerful extension of the basic relational model. The price paid for the simplicity is that we cannot capture independence of attributes of relations—an obvious conceptual drawback and in addition a potential cause of redundancy at the implementation level [8], [13]. Redundancy problems can be alleviated through the use of compression techniques which in some applications have proven not only to reduce space consumption but also to speed up query processing.

B. Backlogs

A *backlog*, B_R, for a relation, R, is a relation that contains the complete history of change requests to relation R. The schema of relation R and its corresponding backlog are shown in Fig. 2.

The tuples of backlogs are termed *change requests* because a backlog contains change requests to its corresponding base relation. As shown, B_R contains three attributes in addition to the attributes of R. Attribute *Id* is defined over a domain of surrogates. Surrogates are logical system generated tuple identifiers that can be referenced but neither seen nor modified by users/application programs [17], [9]. The values of *Id* represent the individual change requests. Attribute *Op* is defined over the enumerated domain of operation types where values of *Op* indicate whether an insertion (*Ins*), a deletion (*Del*), or a modification (*Mod*) is requested. Finally, attribute *Time* is defined over the domain of transaction time stamps, *TTIME*, as discussed in the previous subsection. The value of a time stamp of a change request is the time when the transaction—the request is a result of—commits. Requests are not entered into backlogs until they have been stamped, and they are entered in the sequence, that they were stamped, i.e., sorted on time stamps. This ensures that backlogs only contain valid change requests sorted on time.

The database management system (DBMS) automatically generates a backlog for each base relation. A backlog, B_R, of change requests is illustrated in Fig. 3.

Also, the DBMS maintains the backlog relations. Fig. 4 shows the logical effect on backlogs of update requests to their corresponding relations.

Example: We introduce a sample database to illustrate the concept of backlog and other concepts to be presented later in the paper. The database consists of one user-defined relation, *Emp*, with three attributes: *Emp_Id*, *Name*, and *Salary*. Fig. 5 shows the schema of *Emp* and its backlog, B_{Emp}.

When nothing is deleted from backlogs, constant growth will eventually become a problem. The model can be augmented with vacuuming facilities to solve the problem. The idea is to provide facilities for pinpointing historical data not needed by any application running against the database and delete these without changing the history as recorded in the database. See [18].

$$\left\{\begin{matrix}\boxed{\text{transaction}}\\ \text{logical}\end{matrix}\right\}\times\left\{\begin{matrix}\text{regular}\\ \boxed{\text{irregular}}\end{matrix}\right\}\times\left\{\begin{matrix}\text{discrete}\\ \boxed{\text{stepwise cont.}}\end{matrix}\right\}\times\left\{\begin{matrix}\boxed{\text{true}}\\ \text{arbitrary}\end{matrix}\right\}\times\left\{\begin{matrix}\boxed{\text{automatic}}\\ \text{manual}\end{matrix}\right\}$$

Fig. 1. Characterization of the time concept.

Relation name	R
Attribute name	*Domain name*
A_1	D_1
A_2	D_2
...	...
A_n	D_n

Relation name	B_R
Attribute name	*Domain name*
Id	*SURROGATE*
Op	*{Ins, Del, Mod}*
Time	*TTIME*
A_1	D_1
A_2	D_2
...	...
A_n	D_n

Fig. 2. Schema for the relation R and its backlog, B_{Emp}.

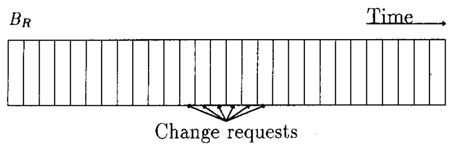


Fig. 3. A backlog, B_R, consists of change requests which are both logically and physically ordered by unique, increasing time stamp values from left to right.

C. Base Relations

As a consequence of the introduction of time stamps, a base relation is now a function of time. To retrieve a base relation it must first be time-sliced. Let R be any base relation, then the following are examples of *time-slices* of R:

$$R(t_{init}) \overset{\text{def}}{=} R_{init}$$
$$R(t_x) \overset{\text{def}}{=} R\text{"}\textit{at time } t_x\text{"},\ t_x \geq t_{init}$$
$$R \overset{\text{def}}{=} R(NOW)$$

When the database is initialized, it has no history, and every relation is equal to the empty set. If R is parameterized with an expression that evaluates to a time value, the result is the state of R as it was at that point in time. The use a time from before the database was initialized or after the present time is undefined. R used without any parameters indicates the current R. Note that this feature helps provide transparency (subsection IV-A) to the naïve user. We also introduce the special variable NOW which assumes the time when the query is executed.

Time-slices of base relations can be either stored or recomputed when needed. There are two ways of *storing* base relations:

- index cache
- materialized data.

First, a base relation can be stored as an *index cache* (or just *cache* for short). A cache is a pointer array where the entries contain physical pointers to the associated backlog. We use the name index cache because the structure inherits both characteristics from caches and indexes. Like a cache, it avoids search—all it has to do is to fetch the data to which it points. Like an index, it stores pointers to actual data records, is buffered into main memory, and is read in its entirety [30]. Second, a relation can be stored as actual materialized data. The co-existence of caches and materialized data is shown in Fig. 6.

The choice between materializing data or creating a cache reflects a tradeoff between replication of data and speed, as does the choice between creating a cache or not using a cache. Time-slices of base relations can be

- *fixed*
- *time dependent.*

If the expression, E, of a time-sliced relation, $R(E)$, contains the variable *NOW* then R is time dependent; otherwise, it is fixed. While fixed time-slices never get outdated, time dependent time-slices do. Thus, stored time dependent relations must be updatable. Update strategies ranging from eager to lazy can be adopted. In an eager approach, change requests are immediately propagated to the time-slices. In a lazy approach, change requests are propagated to a time-slice when the current state of the time-slice is requested.

Example: Fig. 7 shows the extension of *Emp* at two points in time. The extension of the corresponding backlog is shown in Fig. 8. Note that surrogate values cannot be seen by the user.

D. Views on Base Relations and Backlogs

In the previous subsection, we considered only base relations. Here we generalize the presentation to cover views. Views can be created from other views, base relations, or backlogs. What was said about base relations in the previous subsection is also true for views:

- Views can be stored as index caches or as materialized data.
- Views are either time dependent or fixed.

However, some explanation and qualification is needed. *Cached views* have references to the relations and views they

The Effect on Backlogs of Update Requests	
Requested operation on R:	Effect on B_R:
insert R(tuple)	insert B_R(id, *Ins*, time, tuple)
delete R(key)	insert B_R (id, *Del*, time, *tuple*(R, key, -))
modify R(key, new values)	insert B_R(id, *Mod*, time, *tuple*(R, key, new values)

Fig. 4. The table defines system-controlled insertions into a backlog. The function *tuple* takes as arguments a relation name, valid key information for that relation, and an optional list of changes to the identified tuple; it returns the identified tuple of the specified relation with possible changes reflected properly.

Relation name	*Emp*	
Attribute name	*Key*	*Domain name*
Emp_Id	nil	*SURROGATE*
Name	K_1	*STRING(20)*
Salary	nil	*INT*

Relation name	B_{Emp}
Attribute name	*Domain name*
Id	*SURROGATE*
Op	{*Ins, Del, Mod*}
Time	*TTIME*
Emp_Id	*SURROGATE*
Name	*STRING(20)*
Salary	*INT*

Fig. 5. Schema for the user-defined relation, *Emp*, and its backlog B_{Emp}.

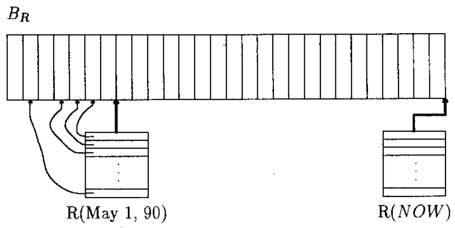


Fig. 6. Differently stored time-slices of a base relation can co-exist. The time-slice R(May 1, 90) is stored as an index cache (to the left), and the time-slice R(NOW)is stored as data (to the right).

Emp(NOW − 20 days)		
Emp_Id	*Name*	*Salary*
emp 2	Mark	90 000
emp 3	Brown	32 000
emp 1	Jensen	10 000

Emp		
Emp_Id	*Name*	*Salary*
emp 4	Smith	30 000
emp 3	Brown	32 000
emp 1	Jensen	11 000

Fig. 7. Time-slices of a relation where NOW = May 1, 1990 4:00p.m.

are derived from and consist of pointers to these; views can only be cached if the relations they are derived from are stored (as data or as pointers). *Time dependent views* stored as materialized data need only references to the relations and views they are derived from to facilitate update. Views of this kind also require the relations and views they are derived from to be stored. Finally, *fixed views* stored as data need no references to the relations and views they are derived from because they never become outdated. Consequently, they can be stored independently, whether the relations and views they are derived from are stored or not.

A view is *time dependent* if at least one of the relations and views it is derived from is time dependent; otherwise, it it fixed. Fig. 9 illustrates a view.

Traditional views are derived, possibly via levels of indirection, solely from time-sliced base relations. If a view ultimately is derived directly, i.e., not via a time-sliced base relation, from at least one backlog, then we term it a backlog view. Backlog views are time-sliced as are base relations and views:

$$B_R(t_x) \overset{\text{def}}{=} \sigma_{Time \le t_x} B_R$$
$$B_R \overset{\text{def}}{=} B_R(NOW)$$

Backlog view time-slices involving NOW are time dependent, as well as backlog views derived from views involving NOW. For an example, see Fig. 10.

Example: We can use B_{Emp} to retrieve all the employees that were modified during the last month. The query and its result is shown in Fig. 11.

E. Differential Files

Stored, time dependent relations (base relations, views, and backlog views) generally get outdated as time passes and change requests are entered into backlogs—the change requests have to be reflected in the relations when they are retrieved.

When a base relation (time-slice) is to be retrieved, the relevant change requests can be found in the backlog of the base relation. For example, if we want to retrieve a previously computed and stored time-slice $R(NOW)$, the needed change requests are the ones entered into B_R since $R(NOW)$ was most recently brought up-to-date. A set of change requests needed to update a stored relation (R) is referred to as a *differential file* (δR). The differential file of a view is derived from the differential files of the relations and views the view is derived from, and it contains the change requests relevant for updating the view. Differential files of base relations and backlog views directly derived from backlogs are physically stored as parts of backlogs—differential files of all other relations (views) are purely conceptual constructs. Fig. 12 shows differential files for a time-dependent time-slice, $R(NOW)$, and for a (traditional) view, $R(NOW) \bowtie S$(May 1, 90), stored as an index cache. (Time-slice S(May 1, 90) is fixed and does not have a differential file.)

B_{Emp}					
Id	*Op*	*Time*	*Emp_Id*	*Name*	*Salary*
id 1	Ins	March 31, 1990 11:31 a.m.	emp 1	Jensen	10 000
id 2	Ins	April 2,1990 2:56 p.m.	emp 2	Mark	90 000
id 3	Ins	April 8, 1990 10:34 a.m.	emp 3	Brown	32 000
id 4	Del	April 15, 1990 12:09 p.m.	emp 2	Mark	90 000
id 5	Ins	April 20, 1990 9:02 a.m.	emp 4	Smith	30 000
id 6	Mod	April 20,1990 12:38 p.m.	emp 3	Brown	42 000
id 7	Mod	April 20, 1990 12:45 a.m.	emp 3	Brown	32 000
id 8	Mod	May 1, 1990 3:55 p.m.	emp 1	Jensen	11 000

Fig. 8. The backlog shown corresponds to the time-slices in the previous figure.

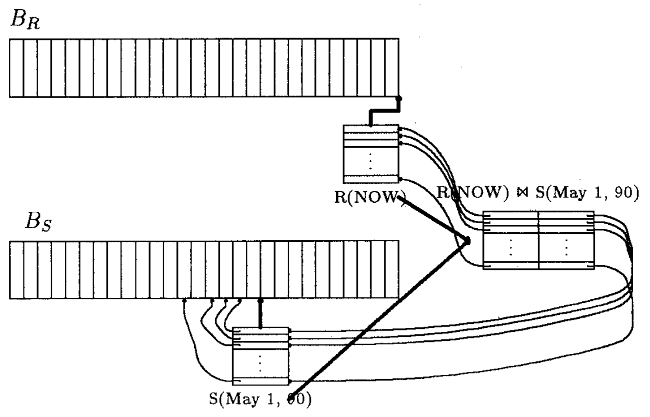


Fig. 9. The figure shows a cached view ($R \bowtie S$) derived from a materialized (R) and a cached relation (S). The derived relation is time dependent because the materialized view is time dependent.

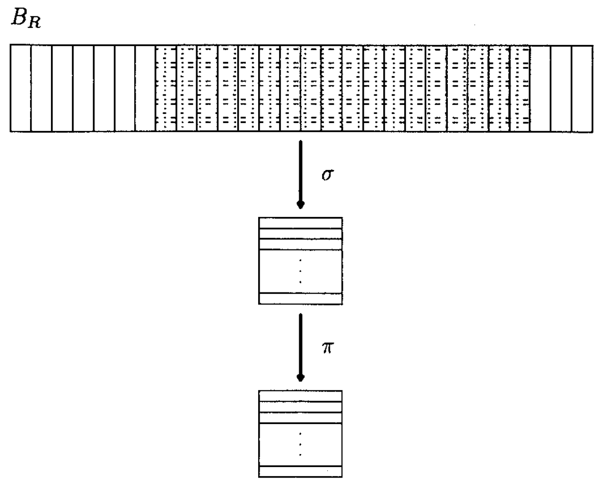


Fig. 10. Views can be derived directly from backlogs.

Fig. 13 shows the differential file of a backlog view (e.g., $\sigma_{NOW-3yrs\leq Time\leq NOW-2yrs}B_R$). As time passes and the window slides, old change requests must be discarded (to the left) and newer ones must be included (to the right).

Example: Suppose that we compute the time-slice $Emp(NOW - 20\ days)$ on May 1, 1990 4:00 p.m. (Fig. 7). The time-slice (assuming it is a view cache) will consist of pointers to the change-requests that indicate the valid tuples on April 12, 1990 4:00 p.m., i.e., it will be pointers to change requests issued before April 12, 1990 4:00 p.m. If we look at this time-slice on May 6, 1990 4:00 p.m., the change requests after April 12, 1990 4:00 p.m. and up till April 17, 1990 4:00 p.m. will constitute the differential file of $Emp(NOW - 20\ days)$ because we now need to display the

$\sigma_{NOW-30\ days<Time \wedge Op=Mod} B_{emp}$					
Id	*Op*	*Time*	*Emp_Id*	*Name*	*Salary*
id 6	Mod	April 20, 1990 12:38 p.m.	emp 3	Brown	42 000
id 7	Mod	April 20, 1990 12:45 a.m.	emp 3	Brown	32 000
id 8	Mod	May 1, 1990 3:55 p.m.	emp 1	Jensen	11 000

Fig. 11. A backlog query and its result. *NOW* = May 11, 1990.

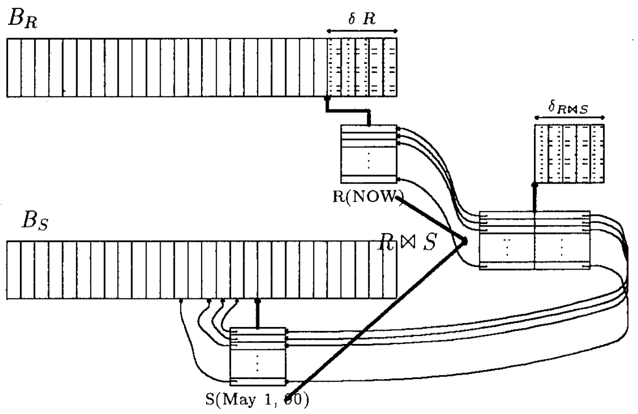


Fig. 12. The differential file ($\delta_{R\bowtie S}$) of a view ($R \bowtie S$) is derived from the differential file(s) (δ_R) of the relations (R and S) it is derived from.

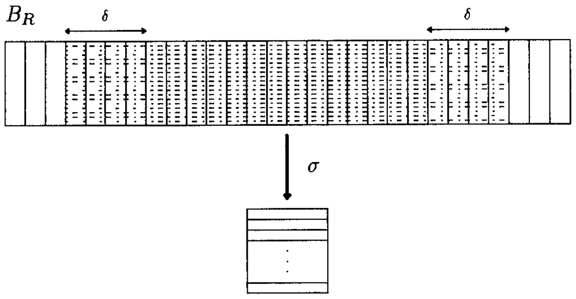


Fig. 13. A "sliding window" backlog view.

Traditional relation concepts	
Concept	*Description*
base relation	Actual data are stored in the database. The relation physically exists in the sense that there exists records in storage that directly represent the relation. The *Emp* relation is an example. A base relation cannot be derived from other relations, and a base relation definition is part of the *schema*.
view	A view is characterized as a *virtual*, *derived*, or *computed* relation, and it can be queried as if it actually were a physically stored relation. A view definition is part of a *subschema*.

Fig. 14. The usual definitions of relation types.

tuples valid on April 17, 1990 4:00 p.m.

F. Summary of Data Structures

It is common practice to distinguish between *views* and *base relations*. In this section we have presented several kinds of relations. To get a better understanding of these, let us look at the generally accepted characterizations of views and base relations. Two dimensions are used to distinguish them. The first dimension concerns physical storage: traditionally, base relations are stored while views are computed or virtual. The second dimension concerns logical derivability: base relations are not derivable from other relations while views are. See Fig. 14 where generally accepted characterizations of base relation and view are summarized [9], [40], [7].

We have given the relation concepts new meaning. Every base relation has a backlog. All data of such relations are stored in the relations backlogs in the form of change requests. This makes backlogs act as base relations, and base relations act as views, derived from backlogs. The new meanings are described in Fig. 15. Henceforth, we will use the definitions presented there.

We can summarize the concepts presented in this section as illustrated in Fig. 16.

We distinguish between *backlog views*, traditional *views*, and *base relations*. The only difference between views and base relations is that the former are derived indirectly from backlogs while the latter are derived directly. A view is valid only at a single point in time, namely at the time-value specified when it was produced using the query language. A backlog has an associated lifespan from the time when the corresponding base relation was created till the current time—if the base relation still exists—or otherwise, till it was deleted. Backlog views inherit this notion of lifespan.

The second dimension in Fig. 16, *time dependence*, distinguishes between *fixed* and *time dependent* views. The time of

Redefined Relation Concepts	
Concept	*Description*
backlog	Backlogs are the relations that now function as base relations in the sense that they are stored and that all other relations are derived, possibly indirectly, from these.
base relation	Base relations are not necessarily stored, and they are derived (directly) from backlogs. Thus, the base relation *Emp* is derivable from B_{Emp} and is not necessarily physically existent.
view	The data of a view still is, directly or indirectly, constructable from base relations, or backlogs. However, even though a view still is derived, it is not necessarily virtual or computed. Views can be materialized.

Fig. 15. Redefinitions of relation types.

$$\left\{\begin{array}{c}\text{view}\\ \text{base relation}\\ \text{backlog view}\end{array}\right\} \times \left\{\begin{array}{c}\text{time dependent}\\ \text{fixed}\end{array}\right\} \times \left\{\begin{array}{c}\text{materialized data}\\ \text{view cache}\\ \text{query modification}\end{array}\right\}$$

Fig. 16. Different types of views.

validity of fixed time-slices of base relations and views and the lifespan of fixed backlog views never change. Because it is possible to use the special variable *NOW* in query expressions, both base relations, views, and backlog views can be time dependent. A time dependent base relation can be visualized as a view that slides along a backlog as time passes. Similarly, a backlog view can be thought of as a filtering window where one or both ends (start time and end time) move along a backlog. Let E be an expression which maps into the domain *TTIME* and R a relation. For each time dependent time-slice $R(E(NOW))$ there is a differential file, $\delta R(E(NOW))$. This differential file is a sequence of change requests in the backlog of the relation. These change requests are not yet reflected in the stored state of the time-slice.

The third dimension of Fig. 16 is *persistence*. When the system chooses to store a view, it can be done in two ways. First, a view can be stored as actual materialized data. Second, a view can be stored as a pointer array, where entries contain pointers to the relations (view or base) from which the view can be derived. The schema entry for a view contains information on how to use the data to materialize the view. As the third possibility, a view might not be stored at all. We have indicated this with the entry *query modification* in the figure.

III. Query Evaluation and Redundancy Control

Evaluation of queries and management of storage are closely tied together in the implementation model. First, we discuss the problem of query evaluation and control of redundancy in the general setting presented in this paper. Second, we make simplifying assumptions to reduce complexity and describe an algorithm for differential computation.

A. A General Framework

We are now in a position to outline the function of the *query evaluation subsystem*. We will present only the choices that the query evaluation subsystem must make in order to evaluate queries—how it actually makes these choices is left for future research.

Assume that a query is issued against our database. To evaluate it, we have to make decisions along the dimensions outlined in Fig. 17.

It is decided whether the result (subresults are treated the same way) shall be stored or not, and in the case of storing, whether a view cache or materialized data shall be chosen. Some restrictions apply. If we choose to store (see subsection II-D) a time dependent result, the subresults that it is derived from must be stored, too. Also, subresults of any (fixed or time dependent) view cached result must be stored. The decision is made by consulting usage and profile statistics for the database. Statistics should include update and retrieval frequencies for relations, relation sizes and cardinalities, attribute sizes and cardinalities, domain cardinalities, etc. [28], [30], [34], [23]. The idea is to estimate whether the results to be computed might come in handy when future queries are issued against the database, and the fundamental tradeoff is one between space/redundancy versus overall computation and retrieval speed.

If we did choose to store the result and if it is time dependent, we must then choose an update strategy. Global system workload is one among several possible triggers for strategies between eager and lazy. The choice is restricted so that a relation derived from another relation not is updated more eagerly than that relation. The choice of update strategy is one of when and how much to process: a lazy strategy might compromise the responsiveness of the system for specific queries, but allows for minimal overall processing time, too.

The subsystem decides how to most efficiently compute the query. In general, a strategy—where already computed and stored, partial results are either decremented or incremented—is selected on the basis of estimated costs. Usage and profile statistics are used for cost estimation.

Previous decisions on what to store and how to maintain stored views are reconsidered and possibly changed. This is necessary to *dynamically* control the level of redundancy and the tradeoff between system overhead and response time.

The sequence in which to carry out the above decisions is not obvious. Even though the activities are interdependent, there are still many alternative interleavings.

There is definitely a need to investigate a general framework as the one presented above because no single choice is optimal in all situations. Due to the overwhelming complexity of the general framework, we suggest that less general, but still promising settings are investigated one at a time.

B. Lazy Evaluation and Cache Indexes

To reduce the complexity of the problem, we make a number of simplifying assumptions on the query evaluation scheme discussed above. We choose to consider only lazy evaluation of time dependent base relations (and views). We disregard base relations, views, and backlog views that are fixed. Base relations and views are stored as index caches. The views

$$\left\{\begin{array}{c}\underline{\text{redundancy}}\\ \text{not store result}\\ \left\{\begin{array}{c}\underline{\text{store result}}\\ \text{index cache}\\ \text{materialized data}\end{array}\right\}\end{array}\right\} \times \left\{\begin{array}{c}\underline{\text{update strategy}}\\ \text{eager}\\ \vdots\\ \text{threshold triggered}\\ \vdots\\ \text{lazy}\end{array}\right\} \times \left\{\begin{array}{c}\underline{\text{initial computation}}\\ \text{computation procedure is}\\ \text{selected using cost}\\ \text{estimation}\end{array}\right\}$$

Fig. 17. Dimensions for decisions made by the query evaluation subsystem.

corresponding to subresults of queries are stored the same way as are the final results. Our choices are outlined in Fig. 18. They coincide with the way the ADMS± system is designed.

The task of computing a query is now reduced to computing and storing time-sliced base relations and views efficiently. This implies the usage of differential techniques. A time-slice can be either incrementally or decrementally computed from either older or more recent, stored time-slices. The empty relation is the oldest possible time-slice, and the current state of the relation is the newest possible, but not necessarily stored time-slice.

In Fig. 4 we showed how insertion, deletion, and modification requests are entered into backlogs. A modification request is only a logical construct: at the implementation level, a modification is represented by an ordered pair of a deletion request and an insertion request. They have identical time stamps to distinguish between logical modifications and pairs of deletions and insertions.

Below, the algorithm *freeze* produces a time-slice $R(t)$ of R at time t from the backlog B_R of R. The time-slice is produced from the neighbor on the left or right whichever is the most promising. See Fig. 19. Note that the existence of differential files of neighbor time-slices is irrelevant; only the references to the backlog telling when the time-slices were up-to-date are required. See Fig. 19.

Algorithm *freeze* identifies the closest neighbor time-slices, chooses the most promising, and calls procedure *increment*, if a left neighbor time-slice is chosen, or procedure *decrement*, if a right neighbor is chosen. These two procedures are described as follows.

Incremental Freezing

```
increment($R$, $t_y$, $t$)
Res ← $R(t_y)$
$tt$ ← $t_y$
while change requests with time stamps between $tt$ and
        $t$ do
    pick oldest change request, at $tt'$, bigger than $tt$
    update $tt$ ← $tt'$
    case request type
     DELETE: remove from Res the pointer pointing to
             delete-requested tuple
     INSERT: insert into Res pointer to change-request
             tuple
return($R(t)$ ← Res)
```

Decremental Freezing

```
decrement($R$, $t_x$, $t$)
    Res ← $R(t_x)$
    $tt$ ← $t_x$
    while change requests with time stamps between
            $tt$ and $t_y$ do
    pick newest unmarked change request, at $tt'$, less
            than $tt$
    update $tt$ ← $tt'$
    case request type
    DELETE: insert into Res the pointer pointing to
            delete-requested tuple
    INSERT: remove from Res the pointer pointing to
            insert-requested tuple
return($R(t)$ ← Res)
```

Top Level Freezing

```
freeze ($R$, $t$)
Find $R(t_y)$, where $t_y = \max\{tt \le t \wedge$
exists a time-slice at time $tt\}$
Find, if it exists, $R(t_x)$, where $t_x = \min\{tt \ge t \wedge$
exists a time-slice at time $tt\}$
if two time-slices are found
then estimate costs of using each time-slice as the outset
  if cost($t_y$, $t$) ≤ cost($t$, $t_x$)
  then increment($R$, $t_y$, $t$)
  else decrement ($R$, $t_x$, $t$)
else increment ($R$, $t_y$, $t$)
```

Computation of Views: Above we covered only base relations. Here we give an example of how a view can be computed. Assume that we have base relations R, S, and T, and $V(t)$ is defined as follows

$$V(t) \overset{\text{def}}{=} \sigma_{F_1} R(t) \bowtie (\sigma_{F_2} S(t) \bowtie \pi_A T(t)).$$

To compute $V(NOW)$, ignoring (for clarity) all the permutations from standard query optimization, we break the defining expression into subexpressions and look for stored results that can be chosen as outsets for differential computations:

1) If $V(t)$, for some t, has been computed already, then at least one—possibly outdated—version of V exists, and becomes the chosen view. If several views containing $V(t)$ exist, again for arbitrary values of t, then the one

$$\left\{\begin{array}{c}\underline{\text{relation type}}\\ \boxed{\text{view}}\\ \boxed{\text{base relation}}\\ \text{backlog view}\end{array}\right\}\times\left\{\begin{array}{c}\underline{\text{time dependence}}\\ \boxed{\text{time dependent}}\\ \text{fixed}\end{array}\right\}\times\left\{\begin{array}{c}\underline{\text{redundancy}}\\ \text{not store result}\\ \left\{\begin{array}{c}\underline{\text{store result}}\\ \boxed{\text{index cache}}\\ \text{materialized data}\end{array}\right\}\end{array}\right\}\times\left\{\begin{array}{c}\underline{\text{update strategy}}\\ \text{eager}\\ \vdots\\ \text{threshold triggered}\\ \vdots\\ \boxed{\text{lazy}}\end{array}\right\}$$

Fig. 18. The boxed choices are considered.

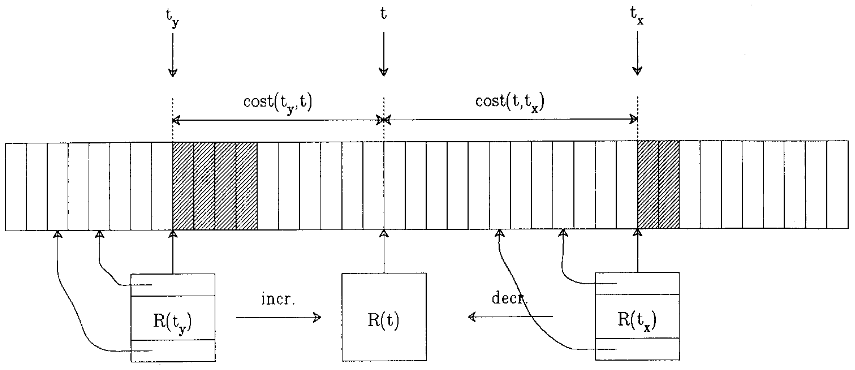


Fig. 19. Costs are estimated, compared, and a strategy is chosen.

with the lowest estimated cost is chosen. The chosen view is, then, used in a differential computation of $V(NOW)$.

2) If $V(t)$ has never been computed, time-slices containing $\sigma_{F_1}R(t)$ are located; one is chosen, and $\sigma_{F_1}R(NOW)$ is computed using the algorithms above.
3) If $V(t)$ had not already been computed, we now continue with the next subexpression of $V(t)$. If a view containing $\sigma_{F_2}S(t) \bowtie \pi_A T(t)$ has already been computed, the most promising is used in the differential computation of the subview. Otherwise, the two subexpressions are computed similarly to the computation involving R.
4) Finally, when subexpressions are differentially computed, they are combined in recomputations and the final result is achieved.

Many details of the computations of views are left unspecified in the description above. Specifically, easily computable and good cost estimation formulas for choosing between candidate views are subject to current research.

IV. Querying a Database

The time extension and the additional data structures require an extended query language. We use the standard relational algebra as a basis for such an extension. In subsection IV-A, we present and briefly discuss the operators of our data model. In subsection IV-B, we illustrate the utility of parts of the query language. Also, we describe how to evaluate queries in terms of the storage model and algorithm in the previous section.

A. Operators and Notation

A relation (e.g., *Emp*) is conceptually three-dimensional, the dimensions being attributes, tuples, and time. However, no operator can be applied to a relation before it is time-sliced, which eliminates the time dimension and results in a flat, two-dimensional time-slice. Therefore, the operators of the standard relational algebra need little or no change to work in our setting—the fundamental ones are presented in Fig. 20 along with time-slice. In the next subsection, we will also use some of the standard, derived operators such as join ($\bowtie$) and semi-join ($\ltimes$).

The special variable *NOW* has as value the current time, and it is useful when specifying conditions on transaction time attributes. Also, observe that tuples of base relations and schema relations have time stamps. These are hidden to obtain transparency, but they can be displayed by an explicit projection. In system generated relations, time stamp attributes are displayed, but they can, of course, be removed by a simple projection. Finally, in projections of backlogs, e.g., B_R, we will use the shorthand $\pi_{Tuple}B_R$ to mean a projection on all attributes of R.

The introduction of backlogs, accessible from the query language, helps make the extension *transparent*. It is a transparent extension of the standard relational model since any query of the latter model needs no syntactical modifications to work in the former and retrieves the data intended; updates have the similar property. Thus, a naïve user expecting only the standard relational model will be able to use the database. This is a major advantage in practical situations when switching

Fundamental Operators of the Query Language		
Notation	*Name*	*Description*
$R(t_x)$	Time-slice	This operator wasintroduced earlier. The expression t_x must evaluate to an element of domain *TTIME*. In the literature, the notation$\tau_{t=t_x}$ R is common. It is slightly more general than our function application notation because time interval can be easily expressed.
π	Projection	The standard projection operator. In the context of backlogs, we will use π_{Tuple} to mean projection on all attributes of the associated user-defined relation.
σ_F	Selection	The standard selection operator. The condition F can contain an arbitrary subquery.
$\times$	Cartesian product	The standard Cartesian product operator.
$-$	Difference	The standard difference operator.
$\cup$	Union	The standard union operator.

Fig. 20. Notation, names, and descriptions of fundamental operators.

from a standard model to a more sophisticated one. The simplicity is a major advantage of our design: The fundamental relational operators still work and the extension is transparent.

As we shall see next, the inclusion of backlogs accessible through the query language also opens to interesting queries related to the change history of the database. Before we do that, we include a *unit* operator (Υ), the aggregate formation operator of [21] (ξ), and aggregate functions min, count, and avg (as defined in standard SQL).

The unit operator is used for changing the units and precision of attribute domains. A detailed definition can be found in the Appendix. Consider the query

$$\Upsilon_{Time=Day}\sigma_{t_x \leq Time \leq t_y} B_S$$

In the result, the domain of attribute *Time* has been changed from *TTIME* to *Day*, and the values of attribute *Time* have been transformed into elements of domain *Day*.

The aggregate formation operator is used for applying aggregate functions to groups of attribute values. Here, we use the syntax of [1]:

$$\xi_{X,\ att\_name=agg\_fct}\ R$$

where X is a grouping specification, *att_name* is a new attribute name, and *agg_fct* is an aggregate function. The result of the query is derived in the following way: the tuples of R are divided into the groups implied by X, *agg_fct* is applied to each group, and the resultant value is associated with each tuple in the group as a value of the *att_name* attribute.

B. Sample Queries

We present a sequence of queries gradually getting more and more complicated. In doing this, we stress the role of differential evaluation. At first, queries on traditional base relations and views are presented. Then, we discuss how queries on backlogs are helpful in answering queries on change history.

Retrieve all employees as of close of business, May 1, 1990.

$$Emp(May\ 1,\ 1990\ 4{:}00\ p.m.)$$

This is an example of a fixed time-slice of a base relation. If no other queries have been issued on the relation *Emp*, this view is incrementally computed from Emp(t_{init}). The result stored in the database is a set of pointers to *Ins* tuples (from insertions and modifications) in B_{Emp} and a reference to B_{Emp} that indicates the time of validity of the time-slice.[1]

Retrieve all current employees.

$$Emp(NOW) \text{ or, alternatively: } Emp$$

This is a time dependent time-slice of a base relation (see Fig. 7). When this view is first computed, the previous fixed time-slice is utilized in an incremental computation. Later retrievals will utilize the immediate predecessor in an incremental computation. The differential file of this view consists of all the change requests to *Emp* that have arrived after it was last retrieved/computed.

Retrieve the employees that were in the company's database 20 days ago (as of now).

$$Emp(NOW - 20\ days)$$

This query, with *NOW* = May 1, 1990 4:00 p.m., was discussed in Fig. 7 and in the example in subsection II-E. It is very similar to the previous one, and a computation of this query will utilize the one of the previously computed and stored views from above with the lowest estimated cost.

Let us *define* a view as follows

$$Rich\_Emp(t) = \sigma_{Salary \geq 50000} Emp(t)$$

A definition does not result in any computation—all that happens is that the query expression itself is stored in the database system. The first step in evaluating any query is to time-slice the constituent relations. Therefore, to retrieve data from the view, an expression that evaluates to a value in the domain *TTIME* must be supplied and substituted for t. Then, the selection is computed.

Retrieve all very rich employees as of the close of business May 1, 1990.

$$\sigma_{Salary \geq 80000} Rich\_Emp(May\ 1,\ 1990\ 4{:}00\ p.m.)$$

This is a fixed time-slice that involves several levels of computation. First, *Rich_Emp* is substituted for its definition as in Ingres-style query modification. Second, the time-slice is computed. Third, the selection(s) is performed. Depending on whether the two selections are collapsed into one (the second) the computation of the query results in two or three separate index caches: the initial time-slice (the first selection), and the second selection.

Retrieve all very rich employees.

$$\sigma_{Salary \geq 80000} Rich\_Emp(NOW)$$

[1] The time of validity is the half-open interval that contains May 1, 1990 4:00 p.m. and is bounded by the two closest time stamps in B_{Emp}.

This query differs from the previous one in that it is time dependent. To compute the very rich employees, the *Rich_Emp*, in general, must be brought up to date first.

Let us now turn our attention toward queries directly involving backlog relations. Initially, let us look at how the usefulness of backlog relations supplement that of usual relations.

Let us suppose that we want **the changes to** *Emp* between t_x and t_y. Although not the only possibility, we can write this query (cf. [15]):

$$Emp(t_y) - Emp(t_x)$$

The result of this query is the tuples in *Emp* at t_y, not in *Emp* at t_x. This does not tell us what took place between t_x and t_y. For example, we will not retrieve any deletions that might have taken place in the time interval. If we really want to know what took place between t_x and t_y, we would be better off using the backlog of *Emp*. We have to make clear precisely what we want. Let us look at some possibilities.

$$\sigma_{t_x \leq Time \leq t_y} B_{Emp}$$

This query retrieves all possible information about what happened to *Emp*. Insertions, deletions, and modifications are distinguished, and the times when the requests were placed are available.

$$\pi_{Tuple} \sigma_{t_x \leq Time \leq t_y} B_{Emp}$$

This query eliminates the special backlog attributes from the result. Thus, several changes back and forth between identical *Emp* tuples will be eliminated, and it will not be possible to distinguish between operation types anymore.

$$\pi_{Tuple,\ Time} \sigma_{t_x \leq Time \leq t_y} B_{Emp}$$

The result of this query differs from the above in that time stamps are retained, potentially allowing more tuples to be retrieved. Still, it is impossible to distinguish a modification from a deletion which in many cases may be unfortunate.

$$\pi_{Tuple,\ Time} \sigma_{t_x \leq Time \leq t_y \wedge Op = Ins} B_{Emp}$$

Here we get the time stamped tuples that were inserted in the interval. The result is a list containing the employees hired in the interval.

$$\pi_{Tuple,\ Time} \sigma_{t_x \leq Time \leq t_y \wedge Op = Del} B_{Emp}$$

Here we retrieve the employees removed from the company's database.

$$\pi_{Tuple} \sigma_{t_x \leq Time \leq t_y \wedge (Op = Ins \vee Op = Mod)} B_{Emp}$$

Finally, we have retrieved all employees that had changes because they were either hired or their previous data were updated.

The possibilities listed above are by no means exhaustive, but are representative of the large number of easily formulated queries possible on backlogs. Let us now take a look at the evaluation of other kinds of queries.

To get **the time when the first employee was removed from the company database after April 30, 1990**, we use the aggregate formation operator and the function min.

$$\pi_{Min\_date}\ \xi_{-,Min\_date = \min(Time)}$$
$$\sigma_{Apr.30,90 \leq Time \wedge Op = Del} B_{Emp}$$

Because change requests are ordered according to time stamps, the system need not first retrieve all *Del* change requests inserted after April 30, and then, find the one with the smallest time stamp value; instead, the qualifying tuple can be retrieved directly.

Find all the employees at t_x that changed between t_a and t_b. This query and the following involve both a time-sliced base relation and a backlog.

$$Emp(t_x) \ltimes \pi_{Tuple} \sigma_{t_a \leq Time \leq t_b \wedge Op = Mod} B_{Emp}$$

The compliment is given by

$$Emp(t_x) - \pi_{Tuple} \sigma_{t_a \leq Time \leq t_b \wedge Op = Mod} B_{Emp}$$

Note that if expression t_x does not contain variable *NOW*, the results of these queries are fixed. Once computed, they never get outdated.

The following query results in **a list of name and time pairs of employees with changes on December 27, 1988, but with a time granularity of one hour**.

$$\Upsilon_{Time = Hour} \pi_{Name, Time}$$
$$\sigma_{Dec27,88 \leq Time \leq Dec.28,88 \wedge Op = Mod} B_{Emp}$$

If an employee changes salary more than once within the same hour (e.g., as a result of a typing error and a following correction), this will not be visible in the result because the unit operator rounds off to the closest hour.

Next, we want to retrieve **the employees that had their record changed about the average number of times during the last 2 years.** It can be done with the queries:

$$Q_1 = \pi_{Name, Count} \xi_{Name, Count = count(Time)}$$
$$\sigma_{Op = Mod \wedge NOW - 2\ yrs \leq Time} B_{Emp}$$
$$Q_2 = \pi_{Avg\_count}\ \xi_{-, Avg\_count = avg(Count)} Q_1$$
$$Q_3 = \pi_{Name} \sigma_{.8 * Q_2 \leq Count \leq 1.2 * Q_2} Q_1$$

In Q_1, we count for each employee the number of times the record was changed within the given 2 year period. The attributes of the view are named *Name* and *Count*. This is an example of a sliding time window on a backlog screening out irrelevant tuples and aggregating relevant tuples (review Fig. 13). Differential techniques are used to keep Q_1 up to date. In Q_2, we then find the average number of changes. In Q_3, we finally compare the number of changes of an employee's record to the average number of changes and keep employees that are close. Note that we use query Q_2 in the selection criteria of query Q_3. While this is nonstandard, it is also well-defined and convenient.

We retrieve **the employees with abnormal change pattern** by

$$\pi_{Name}\ \sigma_{Op = Mod \wedge NOW - 2\ yrs. \leq Time} B_{Emp}\ -\ Q_3$$

If we want only the employees with very few record changes during the last 2 years, we can issue this query, using Q_1 and Q_2:

$$\pi_{Name}\sigma_{Count \leq 0.2 * Q_2} Q_1$$

In summary, we have shown how to conveniently retrieve detailed information about the change history of relations. We have used differential computation of queries formulated in the operation language of our extended data model. In particular, we have demonstrated the convenience of the backlog relation.

V. Conclusion and Future Research

We have presented a relational data model extended with transaction time, and a differential implementation model for it.

The data model is a tuple time-stamping static rollback model—its advantages are twofold. First, it is very simple; only one new operator is fundamental for the model, the time-slice. In addition, we have included the nonstandard operators unit and aggregate formation, the special variable *NOW*, and transaction time valued expressions. The model is, also, a transparent extension of the standard relational model—in part, achieved by introducing backlogs. The accessibility of backlogs through the query language provides the second advantage—it is easy to retrieve detailed information about the change history.

The implementation model exploits techniques for eager/lazy update and differential computation in the context of materialized views stored as either index caches or data, and it is the first to integrate these techniques into a coherent whole.

Several topics touched upon in this paper are subjects for current research. Most prominently, we are investigating further the general framework discussed in subsection III-A. Sets of rules and cost estimation formulas for the decisions of the query evaluation subsystem are being developed. These include rules for when and how views should be stored; rules for how often a stored, time dependent result should be updated; and rules for which existing results should be used in differential computations.

Topics for future research include

- Extending the query language to support queries on change behavior. This includes investigation of facilities for detection of common patterns of behavior and deviations from these.
- The development of vacuuming subsystem that allows for the specification and actual removal of temporal data not needed by any application. While it deletes data, such a subsystem never changes the history as recorded in the database.

Appendix

Here we define the unit operator, Υ, used for changing units and precision of attribute domains. We focus on the precision aspect of the operator and choose to refer to [20] and [14] from where it follows that conversion between compatible units can be done algorithmically applying simple linear algebra techniques. Also, the issue of information loss due to finite arithmetic is beyond the scope of this presentation.

Units			
Domain$_1$	*Domain*$_2$	*Factor*	*Decimals*
D_{i_1}	$D^x_{i_1}$	k_{i_1}	c_{i_1}
D_{i_2}	$D^x_{i_2}$	k_{i_2}	c_{i_2}
$\cdots$			
D_{i_k}	$D^x_{i_k}$	k_{i_k}	c_{i_k}

Fig. 21. The relation *Units*.

The syntax of the operator is as follows:

$$\Upsilon_{A_{i_1}=D^x_{i_1}, A_{i_2}=D^x_{i_2}, \cdots, A_{i_k}=D^x_{i_k}} \cdot R(A_1 : D_1, A_2 : D_2, \cdots, A_n : D_n)$$

where $1 \leq i_j \leq n$, $j = 1, 2, \cdots, k$.

The result of this query is relation R with domain D_{i_1} of attribute A_{i_1} changed to $D^x_{i_1}$, domain D_{i_2} of attribute A_{i_2} changed to $D^x_{i_2}, \cdots$, domain D_{i_k} of attribute A_{i_k} changed to $D^x_{i_k}$.

A special relation, *Units*, contains information about which domains are compatible and how to make the transformations. It has four attributes: *Domain*$_1$ and *Domain*$_2$ are each character strings listing domain names. *Factor* is a real number, and *Decimals* is an integer. See Fig. 21.

The jth entry in the *Unit* relation tells that an element in domain D_{i_j} can be transformed into an element in domain $D^x_{i_j}$ by multiplying it with k_{i_j}, and when representing the element in scientific notation, allowing c_{i_j} decimals. The user can insert, delete, and modify tuples from this relation. In accordance with the normal convention, the operator rounds off to the closest value in a coarser domain.

In connection with domain *TTIME*, we have chosen the default unit to be minutes and the lowest unit to be seconds. Tuples, allowing for *Second*, *Minute*, *Hour*, *Day*, *Week*, *Month* and *Year*, have been inserted into *Units*. These units have zero decimals.

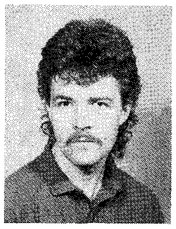
Acknowledgment

We wish to thank the editor Dr. S. Jajodia and the anonymous reviewers. Their detailed and knowledgeable suggestions for improvements greatly improved the quality of the presentation. Shortcomings—despite their efforts—are solely our responsibility.

References

[1] R. Agrawal, "Alpha: An extension of relational algebra to express a class of recursive queries," in *Proc. 3rd Int. Conf. Data Eng.*, Los Angeles, CA, Feb. 1987, pp. 1–11.
[2] M. E. Adiba and B. G. Lindsay, "Database snapshots," in *Proc. Sixth Int. Conf. Very Large Databases*, 1980, pp. 86–91.
[3] A. Bolour, T. L. Anderson, L. J. Dekeyser, and H. K. T. Wong, "The role of time in information processing: A survey," *ACM SIGMOD Rec.*, vol. 12, no. 3, pp. 27–50, Apr. 1982.
[4] J. A. Blakeley, N. Coburn, and P.-Å. Larson, "Updating derived relations: Detecting irrelevant and autonomously computable updates," *ACM Trans. Database Syst.*, vol. 14, no. 3, pp. 369–400, Sept. 1989.
[5] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa, "Efficiently updating materialized views," in *Proc. ACM SIGMOD '86*, May 1986, pp. 61–71.
[6] S. Cammarata, "Deferring updates in a relational data base system," in *Proc. Seventh Int. Conf. Very Large Databases*, 1981, pp. 286–292.

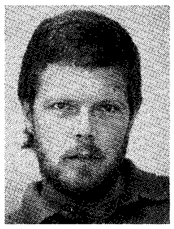
[7] E. F. Codd, "Extending the database relational model to capture more meaning," *ACM Trans. Database Syst.*, vol. 4, no. 4, pp. 397–434, Dec. 1979.
[8] J. Clifford and A. U. Tansel, "On an algebra for historical relational databases: Two views," in *Proc. ACM SIGMOD '85*, 1985, pp. 247–265.
[9] C. J. Date, *An Introduction to Database Systems—Vol. II*, The Systems Programming Series, first, corrected edition. Reading, MA: Addison-Wesley, July 1985.
[10] ——, *An Introduction to Database Systems—Vol. I*, The Systems Programming Series, fourth edition. Reading, MA: Addison-Wesley, 1986.
[11] U. Dayal and P. A. Bernstein, "On the updatability of relational views," in *Proc. Fourth Int. Conf. Very Large Data Bases*, 1978, pp. 368–377.
[12] P. Dadam, V. Lum, and H. D. Werner, "Integration of time versions into a relational database system," in *Proc. Tenth Int. Conf. Very Large Databases*, Aug. 1984, pp. 509–522.
[13] K. S. Gadia, "A homogeneous relational model and query languages for temporal databases," *ACM Trans. Database Syst.*, vol. 13, no. 4, pp. 418–448, Dec. 1988.
[14] N. Gehani, "Databases and unit of measure," *IEEE Trans. Software Eng.*, vol. SE-8, no. 6, pp. 605–611, Nov. 1982.
[15] S. K. Gadia and C.-S. Yeung, "A generalized model for a relational temporal database," in *Proc. ACM SIGMOD '88*, 1988, pp. 251–259.
[16] E. N. Hanson, "A performance analysis of view materialization strategies," in *Proc. ACM SIGMOD '87*, 1987, pp. 440–453.
[17] P. Hall, J. Owlett, and S. J. P. Todd, "Relations and entities," in *Modelling in Data Base Management Systems*, G. M. Nijssen, Ed. Amsterdam, The Netherlands, North-Holland, 1976, pp. 201–220.
[18] C. S. Jensen and L. Mark, "A framework for vacuuming temporal databases," Tech. Rep. CS-TR-2516, UMIACS-TR-90-105, Dep. Comput. Sci., Univ. of Maryland, College Park, MD 20742, Aug. 1990. Submitted for publication.
[19] K. A. Kimball, "The data system," M.S. thesis, Univ. of Pennsylvania, 1978.
[20] M. Karr and D. B. Loveman III, "Incorporation of units into programming languages," *Commun. ACM*, vol. 21, no. 5, pp. 385–391, May 1978.
[21] A. Klug, "Equivalence of relational algebra and relational calculus query languages having aggregate functions," *J. ACM*, vol. 29, no. 3, pp. 699–717, July 1982.
[22] V. Lum, R. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill, "Designing DBMS support for the temporal dimension," in *Proc. ACM SIGMOD '84*, June 1984, pp. 115–130.
[23] M. V. Mannino, "Statistical profile estimation in database systems," *ACM Comput. Surveys*, vol. 20, no. 3, pp. 191–221, Sept. 1988.
[24] L. E. McKenzie, "An algebraic language for query and update of temporal databases," Ph.D. dissertation, 88-050, Univ. of North Carolina at Chapel Hill, Dep. Comput. Sci., CB 3175, Sitterson Hall, Chapel Hill, NC 27599-3175, Oct. 19880.
[25] E. McKenzie and R. Snodgrass, "An evaluation of algebras incorporating time," Tech. Rep. TR-89-22, Dep. Comput. Sci., Univ. of Arizona, Tucson, AZ 85721, Sept. 1989. Conditionally accepted for *ACM Comput. Surveys*.
[26] A. B. O'Hare and A. P. Sheth, "The interpreted-compiled range of AI/DB systems," *ACM SIGMOD Rec.*, vol. 18, no. 1, pp. 32–42, Mar. 1989.
[27] N. Roussopoulos and H. Kang, "Principles and techniques in the design of ADMS±," *IEEE Comput. Mag.*, vol. 19, no. 12, pp. 19–25, Dec. 1986.
[28] N. Roussopoulos, "The logical access path schema of a database," *IEEE Trans. Software Eng.*, vol. 8, no. 6, pp. 563–573, Nov. 1982
[29] ——, "View indexing in relational databases," *ACM Trans. Database Syst.*, vol. 7, no. 2, pp. 258–290, June 1982.
[30] ——, "The incremental access method of view cache: Concept, algorithms, and cost analysis," Tech. Rep. UMIACS-TR-89-15, CS-TR-2193, Dep. Comput. Sci., Univ. of Maryland, College Park, MD 20742, Feb. 1989.
[31] D. Rotem and A. Segev, "Physical organization of temporal data," in *Proc. Third Int. Conf. Data Eng.*, Feb. 1987, pp. 547–553.
[32] R. Snodgrass and I. Ahn, "A taxonomy of time in databases," in *Proc. ACM SIGMOD '85*, 1985, pp. 236–246.
[33] ——, "Partitioned storage for temporal databases," *Inform. Syst.*, vol. 13, no. 4, pp. 369–391, 1988.
[34] P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proc. ACM SIGMOD '79*, 1979, pp. 82–93.
[35] A. Shoshani and K. Kawagoe, "Temporal data management," in *Proc. Twelfth Int. Conf. Very Large Data Bases*, Aug. 1986, pp. 79–88.
[36] B. J. Salzberg and D. Lomet, "Access methods for multiversion data," in *Proc. ACM SIGMOD '89*, June 1989, pp. 315–324.
[37] R. Snodgrass, "The temporal query language TQuel," *ACM Trans. Database Syst.*, vol. 12, no. 2, pp. 247–298, June 1987.
[38] R. B. Stam and R. Snodgrass, "A bibliography on temporal databases," *Data Eng.*, vol. 7, no. 4, pp. 53–61, Dec. 1988.
[39] F. W. Tompa and J. A. Blakeley, "Maintaining materialized views without accessing base data," *Inform. Syst.*, vol. 13, no. 4, pp. 393–406, 1988.
[40] J. D. Ullman, *Principles of Database Systems*, Computer Software Engineering Series, second edition. Rockville, MD: Computer Science, 1982.



Christian S. Jensen received the B.S. degree in mathematics and the M.S. degree in computer science from Aalborg University, Denmark, in 1985 and 1988, respectively.

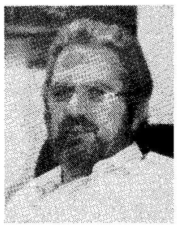
For 2 1/2 years, starting Fall 1988, he was part of the Database Systems Group in the Department of Computer Science at the University of Maryland, College Park. Recently, he received the Ph.D. degree and joined the faculty of the Department of Mathematics and Computer Science at Aalborg University as an Assistant Professor. Currently, he is a visiting faculty member at the University of Arizona. His research interests include the modeling of temporal data, query processing and optimization in temporal databases, version management, and object orientation.

Dr. Jensen is a member of the Association for Computing Machinery.



Leo Mark received the M.S. and Ph.D. degrees in computer science from Aarhus University, Denmark, in 1980 and 1985, respectively.

He was a graduate research assistant at the Department of Computer Science, University of Maryland, from 1983 to 1985, and he joined the faculty as an Assistant Professor in 1986. His research interests include data models, self-describing databases, metadata management, specification languages for operative rules, database generation from grammars, and temporal databases.



Nick Roussopoulos (M'91) received the B.S. degree in mathematics from the University of Athens, Athens, Greece, in 1969 and the M.S. and Ph.D. degrees in computer science from the University of Toronto, in 1973 and 1977, respectively.

He is an Associate Professor in the Department of Computer Science and the Institute of Advanced Computer Studies at the University of Maryland. He has published over 50 papers in refereed journals and conferences. His research interests include database systems, multidatabase management, engineering information systems, geographic information systems, expert database systems, and software engineering. He has worked as a Research Scientist at IBM Research, San Jose, and as a faculty member with the Department of Computer Science at the University of Texas, at Austin. Since 1981, he has been with the University of Maryland.

Dr. Roussopoulos served on the Space Science Board Committee on Data Management and Computation (CODMAC) from 1985 until 1988. He was the General Chairman of the ACM International Conference on Data Management 1986. He has also organized and chaired a series of workshops for the VHSIC Engineering Information System program. He also serves on the editorial board of two international journals, *Information Systems*, and *Decision Support Systems*.