HISTORICAL MULTI-MEDIA DATABASES

M.Adiba , N.Bui Quang

Laboratoire de Genie Informatique-IMAG
GRENOBLE-University, BP.68
38402 Saint Martin d'Hères, France

ABSTRACT

We develop here several notions in order to define, store and manipulate historical data in generalized or multi-media databases i.e databases which are able to handle non-classical data (text, image, voice).

For a given database object X, an history is a sequence of the successive values that X took accross time. This notion has been studied before but here, we generalize histories by considering different types of database objects. Particularly, we apply the history notion to multi-media documents and we propose extensions for data definition and manipulation languages.

The following points are discussed : definition of object histories (periodicity, persistency), propagation of historicity, update and history management, history and schema modification.

1- INTRODUCTION

Today, database systems are only able to provide more recent and consistent versions for data. However for many applications this may not be always desirable. More and more, historical data is needed (for instance in economical databases) or "old" data is required by several applications. As an example consider a rather traditional business application where a user wants to ask queries such as :

"What is the evolution of John's salary during the last five years ?"

Dealing with historical data is dealing with time and several previous works have been done in databases [CLI 83]. Most of these studies introduced time in the data structure, for instance [BOL 82, KLO 81, OVE 82, SNO 85]. In TERM [KLO 81] time is explicitely introduced at the level of the Entity-Relationship model. It is considered as a particular data type with several consistency constraints associated with it. Also [OVE 82] described a time expert incorporated into the relational DBMS INGRES.

Other researches were concerned with the extension of a DBMS in order to provide good data structure for historical data but nothing was said about the definition and manipulation by end-users of such histories [LUM 84].

In non traditional database applications e.g office automation or CAD database objects are complex but it is also necessary to deal with successive versions. As an example [KAT 85] describe a version server for CAD data in order to provide the designers with a way to manage successive states of the object under design.

Extensions for version management in CAD databases are also described by [DIT 85].

Here, we address the problem from a chronological point of view i.e we consider the evolution of a (complex) database object over time in order not to be specific of a given application.

In a previous work [ADI 85] we addressed the problem of incorporating time in a generalized or multi-media database system. Time was treated as a special data type which can be used when defining the schema and manipulated by special extensions to the DML. Here we describe an extension of this work in order to provide solutions for managing histories for generalized data. More precisely our goals are the following :

1) Insure fast access to current data

In general and although providing historical data management we think that most of the queries will concern current versions and that a good response time must be provided.

2) Provide different granularity for historical data

In relational terms this means that the user wants to define what attributes should be historical in a given relational schema. For instance an employee's name and birthdate dont change accross time but his/her address or salary does. In our model, which is based on the Entity-Relationship approach, we can also provide historical aspects on entities and relationships.

3) Extend the data definition and manipulation language in order to manipulate historical data

4) Take into account the evolution of the conceptual schema over time

Because relational DBMSs provide dynamic schema modification we apply also our versioning approach at this level.

The organisation of this paper is the following. In section 2 we define the main concepts for historical data. In section 3 we discuss the problem of historical data with regards to data models and we show that historicity can be provided at several levels of granularity. Sections 4 and 5 deal with history manipulation. We describe several extensions which can be made to a SQL-like language in order to query and update data. Sections 6 and 7 describe mechanisms for historical data management, and storage. In section 8 we discuss historicity with regards to schema modification and we conclude the paper by section 9.

2- HISTORICAL DATA : MAIN CONCEPTS

Consider a database object V which takes over time successive values. Let us denote by (vi,ti) the couples (value,time) for V : at time ti V took value vi. With such histories we can answer to the following kinds of queries :

(1) What was John's salary at January 1985 ?

(2) What were John's salaries during the last twelve months ?

Proceedings of the Twelfth International Conference on Very Large Data Bases **Kyoto, August, 1986**

(3) Give us a complete history of John's salary

Query (1) refers explicitely to a precise time in the past. Query (2) is relative and deals with a periodic notion (month by month) and query (3) must provide all the couples (salary, time) which correspond to successive versions of the salary.

Our approach for historical data is based upon the three following notions :

1) Periodicity of the ti, i.e the sequence of times we want to consider for the vi values. Note that this periodicity may not be related to a change of value. For instance if we want John's salary for the last twelve months, this does not mean that this salary changed each month.

2) Modification: when value v is changed by value v', this modification may or may not give, **in a given history**, a new version for V. It is the database user responsability to define precisely when a change must be incorporated into the history.

3) Persistency of values in the database. Theoretically we can consider that successive versions are stored in an "infinite" space. However, the user may want sometimes to keep only the last **n** values.

Let us consider the relation EMPSAL(E#, NAME, JOB, SALARY) which records for each employee whose number is E# his name, his job and the corresponding salary. We want to keep for each employee the evolution of his/her salary. A first solution is to add an attribute DATE and let the user deal with it. This may lead to inconsistencies if the DBMS does not manage explicitely the time concept. If it does, then all the tuples of the relation are "equally treated" and there is no notion of last (current) version and previous versions. So, it is necessary for the DBMS to handle explicitely historical data and this is precisely what we propose. Without adding the DATE attribute we are going to define EMPSAL as an historical relation with a periodicity of a year and a persistency for 10 years. In this way the last version will refer to the current state of the employees and previous versions to their histories.

Our model is not based upon the n-ary relational model but on the entity relationship approach extended with the notion of type. Our data definition language will be extended by the key word "**dynamic**" for defining historical data. For instance :

Example 1 :

```
type : EMPSAL : dynamic entity
                each  year
                last  10
                with  time > day
                 e# : integer;
                 name : string(10);
                 job : string(10);
                 salary : (200..3000);
               end;
```

In this example "**each year**" defines the periodicity and "**last 10**" defines the persistency. The time we consider here is composed with (year, month, day, hour, minute, second) and "**time > day**" means that we want the ti to be composed only with (year, month).

A first approach for building histories is to let the user decide when he/she wants to keep a new version after one or several modifications took place on a given object. In this case, we can speak about **Manual History** (MH). As an example, consider a document report for which we want to keep in the database several versions :

```
type  report : dynamic manual document
               begin
                 introduction : text ;
                 body : list (1,*) of
                    chapter : list (1,*) of text ;
                 conlusion : text ;
               end ;
```

Editing a given report occurrence r may involve over time many modifications. However at a given time the user may decide that the current version should be kept because it corresponds to a **"good" version**. In this case the history will contain all the "successive" versions that the **user decided** to keep.

If we want to have automatic treatment for histories, we can use an "**each**" clause to define the periodicity and in this case, we speak about a **Periodical History** (PH). In a PH, if an object is modified within the period, this modification affects the current version. New versions are only generated at the end of each period by putting in the history a copy of the current version. However when there is neither the **manual nor each** clause, this means that we want to store successive versions and then we speak about **Successive History** (SH).

As another example, we define a type exchange which can be used for an attribute in an entity or a relationship (see section 3) :

```
type exchange : dynamic
                real ;
```

Here, each time a value of this type is updated, the old value is stored into the history.

The "**last**" clause refers to **persistency** : it indicates how many old values we want to keep. A persistency of **zero** corresponds to a **static** type and if there is no "**last**" clause the persistency is theoretically illimited.

The "**with**" clause as we saw it before indicates the format associated with the time **ti** stored in a given history. This time is an internal time or **physical time** as it is managed by the computer system. We take the assumption that a difference may exist between the real world time of an event and its recording into the database, but this difference is irrelevant. Logical time is also disccussed in [DAD 84] and [SNO 85].

3- HISTORICAL DATA AND DATA MODELS

In the n-ary relational model, the history notion can only be applied at the relation level. This means that one must be able to (re)build all instances **ri** that a given relation R had accross time. Going from an instance (ri,ti) to (r_{i+1},t_{i+1}) is done by inserting, deleting or updating tuples, (r_{i+1},t_{i+1}) is the current version.

Our work is done in the framework of the TIGRE project for generalized databases. The TIGRE data model is based upon the Entity-Relationship approach [LOP 83, VEL 85b]. By introducing types we want to introduce more semantics into the description of database objects. Therefore, the history notion can be seen differently than in the (flat) relational model. Each entity instance has an internal surrogate and eventually several attributes. By defining some attributes as dynamic types it is possible to maintain an history of all the successive values taken by these attributes, for a given entity instance.

More generally there are several kinds of data types : for attributes we can have basic types (integer, real, string etc.), record, list or document. The document constructor type defines a complex tree structure whose leaves are big size objects containing text, graphics, pictures, voice coded data. Entities, relationships and also generalization, specialization or agregation notions are refered as class types in TIGRE. Each class type may have typed attributes as mentioned above. We can apply our history notion on record, list, document types but also on class types but in this case, subcomponents cannot be historical. We do not accept history of history for semantic reasons. However, when an entity is dynamic this dynamicity is **propagated** on its attributes. We will come back later on this problem.

Consider the following example :

Example 2 :

```
type T-employee : dynamic entity
                    name : string(20) ;
                    address : t-address ;
                    salary : (300..40000) ;
                    department : string(10) ;
                  end;
where t-address is defined as :
  type t-address : record
                     number : integer ;
                     street : string(15) ;
                     town : string(15) ;
                     zipcode : integer ;
                   end;
```

With such a definition and for a given employee we can store the history of all addresses and answer queries such as : "Give me all the towns where John lived". We will have also the history of all the employees created into the database since the begining of its implementation.

For the database administrator the choice where to put the dynamicity either at the attribute or entity levels is a design problem which is application dependent. Our goal here is to provide specific tools to handle different kinds of histories in a generalized DBMS. When a type is not defined as an history we refer it as a static type otherwise it is a dynamic one.

Let us now discuss the problem of **history propagation**. From an history type to its components historicity keeps the same **charateristics** that is same periodicity, persistency, MH, SH or PH etc. We said that a type is **explicitely defined** as an history if it is defined as a dynamic type otherwise a type can be **implicitely defined** as an history because it is a component of an history type. This history inheritance can be stated more completely as in the following :

1) History and basic types

- By construction a basic type is predefined therefore it cannot be dynamic.

2) History and built types

- Renamed, record, list, document types and component types of a static document can be dynamic.

For multi-media documents we have often the case where it is not necessary to store all the successive versions of a document but only for subparts of it. However this dynamic document notion must be taken with care because the corresponding objects are big size objects (see section 7).

- Component types for a dynamic record or a dynamic list are static.

- Nodes of a dynamic document type are implicitely dynamic.

3) History and class types

- Component types of a dynamic class are implicitely dynamic.

- A dynamic relationship can only link static entities. In this case we have to propagate the dynamicity to these entities. Let us see this with an example :

```
type shipment : dynamic relationship
                  each month
                between supplier and product
              quantity : integer ;
            end ;
```

We want to store quantities of products supplied month by month by each suppliers. For consistency, the corresponding suppliers and products associated with the corresponding months must exist. This means that historicity has to be propagated to the linked entities. This propagation is also discussed later.

- A specialization of a dynamic type is implicitely dynamic.

- In a dynamic aggregation, components become implicitely dynamic.

Note that in certain cases, history propagation can be cascaded until basic component types.

Example 3 : Consider two entities SCHOOL and STUDENT :

```
type SCHOOL : entity
        sname : string(20) ;
        address : t-address ;
        tel : integer ;
        teaching : description ;
      end;

type STUDENT : entity
        name : string(20);
        b-date : date ;
        address : t-address ;
        scolarship : dossier ;
      end;
```

and dynamic relationship INSCRIPTION
type INSCRIPTION : *dynamic relationship last 5*
between STUDENT : student (1,1)
and SCHOOL : school (1,*)
ins-date : date ;
academic-year : time-interval ;
speciality : string(15) ;
category : (C1, C2, C3) ;
course : list-of-lecture ;
end;
where : date is a renamed type (time > hour);
t-address is a record
description and scolarship are documents
list-of-lecture is a list of string(20)
They are static and predefined.

The relationship INSCRIPTION is a successive history (SH) with a persistency of 5. This historicity is propagated to SCHOOL and STUDENT and then to t-address, dossier, description and list-of-lecture which become dynamic.

The historicity propagation is not trivial. For instance let us consider three static entities types and two relationships R1 and R2, R1 linking A and B, R2 linking B and C. If we define R1 as dynamic this will implicitely make A and B dynamic. Then, if we want R2 to be dynamic we have a problem because B is already dynamic. However if both R2 and B have the same historicity, making dynamic should be possible. More generally and by considering compatible histories we allow the historicity propagation to take place on implicitely defined histories.

4- HISTORY AND DATABASE UPDATE

Updates made on a dynamic type are to be considered differently than those made on static types.

4.1- Update operations

- Insertion of an occurence in a dynamic type is always made on the current version with the timestamp for the creation of this occurence.

- Updating a successive history (SH) is done in two phases : tuples to change with their associated time ti are copied in an history area (see section 6) and then new tuples are stored in the current version. Hence, for a given object its current version is physically separated from its history, insuring fast access to it. If the user wants to update a manual history (MH) he/she should explicitely say that this modification will generate a new version in the history or not. In the DML this means that the UPDATE command modifies the current version without keeping the old or new version in the history. If the user wants to keep a new version either before or after an update, he/she can use the command GENERATE VERSION.

- For a periodic history (PH) modification is done directly as if there were no history. However, periodically, data is copied in the history area.

For deletion we have to consider the following cases :

- If a tuple of a dynamic class is deleted it is written in the history area with a special deletion flag. This deletion must be followed by the deletion of all its component types.

- If a tuple of a static class which contains dynamic attributes is deleted, then we have to delete for each such attribute, the corresponding history. For instance if employees have dynamic addresses when an employee instance is deleted so all his successive addresses must be deleted.

4.2- Persistency

Each time a data is introduced into the history area we have to verify its persistency. This is an important notion because it is a way to control the size of the history area. Hence, if p is the persistency, the (p+1) insertion must trigger the deletion of the oldest version in the history area. In order to improve performance we can think of several techniques to handle space in the history area. For instance we can put a flag on areas which are freed and then, have some garbage collector policy to gather unused space.

4.3- Correction and modification

By definition it is not possible to update data stored in the history area. However it is always possible that some mistakes or inconsistent data have been introduced there. This is a very delicate problem but we think that we have to provide a special manipulation primitive in order to correct historical data. Obviously this primitive has to be used very carefully by authorized users but semantically this operation is different from insertion, deletion and updating which are always done on the current version.

Note also that sometimes we have to correct the current version without affecting the corresponding history. This means that in this case, the automatic history mechanism should be disconnected.

5- HISTORY AND MANIPULATION LANGUAGE

The data manipulation language must provide the user with a way to manipulate current and historical data. In the following we are going to describe some extensions that we made to the TIGRE data definition and manipulation language called LAMBDA [VEL 84, VEL 85b]. This language is a SQL-like language but it is not intended to be given to the end-user, rather, it is going to be embedded into general (typed) programing languages or offered to end-user through graphical interfaces.

In [ADI 85] we described some extensions concerning the manipulation of time in general. Here, we concentrate on the manipulation (interrogation) of historical data. In this case, time can be used in two different ways :

1/ Explicitely (or absolute)

Q1: "Give me the value of V at time t"
Q2: "Give me all the values of V in the interval [t1,t2]"
Q3: "Give me all the values of V after (before) time t"

2/ Implicitely (or relative)

In this case we want to consider versions for V such as :

Q4: "Give me the last version of V (current version)"
Q5: "Give me the three last versions"
Q6: "Give me all the versions of V"

Our main extension in the language is to use the key word "**version**" to refer to historical data.

For illustrating the manipulation of historical data we are going to consider the following example which concerns employees of a given company :

Example 4 :

```
type EMPLOYEE : entity
        name : string(20);
        address : t-address ;
        category : (engineer, secretary, technician);
        contract : t-contract;
  end;
```

The t-address type has been defined in example 2 and let us suppose that the contract of each employee is a document which can be modified accross time and for which we want to keep the successive history (SH) :

```
type t-contract : dynamic document with time > hour
                  structure
                   begin contractant
                     begin employer : constant T0
                           employee : reference bearer
                     end ;
                     body : list(1,5) of clause
                     signature : picture
                   end ;
                  constant text T0 := 'Company ABC'
                  parameter
                     bearer : text;
                     contract-duration : duration;
                     signature-date : time > hour;
                     department : string(20);
                     level : string(20);
                  function salary : sal(level, department);
  end;
```

This definition needs some explanations because it refers to specific notions of the TIGRE model. Leaves of the document are generalized (or multi-media) data whose nature are either text, picture etc. A document is defined by its structure which is a tree but also by specific components which are constants, i.e data which are present in all the instances of the document and parameters. Parameters are used to model variable parts of documents. They may require to appear in a fixed place in the tree structure (for instance the "bearer" parameter above). They are components with direct access to their value. This access is expressed through the use of the LAMBDA language and may involve user defined functions such as the one defined for the salary. Here the salary stored in a contract is computed by the sal function using the two parameters : *level and department*.

Example 5 : The last address of John :

```
SELECT last version of e.address
FROM  EMPLOYEE e
WHERE e.name = 'John'
```

For historical data and by convention we associate to each version an integer or a function called **last** : 1 is associated to the first version, 2 to the second ... **last-1** to the before last version and **last** to the current version. Using this convention the previous example is equivalent to :

```
SELECT  e.address
FROM    EMPLOYEE  e
WHERE   e.name = 'John'
```

Example 6 : Names and first three salaries of all the engineers. The salary is computed from the document by using the function sal :

```
SELECT  e.name,1..3 version of eval
     sal(parameter level, department) of e.contract
FROM   EMPLOYEE e
WHERE e.category = 'engineer'
```

Example 7 : The three last salaries of John

```
SELECT 3 last version of eval sal(parameter level,
                department) of e.contract
FROM   EMPLOYEE e
WHERE e.name = 'John'
```

Example 8 : We want all the versions of the first clause of John's contract :

```
SELECT  all version of clause 1 of e.contract
FROM   EMPLOYEE e
WHERE  e.name = 'John'
```

We can also have an equivalent formulation using **1..last version**.

Let us now give some examples with an explicit time.

Example 9 : John's address at 1982/6/30 :

```
SELECT version at '1982/06/30' of e.address
FROM   EMPLOYEE e
WHERE e.name = 'John'
```

The key-word **version at t** is used to query historical data and refering to a particular time. The time value t must have a granularity greater or equal to the time type associated with the corresponding history.

Example 10 : John's salaries during 1970-1980 (the key-word **during** must be followed by a value of type time-interval)

```
SELECT version during '1970-1980' of eval
     sal(parameter level, department) of e.contract
FROM  EMPLOYEE e
WHERE e.name = 'John'
```

Example 11 : John's salaries after 1980 (see example 1) :

```
SELECT version after '1980' of e.salary
FROM   EMPSAL e
WHERE  e.name = 'John'
```

Note that to a periodic history (PH) on V it is associated a periodic sequence of time T = {t1,t1,..,tn}. Suppose that we want the value of V at time t0. If t0 belongs to T the answer is easy to give. On the contrary, one solution may consist of finding t0 of T "closest" to t0 and to give as an approximative result the corresponding value. However it may exist ti and t_{i+1} of T equidistant from in this case we can take by convention the value at time t_{i+1} because it is the more recent one.

Example 12 : List of employees which are in the company for more than 10 years (we suppose that all the contracts are finishing this year) :

```
SELECT e.name
FROM   EMPLOYEE e
WHERE  SUM[ all version of parameter contract
-duration of e.contract] >= '10 year'
```

Example 13 : We want to correct an error in John's contract : the parameter contract-duration must be 3 years

```
CORRECT parameter contract-duration of e.contract
                        := '3year'
FROM    EMPLOYEE e
WHERE   e.name = 'John'
```

Example 14 : The zipcode of the before last version of Martin (38056) is not correct, it should be corrected to 38072 :

```
CORRECT last-1 version of e.address.zipcode
                        := 38072
FROM    EMPLOYEE e
WHERE   e.name = 'Martin'
```

Note that here the modification is done on an history value.

6- HISTORY MANAGEMENT

Traditionally different operations can be issued on a database : query, insertion, deletion, update or schema modification. In a more general context of database management, applied to different kinds of applications, objects can be of different granularity : tuples, relations, big objects (e.g documents or designs) etc. So, dealing with histories, the question arises : what should be the data unit to store in such histories ?.

For relational databases one way to answer this question is to "historicize" the tuples and solutions for this can be found in [LUM 84], for instance. In our prototype, built upon a relational DBMS, we can make a distinction between data internally mapped to n-ary relations and big objects (documents) which are managed separately but identified by internal surrogates. For the first kind, we also took the tuple as the history component. In a Successive History, before updating a tuple, we store the old value in what we call the history area. The new updated tuple is stored in the current version of the relation.

For a Periodic History, tuples are periodically written into the history area but updates are made directly on the current version as if there was no history. In order not to waste space we store a tuple in the history area only if it differs from the last one in the history.

This strategy implies that we maintain for each "relation" two versions : one for the current data and another for the history (see section 7).

For documents and complex objects, in general, the data unit is a leaf of the tree structure. These leaves are what we call big objects. Modifying a document is changing one (or several) leaf content without modifying the structure of the tree. If a particular document type is updated this will give different versions for particular leaves. In this case leaves are naturally the data unit for histories (see section 7).

For both relational data and documents, history management is simplified by using surrogates. The couple (surrogate,time) is in fact the primary key for the relations stored in the history area.

When we delete a data unit (tuple or document leaf) we store in the history area t[illegible] it has been deleted and this terminates the [illegible].

7- HISTORY AREA

Let us now give some details about the content of the history area which allows to store separately historical data and current data. For each history an additional attribute is put by our system in order to contain the timestamp associated to each operation : creation, deletion, update. Note that this timestamp is however invisible to the users.

7.1-Relations associated with non document historical types

In the database catalogs, for each relation R which corresponds to a dynamic type we generate a corresponding historical relation H-R which is going to contain successive or periodical data. For instance to the type definition of example 1 will corresponds two relations one for the current version of EMPSAL and one for historical data of EMPSAL.

7.2- Document history

Before going into more details about historical document let us describe how these documents are managed by the TIGRE system. Documents are treated separately by a module called the Big Object Manager (or BOM). In this module each big object is identified by an internal surrogate (BOID). A given big object may span over several logical data blocks and these blocks are chained together. The BOM has an internal catalog to describe big object storage. This catalog is composed with two relations :

(1) BIGOB (BOID, BLOCK, SIZE) which gives the identifier of each big object, the address of its first block and its size.

(2) BIGOC (BLOCK, SIZE, NEXT-BLOCK) which gives the identifier of a block, its size and the next block in the chain.

For document storage we have the relation

DOCLEAF (DOCID, LEAF-NAME, BOID)

where each tuple describes a leaf (LEAF-NAME) of the document (DOCID) and this leaf is stored as the big object BOID.

7.2.1- Document and Periodic History

For this kind of history the relation DOCLEAF is historicized : a relation H-DOCLEAF is created. Each document leaf is copied periodically from the current version. This copy is then stored as big object and the catalogs are updated accordingly. In this solution, the different versions of a document leaf are identified by a set of BOID associated with the corresponding time.

7.2.2- Document and Successive History

A similar approach can be taken for this kind of history. When a document leaf is modified the tuple which describes the old version in DOCLEAF is deleted and stored into H-DOCLEAF together with the corresponding timestamp. The new tuple for the new leaf is then inserted into DOCLEAF. This solution, however is rather too expensive because it does not take into account the fact that only some blocks of the document leaf were modified. A more realistic approach consists of historicizing the catalogs BIGOB and BIGOC. In this case, two successive versions of a document leaf may have the same blocks.

Hence, document history management is made **through a subset of the BOM catalogs or via their history area.**

Remark : To allow document sharing between several users we have to historicize the catalog which describes the document tree structure.

8- HISTORY AND SCHEMA MODIFICATION

Several relational DBMS allow their users to dynamically change the database schema [CHA 76, MIC 82] : add or delete attributes. Then, the question arises : how it is possible to apply our history approach to this problem ?.

In the TIGRE model schema modification concerns the addition of a class or document types. It is therefore possible to add (delete) an attribute in a entity or to add (delete, modify) a substructure in a given document type.

In order to maintain historicity in case of schema modification we simply apply the history notion to the database catalogs. For instance the (meta) relation CAT-ATT which describes entity attributes can become a **successive version history** in order to reflect attribute creation and/or deletion.

The same policy is applied to the catalogs which describe the document structure and history structure of a given TIGRE type. For a more detailled description on this problem see [BUI 85].

9- CONCLUSION

We presented here several mechanisms in order to define, store and manipulate historical multi-media data. We think our approach is original for several reasons :

(1) We tried to define basic concepts for historical data, namely periodicity, modification and persistency.

(2) We apply these concepts in the framework of a semantic data model

(3) We are dealing with generalized data.

(4) We tried also to separate clearly between historical data as it is defined and manipulated by users and the internal mechanisms that the DBMS provide in order to insure fast access to current data while managing different kinds of histories : Manual, Periodical or Successive.

This work is done in the framework of the TIGRE project where a prototype of a generalized DBMS is under construction. Application of this prototype is mainly office automation but we think that concepts developed here can also be applied to other applications such as CAD databases. In the prototype we are currently implementing history management. An important point of course is space management because for big size and complex objects histories can be difficult to implement. One solution to this problem is to study the use of optical disks in such an environment [BOU 85].

As an extension of this work we are now considering the snapshot notion [ADI 80] for generalized data. We can consider snapshot as a gereralization of historical data because a snapshot may involve several database objects. A more formal work on this problem in under way [BUI 86].

Acknowledgements : Several ideas for dealing with time in generalised DBMS are due to our colleague J.Palazzo. We are thankful to our colleagues of the TIGRE project and particulary M.Lopez for his discussions about the document manager, F.Velez and C.Collet for patient reading of the manuscript, for their pertinent remarks and their suggestions to this work.

REFERENCES

[ADI 80] M.Adiba,B.Lindsay "Database snapshots" VLDB Montreal Oct 1980

[ADI 81] M.Adiba "Derived relation : A unified mechanism for views,snapshots and distributed data" VLBD Cannes Sept 1981

[ADI 85] M.Adiba,N.Bui Quang,J.Palazzo "Time concepts for generalized data bases" ACM Annual Conference, Denver Colorado Oct 1985

[BOL 82] A.Bolour,L.Anderson,J.Dekeyser,T.Wong "The role of Time in Information processing:A survey" ACM SIGMOD Record 12.3 April 1982

[BOU 85] P.Boursier, M.Scholl, C.Triffaut "Organisation et gestion de données sur disque optique

numérique non réinscriptible" Journées Bases de Données Avancées, INRIA, St.Pièrre de Chartreuse, Mars 1985

[BUI 85] N.Bui Quang "Gestion des historiques pour la base de données généralisées TIGRE" R.R TIGRE N.29 IMAG,Grenoble Juin 1985

[BUI 86] N.Bui Quang "Aspects dynamiques et gestion du temps dans les SGBDs généralisées" Thèse de l'INPG, IMAG Grenoble 1986 (in preparation)

[CHA 76] Chamberlin D.D et al."SEQUEL 2: A unified approach to Data definition, manipulation and control" IBM Journal of Research and development Nov 1976

[CLI 83] J.Clifford, D.S.Warren "Formal semantic for time in databases" ACM TODS Vol 8 Nb 2 June 1983

[CLI 85] J.Clifford, A.Tansel "On an algebra for historical relational databases : Two views" Proceedings of ACM-SIGMOD Austin May 1985

[COD 79] E.F.Codd "Extending the data base relational model to capture more meaning" ACM TODS, Vol.4, N.4, 1979

[DAD 84] P.Dadam,V.Lum,H.D.Werner "Integration of Time versions into a Relational Database system" VLDB Singapour August 1984

[DIT 85] K.R.Dittrich, R.A.Lorie "Version support for engineering database systems" IBM Research Report RJ 4769 July 1985

[KAT 82] R.H.Katz,T.J.Lehmen "Storage structures for versions and alternatives" University of Wisconsin, Madison R.R (479) Mai 82

[KAT 85] R.H.Katz, M.Anwaruddin, E.Chang "A Version Server for Computer_Aided Design Data" Report N. UCB/CSD 82/266 University of California Nov/1985

[KLO 81] M.R.Klopproge "TERM : An approach to include the Time dimension in the Entity-Relationship Model" Proceed 2nd Conf on E-R Approach Washington 1981

[LOP 83] M.Lopez,J.Palazzo Oliveira,F.Velez "The TIGRE data model" R.R TIGRE N 2 IMAG et CII-Honeywell Bul, Grenoble Nov 1983

[LUM 84] V.Lum et al."Designing DBMS support for the temporal dimension" Proceedings of the SIGMOD 84 Conference, June 1984

[MIC 82] F.Fernandez,L.Ferrat,J.Lee Y,G.T.Nguyen MICROBE. Manuel de référence Laboratoire IMAG, Grenoble 1982

[OVE 82] R.Overmyer,M.Stonebraker "Implementation of a time expert in a database system" ACM-SIGMOD Vol 12 N 3 April 1982

[PAL 83] J.Palazzo Oliveira,F.Velez "La correspondance de schémas entre les modèles TIGRE et relationnel" Rapport de recherche TIGRE N 5 Nov 1983

[PAL 84] J.Palazzo Oliveira "Un modèle de données et sa représentation relationnelle dans un système de gestion de base de données generalisées. Projet TIGRE" Thèse de Docteur Ingénieur Juin 1984 INPG Grenoble

[SNO 85] R.Snodgrass, I.Ahn "A Taxonomy of Time in Databases" Proceedings of the ACM-SIGMOD Austin May 1985

[VEL 84] F.Velez "Un modèle et un langage pour les bases de données généralisées" Thèse de Docteur Ingénieur INPG Grenoble Sept 1984

[VEL 85a] F.Velez "La prise en compte des documents structurés dans un SGBD : Aspects modèle, langage et architecture du système" Journées BD Avancées, INRIA St Pièrre de Chartreuse, Mars 1985

[VEL 85b] F.Velez "LAMBDA : An entity-relationship based language for the retrieval of structured documents" Proceedings of the 4th Itern Conf on E-R Approach, Chicago, October 1985