Data & Knowledge Engineering 6 (1991) 1–25
North-Holland

Implementing version support for complex objects*

Henk BLANKEN
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. New applications in the area of office information systems, computer aided design and manufacturing make new demands upon database management systems. Among others highly structured objects and their history have to be represented and manipulated. The paper discusses some general problems concerning the access and storage of complex objects with their versions and the solutions developed within the AIM/II project. Queries related to versions are distinguished in ASOF queries (asking information valid at a certain moment) and WALK-THROUGH-TIME (WTT) queries (obtaining trend information concerning a certain period). In the paper some new algorithms to handle such queries are presented. A brief analysis gives an indication about the performance of query processing in historical databases.

Keywords: Complex object, Historical database, Query handing.

1. Introduction

Traditionally database management systems support applications like planning, accounting, personnel, and so on. Common to these applications is, among others, that the structure of the data is simple and that the current state of the described objects supports adequately the information needs. New applications arise, demanding extra or other facilities from the database management systems supporting them. In an office environment for instance, besides more 'traditional' objects like employees and orders, more complex objects like forms and documents play an important role. These object are highly structured (a document may consist of many chapters, sections and subsections). In many cases the most 'natural' way to describe such objects is to use a hierarchical (data) structure. Another characteristic of this environment is the importance of 'old' data: Tax forms of years ago, historical information concerning land registration, the 'evolution' of sales figures, and so on [1, 2, 4, 9, 20, 22].

Some analogous phenomena can be observed in the area of computer aided design and manufacturing (CAD/CAM), see also [8, 10, 11, 12, 15]. The relevant objects are complex artifacts like chips and cars that have to be designed and produced. To describe these objects more complex structures than are offered by the 'conventional' data models are, again, needed. When products (cars, computers) have been delivered one must be able, for instance for maintenance purposes, to trace very accurately what is included in the products shipped and what changes occurred to them. Hence again, historical data play an important part.

The subject of the paper is query processing in a database management system (DBMS) that has complete control over versioned complex objects. This DBMS is since 1982 under implementation within the Advanced Information Management/II project (AIM/II) at the IBM Heidelberg Scientific Center and offers such advanced features like integral support of

* Work done during one year visit to IBM Scientific Center, Heidelberg.

flat tables and hierarchies (extended NF^2 relation), integrated support of formatted and unformatted (textual) data, and integrated version support (see [5] and [16] for a more detailed description of the project's scope and status).

In 1984 there have been published two papers ([6] and [17]) describing how to integrate version support into a relational database system such that it can be supported in an efficient and storage saving way. Though – for sake of simplicity – described for flat tables, this scheme has been developed for supporting versions of complex tables (extended NF^2 tables) which allow to represent flat tables and hierarchies in a uniform way (see [18] and [19] for a description of the language interface).

It will be shown that the extension of this scheme to support so-called ASOF queries (which query the state of the database for a fixed point in time) is rather straightforward. Opposed to that the extension to support so-called WALK-THROUGH-TIME (WTT) queries (which query the states of the database for a whole time interval) for complex tables or a single complex object in this table is not that simple, especially not if one is concerned with efficiency. That is, update processing related to the current database state should suffer as little as possible while ASOF and WTT processing should be done such that the resulting response times are still acceptable. For this reason the WTT processing has not been implemented until now.

The rest of the paper is organized as follows: In Section two, complex objects and parts of it (subobjects) are considered at the user level. Some attention will be paid to language constructs with which queries concerning historical data can be formulated. The next Section considers the storage of versioned complex objects and topics like clustering and compacting are treated. Although in the AIM/II project some attention has been paid to schema changes [7], it is stressed that this paper will deal only with complex objects of a fixed type (see for the problem of schema evolution also [13]). So versions occur only within this context. In Section four the processing of ASOF queries for some typical examples is described. Section five indicates why a straightforward solution to the problem of handling WTT queries is not adequate and the next Section describes some alternative algorithms. Section seven gives a brief performance analysis of the handling of queries. Some typical cases are defined and an estimate of the performance of time related queries is given. The paper finishes with conclusions.

2. Versioned complex objects: a user view

2.1. Complex object type

In this paper an example will be used that concerns an enterprise that performs work on a contract basis. The company is divided into departments and each department is considered to be a complex object. In a department, projects are performed and for each project employees are hired, who work on a temporary basis. As employees are hired especially for a certain project, the longest hiring period is the project life time. In departments, equipment is available on behalf of the projects. The information concerning a department is very tightly related: When a department is deleted all information in this department (projects, employees) is deleted too. In *Fig. 1* a complex object type and in *Fig. 2* an occurrence are given. All projects, equipment and employees belonging to a certain department constitute a complex object of type DEPT. Besides the atomic attributes DNAME (department name) and ADDR (address) DEPT has two non-atomic attributes PROJ (project) and EQP (equipment). A value of the attribute PROJ is a set of project descriptions. A project forms together with its employees a subobject. The non-atomic

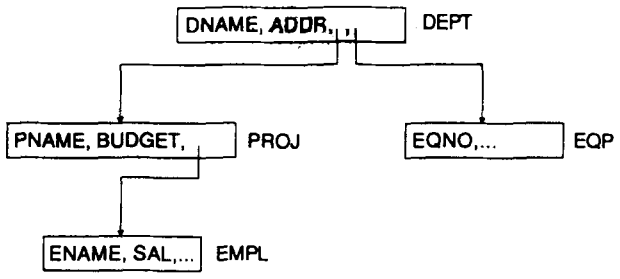

Fig. 1. Complex object type.

attribute EMPL represents a list of employees and the *list order* is determined by the 'period of experience' within the project (the catalog indicates whether EMPL is a set or list). An employee him(her)self is a subobject of the project to which he/she belongs. The semantics of the not mentioned attributes is supposed to be evident.

2.2. *Position changes of subobjects*

A complex object can be manipulated as a whole, but parts of it (subobjects) also: Insertion and deletion operations may have a subobject as an operand. It may also happen that a subobject is moved within a complex object. In this paper only moves within the same complex object are considered. These position moves have to taken into account when the history of complex objects has to be given:

- A project contains a list of employees and an employee may be allocated to another position within the list (when an employee leaves a project, list positions change).
- Assuming that equipment would be available at the department as well as at the project level, a complicated move operation would occur when equipment belonging to a project would be allocated to the department of that project.

2.3. *Querying complex objects*

In this paper queries concerning versioned complex objects are considered. The used language (called from now on Heidelberg Data Base Language HDBL) is described more extensively in [18, 19].

Consider the following query: "*Give current status of department COMPSC*". This can be formulated as:

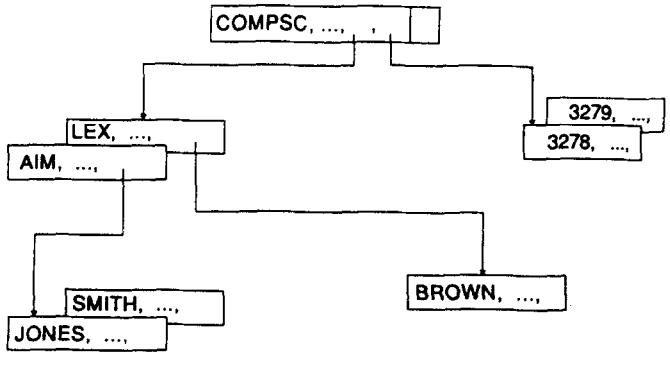

Fig. 2. Complex object occurrence ("COMPSC" department).

```
SELECT    x
FROM      x IN DEPT
WHERE     x.DNAME='COMPSC'
```

In this query x is a so-called element variable and iterates through all department occurrences. If the condition in the WHERE part is satisfied then the requested output as specified in the SELECT part is given. In this case the entire complex object is wanted.

The question arises, how to present a complex object to the user. The way chosen in this paper is inspired by the preorder walk through a tree. First, the atomic attributes of department COMPSC (see *Fig. 2*) are accessed and their values are presented to the user, then the same is done for the atomic attributes of the leftmost subobject (project AIM), followed by the leftmost subobject of this subobject (employee JONES), and so on in preorder. The result is pictured in *Fig. 3* and called the *preorder sequence* of the COMPSC object.

Given a certain complex object. As known, by applying transactions versions of objects are generated. When a transaction has been executed against this complex object a new, consistent version of the object has been obtained. Suppose now that the user does not want information about the current state of the department COMPSC, but instead about the version of January 1st, 1983, then the corresponding query gets an additional ASOF clause to indicate the wished version. Such a query will be called an ASOF query for short. The structure of the output is the same as described above.

```
SELECT    x
FROM      x IN DEPT ASOF 01-01-83
WHERE     x.DNAME='COMPSC'
```

A third query type concerns the evolution of an object in time. Instead of a moment in time the user specifies a certain time period: "*Give the states of department COMPSC during 1981–1984*". In HDBL this is achieved by using a "DURING ⟨FROM, TO⟩" clause where FROM and TO are two moments in time. Such a query is called a WALK-THROUGH-TIME query (WTT query for short). In HDBL:

```
SELECT    x
FROM      x in DEPT DURING ⟨01-01-81, 31-12-84⟩
WHERE     x.DNAME='COMPSC'
```

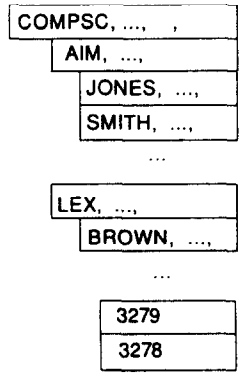

Fig. 3. PREORDER sequence of "COMPSC" department.

Again the representation question arises. One possibility is to present first the history of the COMPSC data tuple (the root tuple), then the history of the AIM data tuple (next tuple in preorder), and so on. That is, to present the history of an object as a sequence of tuple histories.

The user, however, usually does not deal with tuples but with objects. He/she may insert, delete or modify such an object as a whole. In this view the output has to show the evolution of the object in history and has therefore to start with the state of the object at moment FROM (01-01-81) followed by all succeeding states until time TO (31-12-84) is reached. Instead of a set of tuple histories the user gets as output a time-ordered sequence of object versions. Such a query is called a *forward query*.

One can also indicate that the output has to start with the object state at TO and has to go back in history until state FROM is reached. The user has to specify then the "BACKWARD" parameter in the SEQUENCE clause ("FORWARD" is default). Such a query is called a *backward query*.

```
SELECT      x
FROM        x in DEPT DURING ⟨01-01-81, 31-12-84⟩
WHERE       x.DNAME='COMPSC'
SEQUENCE    BACKWARD
```

3. Storage of versioned complex objects

It is generally accepted that in a historical database the current state of objects receives the majority of the accesses, see for instance [1, 12]. Therefore a common approach is to store current and historical data in separate areas. These areas are called from now on the current and the history pool. Moreover, to speed up the processing of the complex object as a whole, the data describing a complex object are stored as closely to each other as possible (this is called *clustering*).

3.1. Current state of object

Storage of complex objects has been treated in several papers [3, 5, 21, 23]. Although complex objects are built up of smaller parts, they are mostly handled as a whole. Another characteristic is that parts are added to or deleted from an object not only at the end but at arbitrary places.

The unit of storage is named a tuple (as in AIM/II a complex object is called an 'NF^2 tuple', a tuple is consequently called a 'subtuple'). To cope with the dynamic growing and shrinking of complex objects, the description of an object consists of two components, namely a structural and a data component. *Fig. 4* gives the storage structure of the object of *Fig. 2* and shows above the dotted line the structural and below it the data component. The collection of *atomic* values related to a subobject is described by a *data tuple*. For instance, the data tuple with the value 'COMPSC' contains all the values of atomic attributes of the department COMPSC. Besides data tuples so-called *control tuples* (in AIM/II terminology 'mini-directory subtuples') exist that contain pointers. In *Fig. 4* they are pictured above the dotted line. This figure shows a department control tuple containing three pointers ⟨DCC⟩. The D refers to the data tuple of the COMPSC department and each C refers to the value of a non-atomic attribute. Such a value is typically a set or list of subobjects of the same type. The first C refers to the projects and the second to the equipment of a department. The projects control tuple (describing the projects of COMPSC) contains a group of ⟨DC⟩ pairs

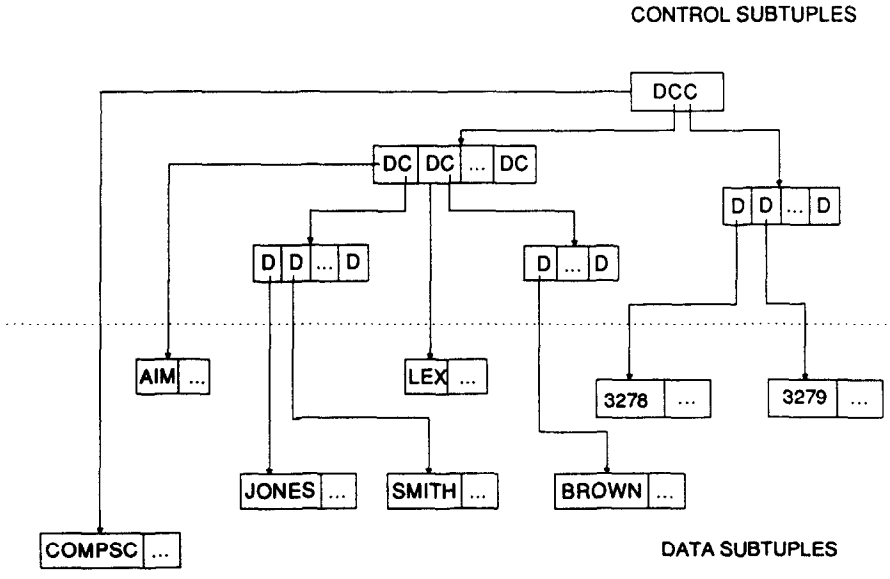

Fig. 4. Storage structure of "COMPSC" department (current object state).

each of them describing a project. The D-pointer, again, referring to the data tuple containing the atomic values of a project and the C-pointer referring to the employee control tuple of this project.

In the SYSTEM R literature so-called tuple-identifiers (TIDs) are introduced. Although in AIM/II a more complex pointer mechanism is chosen, see [5], we will assume for simplicity reasons that pointers are implemented as TIDs. As known, *TIDs remain constant during the life time of a database.*

Some characteristics of the above described storage structure for complex objects are:

- Objects are built up of tuples, which are stored separately. This flexibility eases maintaining clustering when insertions and deletions, which occur at all places in the object, have to be handled.
- It separates control and data tuples, which implies that control tuples can be allocated to as few pages as possible, which benefits the navigation through highly structured objects.
- Handling of queries concerning some aggregate functions (for example COUNT) can be done fast.
- Lists can be implemented easily, namely by placing the pointers referring to the list elements in the right order.

3.2. Versions of objects

Historical databases can become huge, hence one has to keep memory occupation within acceptable bounds. Some observations can be used to design an efficient storage scheme:

1. Two successive versions of an object often share many tuples. So it would be a waste of storage space to store all those object versions totally.

2. Two successive tuple versions may share many attribute values. This fact can be exploited too.

3. Some attribute values are very long (think of megabytes). Two versions may share many bytes and be different in only a few bytes. This observation can be used to store only one complete and many compacted versions of such an attribute.

4. The history of not (every part of) every complex object is interesting to the user. One may be interested in the salary history of an employee, but not in the history of his address.

In this section a structure is described that allows the system to take advantage of the first observation. In the next section the condensed storage of tuple versions (points 2 and 3 above) are considered, while in Section 3.4 versioning of only selected attributes is treated.

In order to exploit the first observation the unit of versioning must be a tuple. Suppose a transaction generates a new object version. Tuples that did not change must not get a new version! (The number of tuples of a complex object that did change depends, of course on the transaction, and the application.) The current state of a tuple is stored in the current pool and versions of a tuple are stored in the history pool. The current tuple and the versions are chained, see *Fig. 5*. Each new tuple (either data or control tuple) will get a timestamp TS equal to the moment of insertion. A modification causes the old contents (with 'old' timestamp) to be inserted in the history pool, while the new contents (with 'new' timestamp) overwrites the tuple in the current pool. A deletion is treated as a special kind of modification: the current tuple gets a special deletion code indicated by 'DEL' (the reason for storing deleted tuples in the current pool is given in the next paragraph).

In *Fig. 5* a strongly simplified example of a versioned complex object (the department 'COMPSC') is given. No schema changes are considered, so the department control tuple (the tuple in the left upper corner with timestamp 07-07-81) has no history chain. It is pictured that JONES has left the company at 01-07-83. SMITH joined the department COMPSC at 05-12-83. Both facts are reflected in the history chain of AIM's employees control tuple. Notice that *C and D pointers in the history pool always point to tuples in the current pool.*

Different sorts of time are considered (see among others [14]), namely the period during which a fact is valid (sometimes called the *valid time*), the moment at which this is noticed

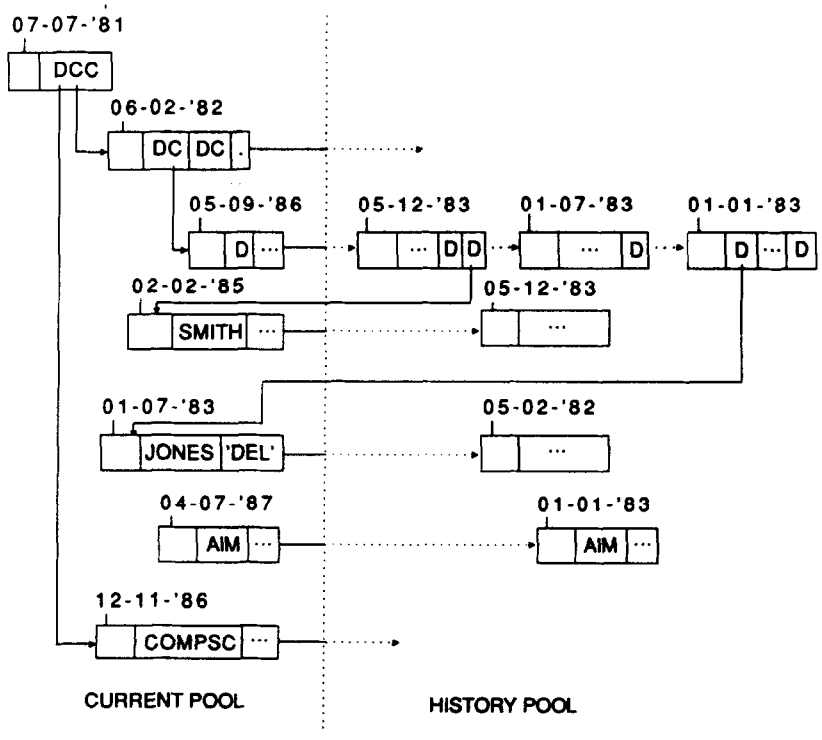

Fig. 5. JONES leaves company at 01-07-'83 and SMITH joins COMPSC at 05-12-'83.

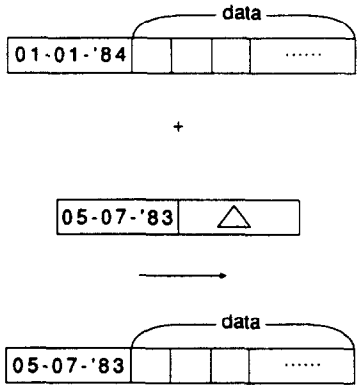

Fig. 6. Reconstruction of an older tuple version.

(*observation time*) and the moment at which a fact is recorded in the computer system (*registration time*). In AIM/II the system takes care of the latter time.

3.3. Storage of tuple versions

3.3.1 Compacting

Tuple versions have to be stored as compactly as possible giving rise to so-called delta versions. Such a delta version contains besides a timestamp the difference between a tuple version and its immediate successor [6]. In AIM/II several kinds of compacting techniques are used but it is outside the scope of this paper to go into details. In order to interpret a delta version (i.e. to reconstruct an older state of a tuple) all versions on the path from current tuple to this delta are needed. *Fig. 6* shows the generation of tuple version with help of the younger version and a delta.

3.3.2 Clustering

In the history pool tuple versions can be stored in two ways. First, one can cluster versions that belong to the same tuple thereby speeding up the reading of the history chain of a certain tuple. This way of access is called 'record-oriented' access in [12].

Second, one can cluster versions with respect to a complex object. This implies that all changes generated by a transaction for one complex object are stored together. Of course, this clustering strongly benefits the processing of object versions (this is called 'version-oriented' access in [12]).

3.4. Selective versioning: a problem

One is often interested in the history of only certain parts of complex objects. This is called selective versioning. Related to selective versioning is the following problem. Suppose that in our example versioning of *only* the employee data is required. Thus the DBMS must be able to answer questions concerning for instance the salary history of employees: "*Which employees, working at 01-01-83 for our company, earned 16 at that moment*"? When a DBMS only takes versions of the EMPL data tuples, then it is impossible to reach the tuples of employees who left the company.

A solution could be to scan the current pool to find the deleted data tuples and try to select those tuples that are of type EMPL. Of those tuples the history has to be scanned to contribute to the answer. As the current pool is normally rather voluminous, this may be very time consuming. Another solution is to keep also the history of the control tuples

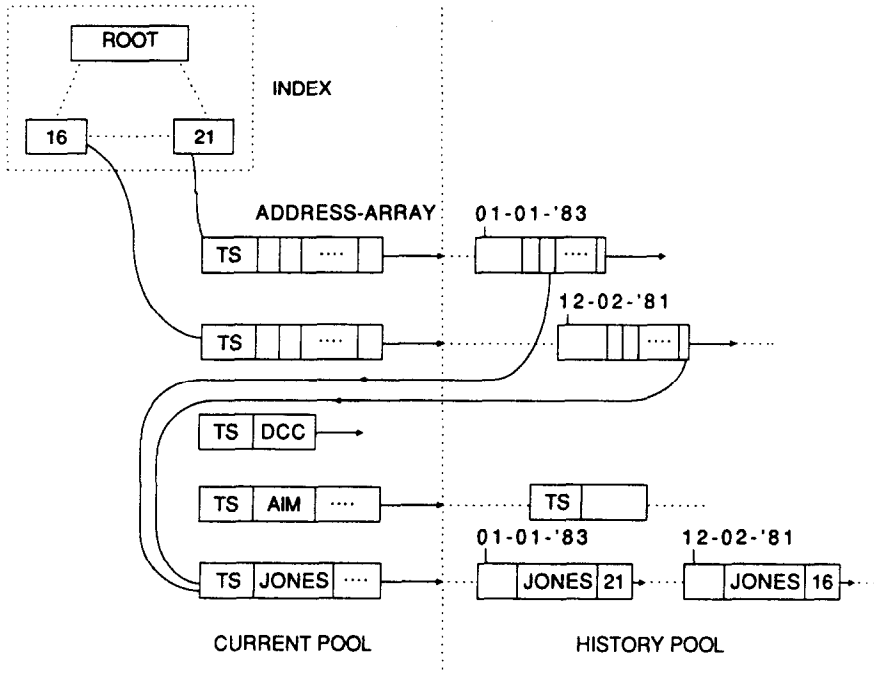

Fig. 7. SAL index for employee subtuples.

describing the employees. But for these control tuples holds the same: When an employee leaves the company and a moment later his/her project is cancelled, then the history of the employee can only be obtained when the history of the control tuple describing the projects of a department has been kept too! Hence: Versioning of the EMPL data tuple implies versioning of all control tuples on the path to the root control tuple.

In AIM/II selective versioning is supported. In the sequel, however, it is assumed that for all attributes versioning has been specified.

3.5. Indexes

In AIM/II indexes are offered to support query handling [5, 17]. An index tree contains leaves with ⟨attribute value, pointer⟩ pairs, where the pointer refers to the current version of an address-array. Such an address-array can be considered to be a tuple and has a current state and history versions. Each version of the address-array has a timestamp TS and points to data tuples, that had at time TS the concerned attribute value. In *Fig. 7* a chain of address-arrays is shown for the salary values 16 and 21. It appears that JONES got a salary 16 at 12-02-81 and of 21 at 01-01-83. For clearness' sake, real values are pictured in the history pool. See for a rather extensive discussion of these concepts [5].

4. ASOF query processing

In this section two queries will be treated. The first concerns the salary history of an employee. Such type of query is perhaps typical for the 'normal commercial' environment. The second query treats the history of a complex object. This kind of query is characteristic for CAD/CAM applications. Given the storage structure of the previous sections the processing of queries is demonstrated.

4.1. ASOF for an attribute

Consider the query: "*Take the* 01-01-83 *version of the database and retrieve from that database all information about those employees earning* 21 *at that time*". In HDBL:

```
SELECT    x
FROM      x IN y.EMPL
          y in z.PROJ
          z IN DEPT ASOF 01-01-83
WHERE     x.SAL=21
```

In this query three element variables, namely x, y and z occur. The latter iterates through the departments, the second through the projects within a department and x iterates through the employees within a project within a department. The steps needed to obtain the result can be characterized as follows (we neglect actions like parsing and optimizing queries):

```
Procedure ASOF for an attribute
        if SAL index on EMPL
        then  read 01-01-83 version of pointer-array of SAL = 21;
              foreach D pointer do
                      read 01-01-83 version of EMPL tuple;
                      print this version
              end
        else
              foreach DEPT object (also deleted ones) in current pool do
                      read 01-01-83 version of DEPT object;
              foreach EMPL do
                      if SAL = 21 then print EMPL version
                      end
              end;
end of ASOF for an attribute;
```

If no index is present all 01-01-83 versions of DEPT objects that existed at 01-01-83 have to be fetched. In the next section it will be shown how this can be done. It appears to be a very time consuming process. So this example illustrates the use of indexes with respect to query processing.

4.2. ASOF for a complex object

"Give current version of department COMPSC". In HDBL:

```
SELECT    x
FROM      x IN DEPT ASOF CURRENT
WHERE     x.DNAME='COMPSC'
```

In section two the layout of the required output has been defined. At moment CURRENT, the state of an object is uniquely defined and hence the preorder sequence. So the problem is to reconstruct the state of the object using the AIM/II storage structures, see *Fig. 4*. First the COMPSC, then the AIM, JONES, SMITH, . . . data tuples will be read. More formally:

```
Procedure ASOF for complex object;
        read control tuple of department COMPSC;
                (* contains ⟨DCC⟩ triple *)
        read COMPSC data tuple;
                (* first C: projects control tuple containing ⟨DC⟩ per project *)
        foreach project do
                read project data tuple;
                        (* C: employees control tuple containing a D per
                        employee *)
                foreach employee do
                        read employee data tuple
                end
        end;
                (* second C: equipment control tuple containing a D per
                equipment *)
        foreach equipment do
                read equipment data tuple
        end;
end of ASOF for a complex object;
```

"*Give version of department COMPSC at 01-01-83*". In HDBL:

```
SELECT    x
FROM      x IN DEPT ASOF 01-01-83
WHERE     x.DNAME='COMPSC'
```

Also at 01-01-83 the state of the department COMPSC is uniquely defined and hence the preorder sequence. The problem is to reconstruct the state of the COMPSC department at 01-01-83 using the structure pictured in *Fig. 5*. We start with the root control tuple. As no schema changes are considered, no history chain is present for this tuple and the 01-01-83 version is equal to the current version. Having this starting point, the 01-01-83 version of the object can be fetched. In essence the above mentioned algorithm can be used; the only difference is that every 'read tuple' has to be replaced by a 'read 01-01-83 version of tuple'.

5. WTT query processing – the problem

A first approach to WTT processing could be based on the ASOF query processing. For ASOF queries a moment T is specified that uniquely defines an object state. The required output can be derived from this state. When we know all moments at which a new state of the object has been generated, a WTT query could be handled by generating an ASOF query for each moment of change. In order to keep the moments at which changes occur, update processing can be extended to include so-called update signalling.

5.1. Update signalling

Update signalling implies that each update of a data tuple causes a 'dummy update' in the control tuples laying on the path from this data tuple to the root control tuple. Such a 'dummy update' states that the control tuple itself did not change, but that a change occurred in one of the 'dependent' data tuples. Hence, every change to a data tuple causes among

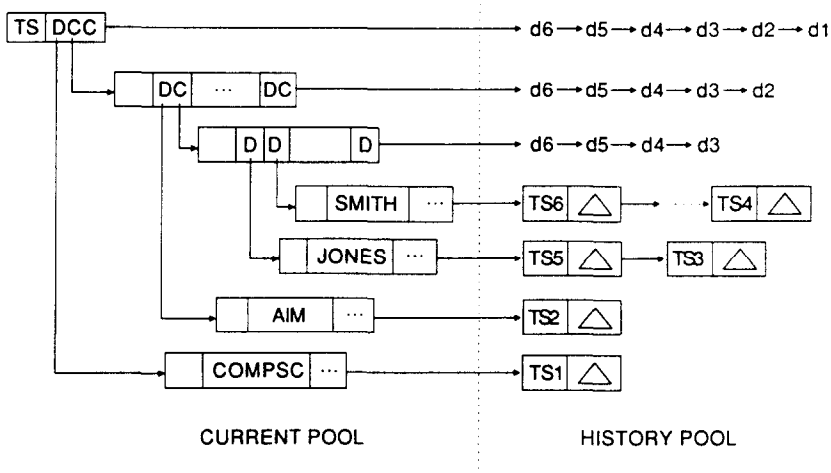

Fig. 8. Update signalling for the COMPSC department (△ = difference; di = dummy delta version for update at TSi).

others a dummy update of the root control tuple. By inspecting the history of this root control tuple, one knows at which times a change of the complex object occurred. In *Fig. 8* the data tuple COMPSC got a new version at TS1. This generated a dummy update d1 of the root control tuple. Later on AIM got a new version generating d2 for the root and projects control tuple. We see that the chain of dummies at the higher levels can become quite long.

5.2. Evaluation

- Update signalling requires additional external storage as the dummy updates occupy space. The needed space per dummy update is small, but in general many dummy updates are required.
- The update process itself is slowed down significantly. Instead of only the data tuple, also some control tuples have to be updated. This drawback is very serious.
- ASOF processing is also slowed down as history chains become longer.
- WTT processing is conceptually simple (a sequence of ASOF queries) but the performance will be bad. Roughly speaking: When n updates occurred in the interval ⟨FROM, TO⟩ then WTT processing is about n times as slow as ASOF processing.

This alternative has been rejected as too many disadvantages are related to it. Although variations on update signalling are possible, all seem to have major drawbacks, hence other strategies have to be developed.

6. WTT algorithms

To avoid slowing down the normal update process the two algorithms described below do not require extra read/write operations for bookkeeping. All information needed to handle WTT queries is stored in the history pool; the only problem is to obtain it efficiently. Both algorithms *read a history chain only once*, which is an important property as history chains may become long.

6.1. HISTCH

HISTCH comes from 'history chain at a time' and processes first the versions of one tuple, then the versions of another tuple, and so on. We only treat a backward query (forward queries are handled similar):
"*Give the history of department COMPSC from* 31-12-84 *until* 01-01-81". In HDBL:

```
SELECT      x
FROM        x IN DEPT DURING ⟨01-01-81, 31-12-84⟩
WHERE       x.DNAME='COMPSC'
SEQUENCE    BACKWARD
```

The output must start with the 31-12-84 version of department COMPSC, followed by older versions until the version of 01-01-81 has been reached. *So the output is a time-ordered sequence of object states.*

The algorithm will be explained in two steps. The first step only considers updates that cause no change in the preorder sequence of an object. That means that only modifications of data tuples are taken into account. In the second step also insertions, deletions and position moves are examined. An example of a position move is an employee leaving a certain project and joining at the same time another one. (It is clear that insertions, deletions and position moves influence the preorder sequence of an object and hence the output.)

6.1.1 Modifications only

As HISTCH processes a history chain at a time, after reading the history of the relevant tuples a set of history chains results, that is a set of time-ordered sequences of tuple versions. The output, however, has to be delivered as a time-ordered sequence of object states, each state being an (ordered) set of tuple versions. So, the ordering that remains after processing the history chains and the ordering required within the output do not match. Hence a sort operation is necessary. In the sort criterion the position of a tuple in the complex object plays a part. This position is defined by a so-called *preorder number* PN. As no schema changes occur, to each tuple type a unique type number $T\#$ can be allocated. Again the preorder sequence is used, implying for *Fig. 1* that DEPT, PROJ, EMPL and EQP get $T\# = 1$, $T\# = 2$, $T\# = 3$ and $T\# = 4$ respectively. Moreover, a (sub)object may 'contain' of a set (or list) of smaller subobjects. For instance a department object consists of a set of projects. For simplicity reasons a set is treated in this paper as a list, so to each project a position number $P\#$ can be allocated being the number in the list. Hence to a project and its related data tuple, a pair $\langle T\#, P\# \rangle$ corresponds. The preorder number PN of a project data tuple can now be constructed by concatenating the pair $\langle T\#, P\# \rangle$ of the department data tuple with the pair $\langle T\#, P\# \rangle$ of the project data tuple. This can be generalized to smaller subobjects, see *Fig. 9* for an example: 1,1 2,2 3,1 identifies the data tuple of employee SMITH via the path "COMPSC, LEX, SMITH".

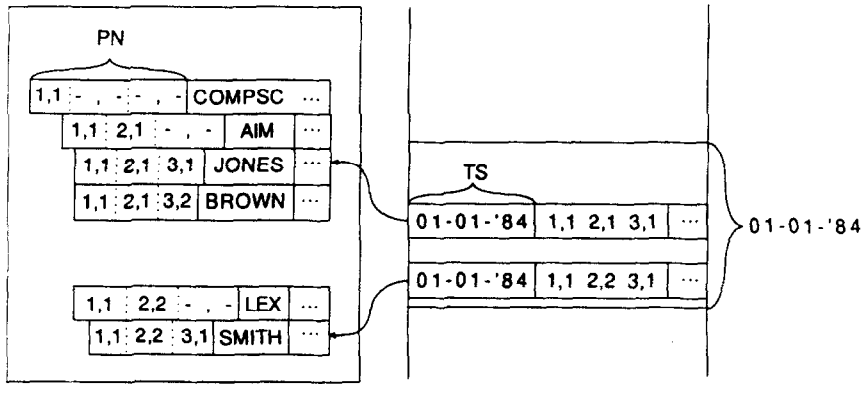

Fig. 9. HISTCH: Creation of version 01-01-'84 (backward query).

In this paragraph we only consider modifications of data tuples, so control tuples do not change. Using the control tuples we can obtain D pointers which refer to data tuples. In the following pseudo-code we take the D pointers for granted. Note that OBJVERS stands for "object version":

```
Procedure HISTCH:
        foreach data tuple do
                read version of 31-12-84;
                write ⟨31-12-84, PN, data⟩ to OBJVERS;
                read next version of this data tuple;
                while timestamp TS > 01-01-81 do
                        write ⟨TS, PN, data⟩ to CHANGES;
                        read next version
                end
        end;
                (* OBJVERS contains tuples of object at 31-12-84 *)
        sort OBJVERS on PN (ascending);
                (* CHANGES contains set of sequences of tuple versions *)
        sort CHANGES on TS (descending) and PN (ascending);
        foreach new object version do
                merge OBJVERS and relevant changes in CHANGES;
                print object version
        end;
end of HISTCH;
```

In *Fig. 9* OBJVERS contains the object version at 01-04-84. In CHANGES the updates are sorted on TS, so all updates of 01-01-84 are stored near each other. Now it is easy to generate version 01-01-84 of the object and store it in OBJVERS.

6.1.2 Changes in preorder sequence

A WTT query considers a certain period (indicated by the DURING clause). In this period the preorder sequence of a subobject may change caused by insertions, deletions or position moves of subobjects. *Fig. 10* shows that at time 01-07-83 employee JONES left

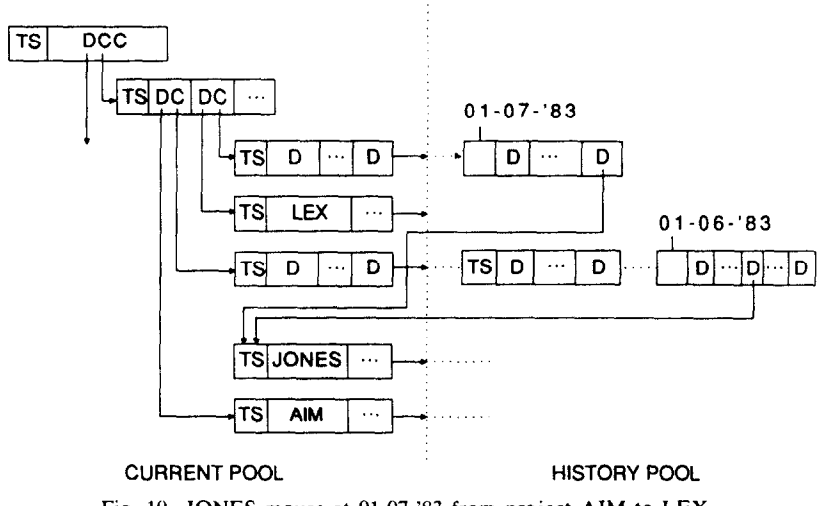

Fig. 10. JONES moves at 01-07-'83 from project AIM to LEX.

project AIM and joined project LEX. This causes a change of the position of JONES in the preorder sequence of the department COMPSC.

Suppose that HISTCH first processes the history of data tuples of employees working for AIM (including that of JONES) and later on the history of employees for LEX (again: including that of JONES). Then the history of JONES' tuples will be *processed twice* and that is not the intention. (Matters are still more complicated if not a 'simple' subobject like an employee, but a more complex one is moved from one position in a complex object to another.)

A control tuple describes a set or list of subobjects of the *same type*. The solution to the above mentioned problem is to scan the history of a control tuple *before* the history of data tuples that are referred to from inside the control tuple. When processing the history of a control tuple, a ⟨TS, PN, D⟩ triple is generated for every 'preorder change' that occurred during the considered time period. For a D pointer first all triples are collected and then the history chain of the data tuple pointed to by the D pointer. It is possible now to reconstruct exactly what happened.

6.2. *WTT algorithm: OBJVTM*

Instead of processing one history chain at a time, OBJVTM processes the tuple versions in an 'object version at a time' way. Suppose that the object version at moment T is given. In order to generate the previous version, the necessary tuple versions are fetched from the history pool. So for a backward query, first the current state of the object is determined, then the state just prior to the current one, and so on.

For this processing two areas are needed. Both areas describe the object at time T and contain *one entry for each control and one for each data tuple*. As insertions and deletions occur, the number of entries in the two areas varies in time.

The first area is named OBJVERS and an entry of this area contains the version of a control or data tuple at time T. The other area, called CLICK, is used to determine those tuple versions of the history pool that contribute to the object version just previous to T. An entry contains four fields (see *Fig. 11*):

- a D/C pointer referring to the current version of a data tuple, if the entry describes a data tuple and to a control tuple otherwise,
- a timestamp TS of the version of the tuple just previous to T,
- a pointer PREVPTR that refers to that tuple version, and
- a pointer OBJPTR referring to the version of the tuple in the area OBJVERS.

In the pseudo-code description of OBJVTM the role of these fields become clear. Notice that CLICK is a kind of translation table: Given a D (or C) pointer a reference (namely OBJPTR) can be obtained to the tuple in OBJVERS.

```
Procedure OBJVTM
        (* initialize OBJVERS*)
    move current version of control and data tuples to OBJVERS;
    T := CURRENT;
        (* initialize CLICK *)
    foreach entry do
        set D/C to TID of tuple;
        set OBJPTR to address of tuple in OBJVERS;
        set TS to TS of youngest tuple version;
        set PREVPTR to TID of youngest tuple version;
    end;
```

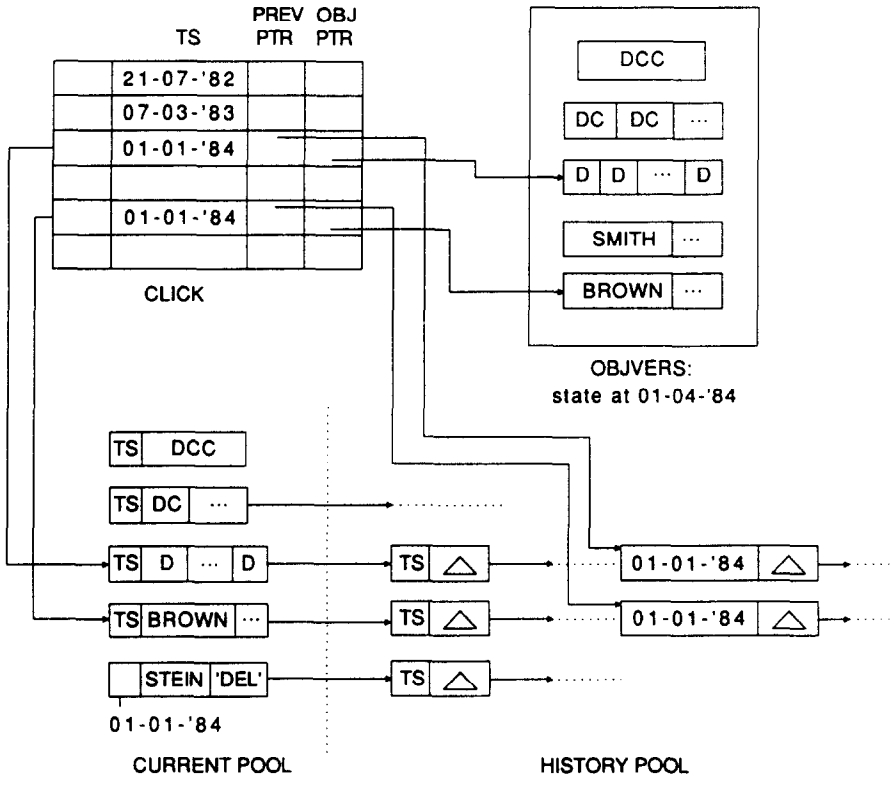

Fig. 11. OBJVTM: Creation of object vesion at 01-01-'84 (backward query).

```
while   T > 01-01-81 do
        determine entries in CLICK with TS-prev = max {TS | TS < T};
        fetch tuple versions with TS-prev (using PREVPTRs);
        use these versions to update OBJVERS (using OBJPTRs);
                (* OBJVERS contains now object version TS-prev *)
        print OBJVERS in preorder (* using CLICK as 'translation table' *);
        T := TS-prev;
        update the fields PREVPTR and TS in used CLICK entries;
    end;
end of OBJVTM;
```

In *Fig. 11* OBJVERS contains the version of 01-04-84. The nearest time click (=TS-prev) is 01-01-84. At that time BROWN's data tuple has been modified. OBJVERS has to be updated to reflect the state before 01-01-84. The history chain of Brown is scanned one version backwards to determine the previous change to Brown's data tuple. With this information TS and PREVPTR of Brown's entry in the CLICK table are updated. The current pool shows that STEIN left the company at 01-01-84 which caused at that time a change in the employees control tuple. In order to reflect this, OBJVTM has to adapt OBJVERS: the employees control tuple has to indicate that STEIN worked for the company before 01-01-84. Moreover, STEIN's data tuple has to be inserted in OBJVERS and STEIN must get a properly initialized entry in the CLICK table.

The preorder is defined by the control tuples that are stored in OBJVERS. Note, that in the control tuples of OBJVERS D (and C) pointers are stored and that the CLICK table has

to be used to 'translate' the D (and C) pointers to OBJPTRs which refer to tuples within OBJVERS. Position moves do not cause any problems in OBJVTM as the control tuples in OBJVERS reflect at each moment the precise state of the complex object.

A *forward query* can be handled by (starting with the current version) generating the relevant object versions while writing every complete object version to a file. When FROM has been reached the file can be read backwards while issueing every version to the user; this process stops when version TO has been shown. To improve efficiency the approach followed in this paper is to write to the file only *differing tuples* instead of *complete* object versions. After reaching the FROM state, this file called EXPANDED will be read backwards and used to reconstruct the successive object versions between FROM and TO.

7. Brief performance analysis

Some database researchers state that query handling in historical databases is not feasible as response times will often be unacceptable. The purpose of this chapter is to give an impression of the performance of query handling. The criterion is the number of page accesses needed to process a query. As known this quantity is already important in conventional databases; it is expected that historical databases bear still a 'higher I/O burden' than conventional ones.

In Section 7.1 the assumptions are given on which the formulas are based (the formulas themselves are given in the appendix). Two ASOF queries are considered, the first one requests the value of an attribute in the past. The second query retrieves the current version of a complex object, a query that has to be processed probably very often in all kinds of environments. Section 7.3 is devoted to the main part of this chapter, namely the processing of forward and backward WTT queries. For several cases the performance of the described WTT algorithms will be analysed.

7.1. Assumptions and notations

The most important assumptions underlying the used formulas are:

1. The current pool is clustered on complex object and each object is stored in preorder.
2. Only two forms of clustering the history pool are considered, namely clustering on object version and on history chain. The latter means that all versions belonging to the same tuple are stored consecutively.
3. On a page of the history pool only versions of the same tuple (if clustering is on chain) or of the same object (if clustering is on object version) are stored.
4. To make the evaluation more 'system independent' no specific operating system characteristics concerning size and management of virtual memory are regarded. Instead it is assumed that a buffer is available in internal memory ('real core').
5. Processing of control tuples is neglected. Moreover, only modifications of data tuples are considered (so no insertions or deletions).

The last assumption deserves some extra attention. Neglecting the control tuples is justified by the fact that the number of control tuples is much smaller then the number of data tuples (a factor of 10 to 50 may be a reasonable estimate). Hence processing control tuples, although logically interesting, is from a performance point of view not relevant. (Considering only modifications means that the control tuples have an empty history chain.)

Besides assumptions also a couple of abbreviations are used, see *Table 1*. The default values for these parameters are given between brackets, for example NTUP (giving the number of data tuples in a complex object) has as default 500, and the internal buffer length

Table 1
Used abbreviations

NTUP	: Number of data TUPles in object (500).
TUPLEN	: (Data) TUPle LENght in bytes (300).
DELTLEN	: DELTa version LENgth in bytes (40).
CHNGLEN	: CHaNGes record LENgth in bytes.
NCHTUPLS	: Number of CHanged TUPLeS between two successive object versions (8).
NTUPVERTF	: Number of TUPle VERsions between TO and FROM (2).
NTUPVERCT	: Number of TUPle VERsions between CURRENT and TO (2).
NOBVERTF	: Number of OBject VERsions between TO and FROM ((NTUP * NTUPVERTF) / NCHTUPLS).
NOBVERCT	: Number of OBject VERsions between CURRENT and TO. Is equal to (NTUPVERCT/NTUPVERTF) * NOBVERTF.
PAGLEN	: PAGe LENgth in bytes (4000).
IBLEN	: Internal Buffer LENgth in bytes (100.000).
NPGACC	: Number of PaGe ACCesses needed.

IBLEN has been set to 100 Kbytes. A parameter name ending on CT gives information about the period CURRENT to TO, and TF about TO to FROM. For example, the parameter "NOBVERTF" gives the number of object versions that has been generated in the past between TO and FROM. As stated before, a transaction transforms the version of a complex object into its next one. The number of tuples that has been changed after the transformation is called NCHTUPLS. Depending on the application this number can be high or low (default is 8, see *Table 1*).

7.2. *ASOF queries*

7.2.1 *ASOF query for an attribute (Table 2)*

Consider the query: "*Give salary of employee JONES at* 01-01-83". In the processing two parts can be distinguished: Fetching the most recent version of a data tuple and processing the history. The second part is sensitive to the length of the interesting part of the history chain.

In the history pool, two ways of clustering are possible (assumption 2). Consider the case that clustering is on chain. Compacting causes that versions are kept as small as possible (length equals DELTLEN bytes). *Table 2* gives an impression of the number of needed page accesses. As the considered query seems to be typical in the conventional data base environment where many short transactions are executed against the database, the average number of interesting versions for a data tuple, called NTUPVERCT, is taken high: 100, 1000 and 10000. Although the computation is very rough it indicates that in many situations history processing is feasible.

The second case (clustering is on version) implies that each tuple version requires one page access as it is assumed that a page may contain tuple versions belonging to one complex object version only (assumption 3). This case is so simple that it will not further be discussed.

Table 2
ASOF query: number of page accesses needed to process attribute history (clustering = chain)

	30	100	300	DELTLEN
NTUPVERCT				
100	1	3	8	
1000	8	25	75	
10000	75	250	750	

7.2.2 Processing the current state of a complex object

It is generally agreed that also in historical databases the current version of an object has to be fetched very often. In 7.1 it has been stated that the current version is stored in preorder; this assumption minimizes the I/O load. The number of page accesses needed to present the output to the user is given by NPACUOB (see appendix) and typical values may be between 1 and 500.

7.3. WTT queries

In this section WTT queries that give a picture of the evolution of an object during the past are considered. The performance of the two algorithms to process WTT queries is evaluated. It is clear beforehand that clustering on chain strongly benefits HISTCH, while clustering on version does the same for OBJVTM. These and other figures are given below.

7.3.1 HISTCH and OBJVTM (Table 3)

First we concentrate on *backward queries*. HISTCH reads history chains, sorts the CHANGES file and applies a merge operation for each version of the object that has to be generated. OBJVTM generates all versions between CURRENT and FROM.

For small hierarchical objects that fit in internal buffer (small NTUP values), reading of the history appears to be the dominant factor for both algorithms among others because of the fact that the generation of older object versions can be done efficiently. Hence clustering of the history pool determines more or less which of the two algorithms performs the best. When clustering is on chain then HISTCH performs better and in the other case OBJVTM beats HISTCH.

When the hierarchical object is too big to fit in internal buffer, then it must (partly) be stored on disk and the generation of (older) versions requires many extra I/O's. In this case OBJVTM needs more time to reach the TO object version: Both algorithms have to read the history chains and to generate the object versions between TO and FROM, but OBJVTM also has to generate the object versions between CURRENT and TO. This explains the fact that for higher NTUP values HISTCH performs relatively better.

7.3.2 Forward versus backward queries (OBJVTM) (Table 4)

For HISTCH the performance difference between the processing of backward and forward queries is only small.

For a backward query the algorithm OBJVTM starts with fetching the current version of the object, then generates successively older versions and finally stops when the FROM version has been reached. In case of a forward query, however, the generation of the older versions is accompanied by the creation of the EXPANDED file, which contains the information to build a newer version from an older one. When the FROM version has been reached it is output and EXPANDED is used to generate successively newer versions until

Table 3
HISTCH versus OBJVTM (backward WTT query)

	chain		version	
	HISTCH	OBJVTM	HISTCH	OBJVTM
NTUP				
8	8	8	32	5
40	44	76	163	25
200	222	776	815	124
1000	4178	9140	7139	5684
5000	26717	57324	41518	40028

Table 4
OBJVTM: comparison of FORWARD and BACKWARD queries

NTUPVERTF = 2	chain		version	
	FORWARD	BACKWARD	FORWARD	BACKWARD
NTUP				
8	29	8	5	5
40	160	76	25	25
200	836	776	184	124
1000	11969	9140	8513	5684
5000	77292	57324	59996	40028

NTUPVERTF = 10	chain		version	
	FORWARD	BACKWARD	FORWARD	BACKWARD
NTUP				
8	85	8	12	12
40	280	220	127	67
200	2596	2296	638	338
1000	41386	27240	31016	16870
5000	271021	171177	219131	119287

TO is reached. Hence in this case many versions have to be generated twice. For big objects the generation of the object versions dominates the process. When NTUPVERTF is big compared to NTUPVERCT (hence the second part of *Table 4*, the part with "NTUPVERTF = 10") then a forward query can be handled just within less than two times the time a backward query takes.

7.3.3 Position of TO in history (Table 5)

Consider again a backward query. The farther we put the moment TO in history the more time it will take to generate the answer. An interesting question is: What part of the total I/O load is caused by processing the history chains from CURRENT to TO.

Assuming clustering on chain and using HISTCH, it appears that increasing NTUPVERCT with 20 implies 100 extra page accesses, see the second column of *Table 5*. Hence, the influence on total performance of generating object versions (and sorting the CHANGES file) strongly dominates the reading of history chains. When, however, clustering is on version, then the reading of the history chains is dominant: Each delta version requires a page access. Observe that an increase of m in NTUPVERCT implies an increase of (NTUP * m) in I/O load (for instance NTUP = 500 and $m = 20$ generates an extra load of 10.000).

For OBJVTM holds that given a certain object version, the cost of generating an older version is independent of the position of the version in the interval ⟨CURRENT,TO⟩. Hence the I/O load generated during the processing of the versions in the interval ⟨CURRENT,TO⟩ is (about) linearly dependent on the length of the interval, see *Table 5*, columns 4 and 5.

Table 5
Position of TO in past

NTUPVERTF = 2	HISTCH		OBJVTM	
	chain	version	chain	version
NTUPVERCT				
10	1423	6864	9632	4454
30	1523	16864	25616	11808
90	1823	46864	73567	33869
270	2723	136864	217418	100050
810	5423	406864	648974	298596

Table 6
Number of changed tuples NCHTUPLS per object version

	HISTCH		OBJVTM	
	chain	version	chain	version
NCHTUPLS				
10	4108	9549	9537	4207
30	3468	8909	8779	3049
90	2433	7874	7526	1662
270	1632	7073	6575	667

7.3.4 Number of changed tuples per object version (Table 6)

Keeping the number of tuples in an object (NTUP) and the number of versions per tuple between TO and FROM (NTUPVERTF) constant, we can still vary the number of object versions generated in the period ⟨FROM, TO⟩. If NOBVERTF is small, then the number of tuples changed in an object version NCHTUPLS is big and vice versa as the equality (NOBVERTF*NCHTUPLS) = (NTUP*NTUPVERTF) holds, see appendix. Consider the problem of generating an older object version while the object is too big to stay in the internal buffer. Then the object version must be stored partly on disk. (In *Table 6*, the internal buffer has a length of 100 Kb and that 500 tuples of 300 bytes occupy 150 KB, so 1/3 of OBJVERS has to be stored on disk.) If NCHTUPLS is small, then each update for a tuple of OBJVERS residing on disk will cause a page access to update the object version. If, however, NCHTUPLS is big, then it will often occur that tuples of OBJVERS that are stored on disk and have to be updated, reside on the same disk page; so page accesses can be saved then (see the formula NPAGHITS in the appendix). In *Table 6* NTUPVERTF has been set to 10 to allow for big NCHTUPLS values. The results show that the larger NCHTUPLS, the faster the processing of the WTT query.

8. Summary and conclusions

A database management system supporting versioned complex objects has been discussed. Complex objects are composed of small pieces, called tuples. Versions of the same complex object often share many tuples. This observation is exploited to reduce disk occupancy by storing only tuples that are changed. The storage structures of the system allow fast processing of current data by separating current and historical data. Moreover, historical data are stored efficiently by allowing compacting.

The processing of ASOF queries (asking for the state at moment T) and WALK-THROUGH-TIME (WTT) queries (asking for trend information) are elaborated. It appears that ASOF queries can be implemented rather straighforward. WTT queries, however, require a more sophisticated approach. Observe in this respect that one can distinguish two ways of processing historical databases. The first way considers processing of the history of small parts of an object (for instance tuples); in the other way one is interested in versions of objects as a whole. Probably, the history of tuples is needed most often in an office environment (think of salary and sales histories). On the other hand, versions of objects as a whole are more important in CAD/CAM environments, although here processing of the history of tuples is needed too [12].

As a consequence two algorithms to handle WTT queries were (rather detailed) described, namely HISTCH and OBJVTM. HISTCH first processes the history of the tuples of a complex object, then sorts the changed tuples and finally reconstructs the versions of the complete object. OBJVTM constructs immediately versions of the complete object using the data in the history pool.

The performance of these WTT algorithms depends very strongly on the way versions are stored in the history pool. One way is to store all versions that belong to the same tuple together. The other way is to cluster versions of changed tuples that make up an older object. As expected it appears that clustering on history chain strongly benefits HISTCH, while OBJVTM beats HISTCH when clustering is on object version. It can be concluded that if a database management system considers the office as well as the CAD/CAM environment as important then both algorithms and both ways of storing versions have to be considered seriously.

When the time period that is mentioned in the WTT query, is part of the long-ago, then the response times will deteriorate linear with the past. Only when the history pool is clustered on chain and the algorithm is HISTCH then performance is better than that. In all other cases the response times can degrade significantly and when many queries concern the long-ago the database system has to offer additional storage structures and/or facilities.

Acknowledgments

The colleagues of the IBM Scientific Center are gratefully acknowledged. Especially P. Dadam, K. Kuespert, U. Hermann, J. Guenauer, M. Scalas and Y. Teuhola have to be mentioned for their support in preparing this paper. I like to thank W. Witte for his effort in drawing all figures and B. van den Akker for his careful reading of the paper.

Appendix: Used formulas

No closed formulas for the I/O load of the algorithms will be given. Instead formulas for 'basic' quantities are derived. These formulas are used to estimate the I/O load generated by the ASOF and WTT algorithms. Programs have been written to obtain estimates.

Number of changed tuples between two object versions NCHTUPLS

The time period ⟨TO,FROM⟩ has NTUP * NTUPVERTF tuple versions and NOBVERTF object versions. This implies that NCHTUPLS, the average number of changed tuples in an object version is given by the formula:

$$\text{NCHTUPLS} = \text{NTUP} * \text{NTUPVERTF} / \text{NOBVERTF} .$$

Size of current object NPACUOB

The current object is stored clustered in the current pool. The number of pages occupied by this object is equal to NPACUOB.

$$\text{NPACUOB} = \text{NTUP} * \text{TUPLEN} / \text{PAGLEN} .$$

Size of CHANGES file NPACHGS

The CHANGES file occupies NPACHGS pages and is used in the HISTCH algorithm in the processing of both forward queries. The CHANGES records have a length of CHNGLEN bytes containing a prefix connected to a tuple version. In case of a backward query CHNGLEN = DELTLEN + 40, otherwise (forward query) CHNGLEN = TUPLEN + 40.

$$\text{NPACHGS} = \text{NTUP} * \text{NTUPVERTF} * \text{CHNGLEN} / \text{PAGLEN} .$$

Size of EXPANDED file NPAEXPD

This file is used in the OBJVTM algorithm when a forward query has to be processed. The records of the EXPANDED file contain the complete after image of a tuple. The size NPAEXPD is computed easily.

$$NPAEXPD = NTUP * NTUPVERTF * TUPLEN / PAGLEN .$$

Sequential reading of versions NPACHNSEQ and NPAVERSEQ

The algorithm HISTCH processes the history of a chain at a time. When the history pool is clustered on chain, then the number of page accesses to read all tuple versions sequentially is approximated by NPACHNSEQ. There are NTUP chains and each chain starts on a new page (assumption). Starting at the current tuple one page access is required to fetch the first version in the history pool.

$$NPACHNSEQ = NTUP * (1 + DELTLEN * (NTUPVERCT + NTUPVERTF)/ PAGLEN)$$

Consider now the algorithm OBJVTM and suppose that the history pool is clustered on object version. It is assumed that each version starts on a new page. The formula for NPAVERSEQ, the number of page accesses needed to read all tuple versions sequentially, is derived analogously to NPACHNSEQ.

Again the factor '1' models the access from current to history pool.

$$NPAVERSEQ = (NOBVERCT + NOBVERTF) * (1 + DELTLEN * NCHTUPLS / PAGLEN) .$$

Direct reading of versions NPACHNDIR and NPAVERDIR

In OBJVTM the CLICK table dictates from which chain the next delta version(s) have to be taken (see *Fig. 11*). A delta version is located on a certain page. Assume clustering on the history pool on chain and call AVAILPG (a 'local' variable) the number of pages in the internal buffer that are available for buffering of history pool pages. (AVAILPG is equal to IBLEN/PAGLEN – size of OBJVERS.) The number of chains is NTUP and within a chain the tuples are time ordered. If we assume that AVAILPG < NTUP and that the chain from which the next tuple version has to be taken is randomly selected, then (AVAILPG/NTUP) is the probability of having the required page already in internal buffer. NPACHNDIR gives the number of page accesses to read all versions directly.

$$NPACHNDIR = NTUP * (NTUPVERCT + NTUPVERTF) * (1 - AVAILPG/ NTUP) .$$

When clustering is on object version, then the scanning of all history chains HISTCH requires NPAVERDIR page accesses. As above, use of available buffer space could be made. In this case, however, the benefit is not high. It is reasonable to assume that within an object version the tuple versions are stored 'randomly' and not ordered in one way or another like was the case in NPACHNDIR. Hence if there are only a few, long object versions occupying many pages, buffering will not help very much. This justifies the following approximation for NPAVERDIR.

$$NPAVERDIR = NTUP * (NTUPVERCT + NTUPVERTF) .$$

Generating object versions NPAGHITS

Suppose a new object version has to be created with the help of NCHTUPLS changes consisting of, among others, the number of the page on which the tuple to be modified is stored. After ordering all changes on page number the following question arises: How many page accesses are needed to access all tuples that have to be updated, see *Fig. 9.* This problem occurs in both algorithms and is known in literature ([14], [24]). Under certain conditions, which satisfied in this paper, a good approximation is given by NPAGHITS.

$$\text{NPAGHITS} = \text{NPACUOB} * (1\text{-exp}(\text{-NCHTUPLS} / \text{NPACUOB})) .$$

Suppose the start version and NCHTUPLS records (for instance from CHANGES) defining the updates are given. The problem is now to estimate the number of page accesses needed to generate a new object version. Suppose that a fraction 'fract' of the start version fits into internal buffer. To update a page of the object that is not in internal buffer, a read and a write are necessary. NCHTUPLS records give rise to NPAGHITS page hits. The formula below gives the number of page accesses needed to generate an object version. Notice that fract = 1 implies that no accesses are required.

$$(1 - \text{fract}) * 2 * \text{NPAGHITS} .$$

Sorting M bytes with a buffer of N bytes SORT(M,N)

Suppose a file of M bytes is stored on disk. How many page accesses are needed to sort this file given the fact that an internal buffer of N bytes is available? If $M < N$ then the file has to be read, internally sorted and written back. Otherwise, first strings are made and a k-way merge has to applied. Using a replacement selection method, strings of size 2*N bytes may be expected. In the formula below 'trunc' indicates the truncate function.

$$\text{SORT(M,N)} = 2 * \text{trunc}((2 + \text{trunc(b)}) * \text{M/PAGLEN})$$
$$\text{where } b = \ln(M/(2*N)) / \ln(k) .$$

References

[1] M. Adiba and N. Bui Quang, Historical multimedia databases, in: *Proc. VLDB 86*, Kyoto, Japan (1986).

[2] T.L. Anderson, Modeling time at the conceptual level, in: *Proc. Second Internat. Conf. Databases*, Jerusalem (1982) 273–297.

[3] M.J. Carey et al., Object and file management in the EXODUS extensible database management system, in: *Proc. VLDB 86*, Kyoto, Japan (1986) 91–100.

[4] J. Clifford and D.S. Warren, Formal semantics for time in databases, *ACM TODS* 8 (1983) 214–254.

[5] P. Dadam et al., A DBMS prototype to support extended NF^2 relations: an integrated view on flat tables and hierarchies, in: *Proc. SIGMOD '86*, Washington D.C. (1986) 356–367.

[6] P. Dadam et al., Integration of time versions into a relational database system, in: *Proc. VLDB '84*, Singapore (1984) 509–522.

[7] P. Dadam and Y. Teuhola, Managing schema changes in a time-versioned non-first-normal-form relational database. Datenbanksysteme in Buro, Technik und Wissenschaft, in: *Proc. GI-Fachtagung*, Darmstadt (1979) 161–179.

[8] D. Ecklund et al., DVSS: A distributed version storage server for CAD applications, in: *Proc. VLDB 87*, Brighton, England (1987) 443–454.

[9] L. Gruendig and P. Pistor, Land-informations-Systeme und Ihre Anforderungen an Datenbank-Schnittstellen, in: J. W. Schmidt, ed., *Informatik-Fachberichte* 72 (Springer, Berlin, 1983) 61–75 (in German).

[10] T. Haerder et al., PRIMA – a DBMS prototype supporting engineering applications, in: *Proc. VLDB 87*, Brighton, England (1987) 433–442.

[11] R. H. Katz and E. Chang, Managing change in a computer-aided design database, in: *Proc. VLDB 87*, Brighton, England (1987) 455–462.

[12] R. H. Katz and T. J. Lehman, Database support

for versions and alternatives of large design files, *IEEE Trans. Software Engrg.* SE-10 (1984) 191–200.

[13] W. Kim et al., Composite objects revisited, in: *Proc. SIGMOD* '89, Oregon (1989) 337–347.

[14] M. R. Kloprogge and P.C. Lockemann, Modelling information preserving databases: consequences of the concept of time, in: *Proc. VLDB 83*, Florence, Italy (1983) 399–416.

[15] R. A. Lorie and W. Plouffe, Complex objects and their use in design transactions, in: *Proc. Annual Meeting – Database Week: Engineering Design Applications IEEE*, San Jose, CA (1983) 115–121.

[16] V. Lum et al., Design of an integrated DBMS to support advanced applications, in: *Proc. Internat. Conf. Foundations of Data Organization*, Kyoto, Japan (1985).

[17] V. Lum et al., Designing DBMS support for the temporal dimension, in: *Proc. SIGMOD 84*, Boston (1984) 115–130.

[18] P. Pistor and F. Andersen, Designing a generalized NF^2 model with an SQL-type language interface, in: *Proc. VLDB*, Kyoto, Japan (1986) 278–298.

[19] P. Pistor and R. Traunmueller, A data base language for sets, lists, and tables, Technical Report TR 85.10.004, IBM Scientific Center, Heidelberg, West Germany, 1985.

[20] B.-M. Schueler, Update reconsidered, in: G. M. Nijssen, ed., *Architecture and Models in Data Base Management Systems* (North-Holland Publ. Comp., Amsterdam, 1977) 149–164.

[21] M. Stonebraker, The design of the POSTGRES storage system, in: *Proc. VLDB 87*, Brighton, England (1987) 289–300.

[22] A. U. Tansel and L. Garnett, Nested historical relations, in: *Proc. SIGMOD '89*, Oregon (1989) 284–293.

[23] P. Valduriez et al. Implementation techniques of complex objects, in: *Proc. VLDB '86*, Kyoto, Japan (1986) 101–111.

[24] A. IJbema and H. M. Blanken, Estimating bucket accesses: a practical approach, in: *Proc. Second Conf. Data Engineering*, Los Angeles, CA (1986).

Henk Blanken joined, after receiving his masters degree in mathematics, in 1966 Philips Computer Industries in the Netherlands. From 1971 he worked at the University of Twente in the group Information Systems. From this university he received in 1984 a PhD degree in Computer Science. During a one-year visit to the IBM Scientific Center at Heidelberg he contributed to the AIM/II project. His current research interests are non-standard database applications and storage structures.