Université de Namur (FUNDP)
Laboratoire d'ingénierie des applications de bases de données
Institut d'Informatique
Rue Grandgagnage, 21
B-5000 Namur
http://www.info.fundp.ac.be/libd

TimeStamp

Understanding
Developing
Processing Temporal databases

Responsable : Jean-Luc Hainaut

Chercheurs : Virginie Detienne, Didier Roland, Thomas Lieutenant

Volume 1

Introduction aux bases de données temporelles

Financé par le Ministère de la Région Wallonne (contrat 9713563)

TimeStamp

Comprendre, développer et exploiter les données temporelles

Volume 1

Introduction aux bases de données temporelles

Janvier 2002

Le projet TimeStamp a été financé par le Ministère de la Région Wallonne (contrat 9713563)

•

Les données temporelles font partie des systèmes d'information de nombreux secteurs industriels et administratifs. Elle décrivent les états passés (données historiques), courants et futurs de l'entreprise. Outre leur gestion classique, elles sont notamment utilisées dans le domaine de l'aide à la décision, leur consultation permettant de livrer les lois d'évolution de certains paramètres de l'organisme, tels que l'évolution des ventes, la rotation des employés,...

Malheureusement, l'introduction de la dimension temporelle dans les bases de données les rendent rapidement très complexes, tant au niveau de la gestion que de l'exploitation. De plus, les technologies des bases de données actuellement utilisées (SQL-2 par exemple) offrent peu de support à la gestion de telles données.

Pour répondre à ces difficultés l'objectif du projet TimeStamp est de présenter une méthodologie et des outils d'aide à la gestion et la consultation des données temporelles.

Le volume d'introduction aux bases de données temporelles décrit les particularités de ce type de données, ainsi que la manière de les structurer et de les gérer. Il démontre également la nécessité d'avoir recours à une méthodologie et à des outils pour faciliter le traitement des données temporelles.

L'article "CASE Tool Support for Temporal Database Design" donne un aperçu de l'utilité des outils de gestion des bases de données temporelles. Les modèles, la méthodologie, et les outils y sont présentés de manière succincte.

Le tutoriel "Introduction pratique aux bases de données temporelles" décrit les caractéristiques des données temporelles et initie le lecteur aux principes des structures de données temporelles et de leur traitement en SQL.

Enfin, les exposés qui ont été présentés à des responsables de bases de données de diverses industries et administrations, décrivent succinctement les particularités des données temporelles et les outils destinés à faciliter leur gestion et leur exploitation.

Table des matières

Volume 1 - Introduction aux bases de données temporelles

Volume 2 - Méthodologie et exploitation des bases de données temporelles

Volume 3 - Documents techniques

Première partie - Rapports techniques de recherche

Les Données bitemporelles

9. Bitemporalisation
10. La bitemporalisation et le futur

Les Structures de données

11. Structures de données bitemporelles - Etudes de cas

Les Transformations

12. Héritage de la dimension temporelle des objets lors de transformations de schémas

La Normalisation

13. Normalisation des schémas conceptuels temporels

Les Associations non fonctionnelles

14. Historiques d'associations non fonctionnelles

Les Modèles

15. Comparaison des modèles temporels de Snodgrass et de TimeStamp

L'Exploitation des données

16. Contribution à la mise au point d'un langage d'accès aux bases de données temporelles - Mémoire de fin d'études

Deuxième partie - Documentations techniques des outils

Les Outils de génération de bases de données temporelles

17. Performances des bases de données temporelles générées par DB-Main
18. Patterns de génération de triggers destinés à la gestion de l'aspect des bases de données

Les Outils d'exploitation de bases de données temporelles

19. Elaboration d'un générateur de scripts SQL de population d'une base de données
20. T-ODBC : Documentation technique
21. Petit plan de test de T-ODBC

TimeStamp

Volume 1 - Introduction aux bases de données temporelles

Introduction générale

1. ***CASE Tool Support for Temporal Database Design* (ER-2001 Conference, Yokohama, November 2001)**

CASE Tool Support for Temporal Database Design

Virginie Detienne, Jean-Luc Hainaut
Institut d'Informatique, University of Namur
rue Grandgagnage, 21 - B-5000 Namur - Belgium
tel: +32 81 724985 - fax: +32 81 724967

Abstract. Current RDBMS technology provides little support for building temporal databases. The paper describes a methodology and a CASE tool that is to help practitioners develop correct and efficient relational data structures. The designer builds a temporal ERA schema that is validated by the tool, then converted into a temporal relational schema. This schema can be transformed into a pure relational schema according to various optimization strategies. Finally, the tool generates an active SQL-92 database that automatically maintain entity and relationship states. In addition, it generates a temporal ODBC driver that encapsulates complex temporal operators such as projection, join and aggregation through a small subset of TSQL2. This API allows programmers to develop complex temporal applications as easily as non temporal ones.

1 Introduction

A wide range of database applications manage time-varying data. Existing database technology currently provides little support for managing such data, and using conventional data models and query languages like SQL-92 to cope with such information is particularly difficult [12]. Developers of database applications can create only ad-hoc solutions that must be reinvented each time a new application is developed.

The scientific community has long been interested in this problem [9]. The research has focused on characterizing the semantics of temporal information and on providing expressive and efficient means to model, store and query temporal data.

A lot of those studies concerned temporal data models and query languages. Dozens of extended relational data models have been proposed [13], while about 40 temporal query languages have been defined, most with their own data model, the most complete certainly being TSQL2 [11]. However, it seems that neither standardization bodies nor DBMS editors are really willing to adopt these proposals, so that the problem of maintaining and querying temporal data in an easy and reliable way remains unsolved.

A handful of temporal DBMS prototypes have been proposed [1], [13]. Attention has been paid to performance issues because selection, join, aggregates and duplicates elimination (coalescing) require sophisticated and time consuming algorithms [2].

Several temporally enhanced Entity-Relationship (ER) models have been developed [6]. UML (Unified Modelling Language) also has been extended with temporal semantics and notation (TUML) [14].

Among those studies, few methodologies and tools support have been proposed for temporal database design.

About this paper

Like for conventional databases, a temporal database design methodology must lead to correct and efficient databases. However, the design of even modest ones can be fairly

complex, hence the need for CASE tools, specially for generating the code of the database. This paper describes a simple methodology and a CASE tool that is to help practitioners develop temporal applications based on SQL-92 technology.

These results are part of the TimeStamp project whose the goal is to provide practitioners with practical tools (models, methods, CASE tools and API) to design, manage and exploit temporal databases through standard technologies, such as C, ODBC and SQL-92. Though the models and the languages used are more simple than those available in the literature, the authors feel that they can help developers in mastering their temporal data.

Sections 2, 3 and 4 introduce the concepts of temporal conceptual, logical and physical models for relational temporal databases that are specific to the TimeStamp approach. Section 5 describes the methodology for temporal database design, and presents a CASE tool that automates the processes defined in the methodology.

2 A Temporal Conceptual Model

The database conceptual schema of an information system is the major document through which the user requirements about an application domain are translated into abstract information structures. When the evolution of the components of this application domain is a part of these requirements, this schema must include temporal aspects. This new dimension increases the complexity of the conceptual model, and makes it more difficult to use and to understand. To alleviate this drawback, we have chosen a simple and intuitive formalism that must improve the reliability of the analysis process. This model brings three advantages, which should be evaluated against its loss of expressive power. First, it has been considered easier to use by developers when the time dimension must be taken into account (for instance, temporal consistency through inheritance mechanism is far from trivial). Secondly, the distance between a conceptual schema and its relational expression is narrower, a quality that is appreciated by programmers. Thirdly, schema expressed in a richer model can be converted without loss into simpler structures through semantics-preserving transformations [8].

Though the concepts of temporal conceptual schemas have been specified for long in the literature, we will describe them very briefly [14], [6].

The conceptual model comprises three main constructs, namely entity types, single-valued atomic attributes and N-ary relationship types. Each construct can be non-temporal or temporal. In the first case, only the current states are of interest, while in the latter case, we want to record past, current and future states. In this presentation, we will address the modelling and processing of historical states, that is the past and current states only[1].

The temporal dimension can be based on valid time (/v), on transaction time (/t) or on both (/b for bitemporal). The instances of a non-temporal *monotonic* entity type (/m) can be created but never deleted, so that the letter enjoys some properties of temporal entity types. A construct is non-temporal, unless it is marked with a temporal tag: /m, /v,

1 Though most of the principles described in this paper can be extended to future states as well, the latter induce some constraints that we do not want to discuss in this paper. For instance, not all data distribution patterns proposed in the physical model can accommodate future states.

/t or /b (Fig. 1). An entity type can have one primary identifier (or key) and some secondary identifiers. A relationship type has 2 or more roles, each of them being taken by an entity type. A role has a cardinality constraint defined by two numbers, the most common values being 0-1, 1-1, 0-N.[2] A non-temporal attribute can be declared *stable*, i.e., non-updatable. The temporal attributes of the time intervals of the states are implicit.

If an entity type is temporal, then, for each entity that existed or still exists, the birth and death instants (if any) are known (valid time), and/or the recording (in the database) and erasing instants (transaction time) are known. This information is implicit and is not part of the attributes of the entity type. If an attribute is temporal, then all the values associated with an entity are known, together with the instants at which each value was (is) active. The instants are from the valid and/or transaction time dimensions according to the time-tag of the attribute. If a relationship type is temporal, then the birth and death instants are known. The two time dimensions are allowed, according to the time-tag.

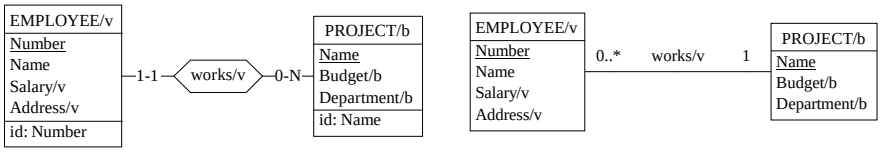


Fig. 1. A conceptual schema. Both ERA (left) and UML (right) notations are provided.

To ensure the consistency of the future temporal database, but also to limit its complexity and to make its physical implementation easier and more efficient, the model imposes some constraints of the valid schemas[3]. A conceptual schema is said to be *consistent* if:

1. the temporal attributes of an entity type have the same time-tag (mixing temporal and non-temporal attribute is allowed);
2. the time-tag of an entity type is the same as that of its temporal attributes; there is no constraints if it has no temporal attributes;
3. each temporal entity type has a primary identifier (or key) made up of mandatory, stable and unrecyclable[4] attributes; there is no constraints on the other identifiers;
4. the entity types that appear in a temporal N-ary relationship type are either temporal or monotonic;
5. the entity type that appears in the [i-1] role[5] of a *one-to-many* temporal relationship type R has the same time-tag as R; *one-to-one* relationship types are constrained as if they were one-to-many;
6. the entity type that appears in the [0-N] role[6] of a *one-to-many* temporal relationship type R has a time-tag compatible (in a sense that translates into valid foreign keys as stated in Sec. 3.4) with that of R; *one-to-one* relationship types are constrained as if they were *one-to-many*;
7. the entity types that appear in the roles of a N-ary relationship type R have a time-tag

2 Though they have the same expressive power in binary relationship types, the ERA cardinality and UML multiplicity have different interpretations.
3 We leave this undemonstrated for space limit.
4 An attribute is unrecyclable if its values cannot be used more than once, even if its parent entity is dead.
5 i.e., the domain of the function that R expresses;
6 i.e., the range of the function that R expresses;

that is *compatible* (same meaning as above) with that of R.

A conceptual schema that meets all these conditions can be translated into a *consistent relational* schema, as defined in Sec. 3.

3 A temporal Relational Logical Model

This model defines the interface used by the programmer, that is, the data structures, the operators and the programming interface. We could have adopted a more general temporal relational model, such as that from [12]. However, the fact that the tables derive from a consistent conceptual schema induces specific properties that will simplify the implementation and (hopefully) the mental model of the programmer.

This model comprises tables, columns, primary keys, secondary (i.e., candidate, non-primary) keys and foreign keys. These constructs can be temporal (except for primary keys) or non-temporal, according to the same time-tag as those used in the conceptual model. A table that implements an entity type is called an *entity table*, while a table that translates a N-ary or *many-to-many* relationship type is called an *relationship table*[7]. Only entity tables can be monotonic.

The structure of temporal tables is as usual [12]:

1. valid-time tables have two additional timestamp columns, called Vstart and Vend, such that each row describes a fact (such as a state of an entity or relationship) that was (or is) valid in the application domain during the interval [Vstart,Vend);[8] these new columns can be explicitly updated by users according to limited rules;
2. any transaction-time table has two timestamp columns Tstart and Tend, that define the interval during which the fact was (is) recorded in the database; these columns cannot be updated by users;
3. in a bitemporal table, these four columns are present.

An *entity table* comprises three kinds of columns, namely the entity identifier, which forms the primary key of the set of current states of the entities, the timestamp columns Vstart, Vend, Tstart, Tend and the other columns, called attribute columns, that can be temporal or not. A *relationship table* is similarly structured: the relationship identifier, made up of the primary keys of the participating entity types, the timestamp columns and the attribute columns, if any. For simplicity, we ignore the latter in this paper.

For each temporal dimension, the right bound of the interval of the current state is set to the infinite future, represented by a valid timestamp, far in the future (noted ∞ here).

The entity and relationship tables together form the set of *database tables*. However, other tables can be built and used, mainly by derivation from database tables. These tables may not enjoy the consistency properties that will be described, and therefore will require special care when used with database tables.

3.1 Temporal State Properties

The base tables of the database derive from the conceptual schema, so that not all data

---

7 Though complex mapping rules can be used to translate entity types and relationship types, those that we adopt in the methodology are sufficiently simple to make these concept valid.

8 `[i,j)` is the standard temporal notation for a left-closed, right-open. Also noted `[i,j[`.

patterns are allowed. In this sense, the model is a subset of those proposed in the literature, e.g., in [12].

Let us first define their properties for temporal tables *with one dimension only.* The timestamp columns are simply called Start and End, since both kinds of time enjoy the same properties.

The granularity of the valid time clock is such that no two state changes can occur during the same clock tick for any given entity or relationship[9]. Similarly, no two states of the same entity/relationship can be recorded during the same transaction time clock tick. This gives a first property: for any state s, s.Start < s.End.

In a *temporal entity table*, be it transaction or valid time, all the rows related to the same entity form a *continuous history*, that is, each row, or state s1, but the last one, has a next state s2, such that s1.End = s2.Start. This property derives from the fact that, at each instant of its life, an entity is in one and only one state. Thirdly, any two states (s1,s2) such that s1.End = s2.Start (i.e., that are *consecutive*) must be different, that is, the values of at least one attribute column are distinct.

In a bitemporal entity table, each transaction time *snapshot*, i.e., the state of the table known as current at a given instant T, must be a valid time entity table that meets the properties described above.

In a *temporal relationship table*, a row tells that the participating entities were (are) linked during the interval [Start,End). For any two rows r1 and r2 defined on the same set of entities, either r1.End < r2.Start or r2.End < r1.Start hold.

3.2 Consistency State of a Table

The model defines four consistency states: a table can be corrupted, correct, normalized and fully normalized. In these definitions, two rows are said *value-equivalent* if they have the same values for all the columns (timestamp columns excluded).

A entity table is **corrupted** if, for some entity E and for some time point, it records at least two different states, i.e., states whose values differ for at least one attribute column. It is **correct** if, for any two states of the same entity that overlap, the values of the attribute columns are the same. It is **normalized** if, for any entity, its states do not overlap. It is **fully normalized** if any entity has a state for each instant of its life (continuous history). All entity tables must be fully normalized. Derived tables, i.e., tables resulting from the application of DML operators, that represent some part of the history of a database object must be at least correct.

A relationship table is **corrupted** if, for some value of the relationship identifier, there exist at least two non value-equivalent rows whose temporal interval overlap. Such a table is **correct** otherwise.

This classification is irrelevant for plain temporal table that are neither entity or relationship tables. In general, such tables will be said *non corrupted.*

3.3 Candidate Keys

If EI is the primary identifier of the entity/relationship type described by the valid time table T, then {EI,Vstart} is the primary key of T. {EI,Vend} is a secondary key, as well

9 When this property is not ensured by the natural time(s), techniques based on an abstract time line can be used. This point is out of the scope of this paper.

as {ESI,Vstart} and {ESI,Vend}, where ESI is any secondary identifier of the entity/relationship type. For simplicity these candidate keys will not be represented in the logical schema, though they will be maintained in the database by the triggers of the physical schema. Similarly, the primary key of the transaction time table T is {EI,Tstart}. Its secondary keys are derived in the same way as in valid time tables. Finally, the primary key of bitemporal table T is conventionally {EI,Vstart,Tstart}. Note that these definitions are valid for database tables only and not necessarily for derived tables.

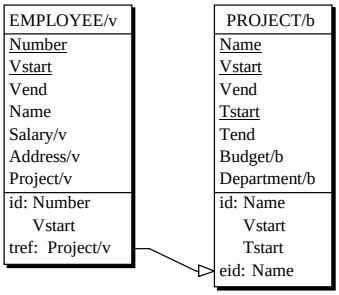


Fig. 2. A logical relational schema showing the transaction/valid timestamp columns. Tables and columns can have time-tags. EMPLOYEE.Project is a temporal foreign key to PROJECT, whose column Name is the entity identifier (eid).

3.4 Foreign Keys

Let us first define this concept for non bitemporal tables. A column or a set of columns FK of a source table S is a *foreign key* to the target table T, with the primary key (PK,Start), if, for each state s of S where FK is not null, and for each time point p in [s.Start,s.End), there exists at least one state t in T such that s.FK=t.PK and t.Start ≤ p < t.End. This property, which must be checked when tuples are inserted, updated and deleted, is complex and expensive to evaluate for temporal databases in which the target tables are only required to be normalized or even correct [12].

In this model, a foreign key belongs to an entity or relationship table, but the target table always is an entity table. Considering that entity tables are fully normalized by construction (*no gap, no overlap*), the definition degenerates into a property that is more straightforward and easier (i.e., *cheaper*) to check. Let us consider the source table S(..., Start, End, ..., FK) and the target table T(PK, Start, End, ...). S.FK is a temporal foreign key to S iff,

$\forall s \in S, \exists t1, t2 \in T$:

$$t1.PK = t2.PK = s.FK \wedge t1.Start \leq s.Start < t1.End \wedge t2.Start < s.End \leq t2.End$$

For bitemporal databases, this definition must be valid for each snapshot.

The foreign key, its source table and its target table need not have the same time-tag. For instance, a non temporal foreign key (in a temporal or non temporal source table) can reference a non temporal (including monotonic) or a temporal table; a valid time foreign key can reference monotonic, valid time and bitemporal target table. The allowed pairs of source and target time-tags define the compatibility rules, that will not be developed further in this paper.

3.5 Operators

The semantics of the usual relational operators have been extended to temporal tables,

while new ones have been designed to cope with specific problems of temporal data [11]. This model includes four extraction operators, namely selection, projection, join, aggregation, and normalization transformations.

Temporal projection. This operator returns, for any correct source entity table S and for any subset A of its columns, a temporal table in which only the values of the columns in A are kept. If the set of the projection columns includes the entity identifier, the result is a normalized entity table. Conceptually, the temporal projection can be perceived as a standard projection followed by the merging (or *coalescing*) of the rows that have the same values of the non-temporal columns, and that either overlap or are consecutive.

Temporal selection. This operator returns the rows that meet some selection predicate in a correct source table. The selection can involve the temporal columns, the other columns, or both.

Temporal join. Considering two correct temporal source tables S1 and S2, and a predicate P, this operator returns, for each couple of rows (s1, s2) from S1xS2 such that P is true and the temporal interval i1 of s1 and i2 of s2 overlap, a row made up of the column values of both source rows and whose temporal interval is the intersection of i1 and i2. The result is a correct temporal table.

Temporal aggregation. Due to the great variety of aggregation queries, the process has been decomposed into four steps that are easy to encapsulate. Let us consider a correct entity table T with one time dimension (the reasoning is similar for other tables). The query class coped with has the general form : select A, f(B) from T group by A, where f is any aggregation function. First, a normalized state table minT is derived by collecting, for each value of A, the smallest intervals during which this value appears in T[10]. This table is joined with T to augment it with the value s of B, giving the correct table minTval. Then, the aggregation is computed through the query select select A, f(B) from T group by A. Finally the result is coalesced. In particular, this technique provides an easy way to compute temporal series (in this case, minT is a mere calendar).

Temporal normalization. This family of operators augment the consistency state (Sec. 3.2) of a correct table. By merging the value-equivalent overlapping or consecutive states, they produce normalized tables (no overlap), and by inserting the missing states of a non-fully normalized table, they produce a continuous history (no gap, no overlap).

3.6 The DML Interface

Though temporal operators can be expressed in pure SQL-92, their expression generally is complex and resource consuming [12], so that providing the programmer with a simple and efficient API to manipulate temporal data is more than a necessity. Developing a complete engine that translates temporal SQL queries would have been unrealistic, so that we chose to implement a (very) small subset of a variant of TSQL2 [11], called miniTSQL, through which the complex operators, such as project, join and aggregate can be specified in a natural way and executed[11]. Combining explicit SQL-92 queries

---

10 If *T(E,Start,End,A,..)* has instances *{(e1,20,45,a1,..), (e2,30,50,a2,..), (e3,35,55,a1,..)}*, this step generates the states *{(a1,20, 35), (a1,35,45), (a1,45,55), (a2,30,50)}*.

11 miniTSQL and Temporal ODBC, as well as a procedural solution to bitemporal coalescing have been defined and prototyped by Olivier Ramlot (*Contribution à la mise au point d'un langage d'accès aux bases de données temporelles*, Mémoire présenté en vue de l'obtention du grade de Maître en Informatique, Université de Namur, Belgique, 2000).

with miniTSQL statements allows programmers to write complex scripts with reasonable effort. The API is a variant of ODBC, through which miniTSQL queries can be executed. The driver performs query analysis and interpretation based on a small repository that describes the database structures and their physical implementation (Sec. 4).

The following program fragment displays the name and salary of the employees of project BIOTECH as on valid time 35. The SQL query uses a temporal projection (including coalescing) and a temporal selection. It replaces about 100 lines of complex code that would have been necessary if operating directly on the tables of Fig. 2.

```
char name[50], salary [20], output[100];
sdword cbname, cbsalary ;
. . .; rc=SQLConnect(hdbc,...); ...
rc=TSQLExecDirect(hdbc,hstmt,"select snapshot Name, Salary
                              from  EMPLOYEE
                              where valid(EMPLOYEE) contains
                                    timepoint'35'
                             and   Project = 'BIOTECH'",type);
rc=SQLBindCol(hstmt,1,SQL_C_CHAR,name,50,cbname);
rc=SQLBindCol(hstmt,2,SQL_C_CHAR,salary,20,cbsalary);
do { rc = SQLFetch(hstmt);
     if(rc == SQL_NO_DATA)  break;
     strcpy(output,"Name: ") ; strcat(output,name);
     strcat(output,"Salary: "); strcat(output,salary);
     MessageBox(output,"TUPLE",MB_OK);
  }while(rc!= SQL_NO_DATA);
...; rc=SQLDisconnect(hdbc); ...
```

To make the programmer's work easier and more reliable, the modification statements insert, delete and update apply on a view that hides the transaction temporal columns Tstart and Tend. More specifically, this view returns, respectively, (1) the current states of a transaction time table, (2) all the states of a valid time table and (3) the valid history of a bitemporal table.

4 A Temporal Relational Physical Model

The physical schema describes the data structures that actually are implemented in SQL-92. When compared with the logical schema, the physical schema introduces four implementation features.

Data distribution. The logical model represents the evolution of an entity/relationship set as a single table where each row represents a state of an entity/relationship. At the physical level, the states and the rows can be distributed, split and duplicated in order to gain better space occupation and/or improved performance. The first rule concern the distribution and duplication of states.

A bitemporal historical table includes valid current states (Tend=∞ ∧ Vend=∞), valid past states (Tend=∞ ∧ Vend<∞) and invalid states (Tend<∞). This suggest various patterns of distribution, which each has advantages and drawbacks as far as performance and complexity are concerned: all the states in the same table, all the states in the same table + a copy of the valid current states in another table, the valid current states in a table + all the other states in another table, the valid states in a table + the invalid states in another table, to mention the most important. Tables with one dimension only can be organized in a similar way. Should future states be included, they would have to be stored in the same table as the current states.

A logical table comprises all the columns that implement the entity attributes and the one-to-many relationship types (as foreign keys), be they temporal or not. Storing rows in a single table may induce much redundancy[12], so that distributing the columns into temporally homogeneous tables can decrease it dramatically. Three patterns are of particular importance: all the columns are collected in a single table (as in the logical schema), the non temporal columns form a table while a second table collects the temporal columns, the non temporal columns form a table while each temporal column forms a specific table. Other splitting patterns can be useful, that mainly pertain to the temporal normalization domain [16].

Indexing. As in conventional databases, indexes will be defined to improve the access time for the most frequent operations. Besides the primary keys, foreign keys, arguments of group by and order by clauses, frequent selection criteria, are candidate for indexing. Some temporal operators can be accelerated by using auxiliary structures. For instance, an entity table that stores the life span of each entity can be used to quickly check referential constraints in a bitemporal database. A *pre-join table* TS, that stores, for joinable tables T and S, the couples (s,t) of rows from T and S that overlap, can be use to replace the temporal join T*S by the standard join T*TS*S, which generally is faster.

Automatic data management. Managing a physical temporal database is particular complex, so that its automation must be pushed as far as possible. The approach we have chosen consists in implementing the logical database, as described in Sec. 3, as an *active database* whose active components are responsible for guaranteeing the consistency properties of the data and controlling the logical/physical mapping. Each logical table is given a set of triggers that control the insert, delete and update operations by checking their validity and by propagating them among the physical tables.

For instance, the statement,

```
insert into EMPLOYEE(Number,Vstart,Salary,Address,Project)
values(:N,:VS,:SAL;ADD,:PRO);
```

triggers a procedure that performs the following operations, that can span several hundreds of lines of code for complex tables:

1. check: no current state where NUMBER=:N already exists (uniqueness); *2. check*: no past states, where NUMBER=:N already exist (non recyclability); *3. check*: Project=:PRO is a valid temporal foreign key (referential integrity); *4. check*: :VS is a past or current timepoint (pure history); *5. execute*: Vend is set to ∞ (current state); *6. execute*: the state is stored in the physical table(s) (logical/physical mapping); *7. execute*: the auxiliary structures are updated (logical/physical mapping).

5 Methodology and CASE Support for Temporal Databases

Despite the important research area of temporal databases, few methodologies for temporal databases design have been developed.

Some mappings from temporally extended ER models to relational model have been proposed [5], [7], [10], [15]. The models of [5], [7], [15] support only valid time, while

12 The change of a single column in a row triggers the insertion of a new state, in which all the unchanged columns are merely copied.

the TempEER model [10] supports both valid time and transaction time of data. The TIMEER model [7] captures aspects such as the life span, valid time and transaction time of data too. A set of 31 constraints is defined to enforce the ER-specified time-related semantics in the relational context.

Those mappings allow to configure temporal data in only one way. However, we saw in Sec. 4 that it was possible to distribute data differently. Each data configuration has advantages and drawbacks, and designers must choose the distribution that corresponds best to the needs of their application. So, it would be interesting to allow different configurations, while hiding their complexity to the programmer.

Most often, the mappings are not supported by tools. However, the design of even modest temporal databases can prove very complex so that it cannot, most of the time, be carried out without the support of CASE tools. To mention one example only, a single *update trigger* controlling a bitemporal table with two foreign keys and referenced by another one, and that supports evolution and correction modifications, can be made up of *more than 500 lines of complex code.*

We will propose a solution to this problem in terms of the TimeStamp methodology for temporal databases design and of an extension of the CASE tool DB-Main that supports it. The products of the methodology are the temporal conceptual, logical and physical schemas, as well as the code necessary to manage and exploit the corresponding temporal relational database, as described in Sections 2, 3 and 4. The CASE tool allows to execute automatically all the processes of the methodology, including code generation, according to three different physical data configurations.

In this section, we first describe the conventional methodology, then we present its extension to temporal data together with the CASE tool DB-Main.

5.1 Conventional Methodology

Database design is usually carried out in three main phases: conceptual design (or analysis), logical design and physical design.

Conceptual design consists in expressing the concepts of the application domain into a high-level abstract model that is independent of the particular data model of the target DBMS. This expression is called the conceptual schema. The goal of logical design is to translate the conceptual schema into a structure adapted to the data model of the DBMS, namely the logical schema. In short, the logical schema is all the programmer have to know, and nothing more, in order to develop programs on the database. Physical design includes choosing technical implementation (e.g., indexes, data storage and clusters) and setting physical parameters.

These methodologies are now mastered and can be considered a integral part of the culture of developers.

5.2 Extension of the Methodology to Temporal Databases and CASE Tool DB-Main

The methodology we propose is quite similar to the conventional one. Addressing the temporal dimension of data, merely adds new aspects to the standard processes (Fig. 3).

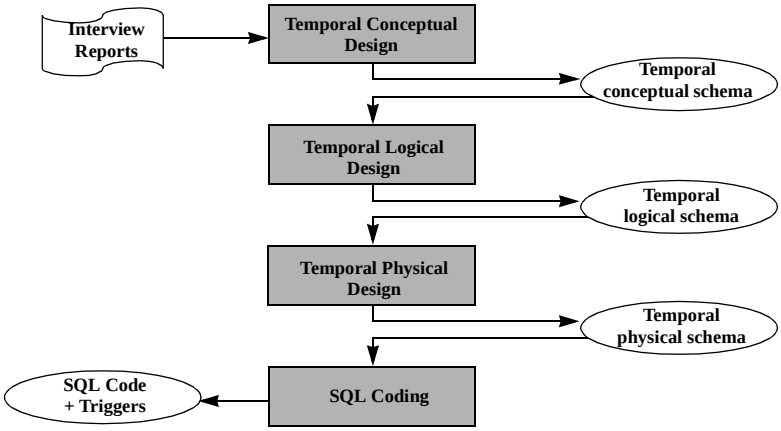


Fig. 3. Temporal Database Design Methodology. The information source is symbolically represented by *Interview reports.*

The tool that is being developed in the TimeStamp project is built as a plug-in of the DB-MAIN generic CASE platform [3]. It supports all the processes of the TimeStamp methodology, including code generation. In this section, we describe the different steps of the methodology and illustrate the main aspects of the tool through the processing of an example.

5.3 Conceptual Design

Temporal conceptual design is made up of three steps: non temporal analysis, temporal tagging and normalization.

Analysis and Temporal Tagging. In this first step, a non-temporal conceptual schema is built according to any standard methodology. Then, the schema objects that are to be temporally dimensioned are marked with the desire time-tag, namely transaction, valid, bitemporal or monotonic, as shown in Fig. 4.

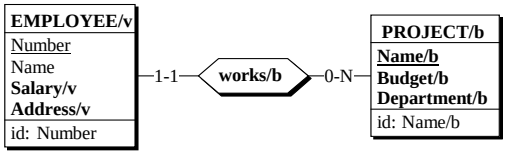


Fig. 4. Example of raw (un-normalized) temporal conceptual schema. An entity primary identifier is declared by the clause **id**. The tag /t, /v, /b, /m shows that the entity type, the relationship type or the attribute is timestamped with transaction time (t), valid time (v), both (b=bitemporal) or is monotonic (/m, for entity types only).

CASE support. DB-MAIN includes a graphical schema editor that allows designers to define their schemas according to various styles (UML, ERA, OO, etc.). A special property (T_HistoryType) is attached to each data structure to define its temporal characteristics. The designer chooses the way object names are tagged to show this property graphically (Fig. 4 and Fig. 5)

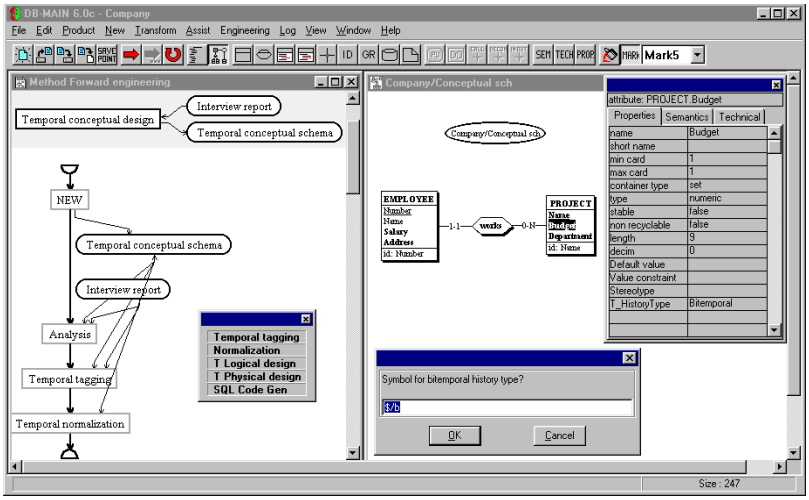


Fig. 5. .*Temporal conceptual design*. The main window is that of DB-MAIN. The left hand side window shows the structure of the temporal conceptual design process. The toolbar located in this window is specific to temporal database design. It allows the automatically execution of the main four processes of the methodology. The designer draws the conceptual schema in the right side windows. The right side box shows the properties of the selected object *Budget*. The property *T_HistoryType* defines the type of time of the object (*Bitemporal*). The small box below permits to choose a suffix or a prefix to add automatically to the name of the temporal objects (represented by *$*). To tag objects, the designer selects a set of objects, then chooses the corresponding time-tag.

Normalization. Though the concept of normalized conceptual schema is well defined, introducing the time dimension induces new criteria of normalization. In short, a temporally normalized conceptual schema satisfies the consistency rules defined in Sec. 2.

As an example, the schema of Fig. 4 violates rule 5: the relationship type works is *bitemporal* while its 1-1 role is *valid-time* (this will introduce a temporal heterogeneity in the future relational table EMPLOYEE). To fix this problem, we can either mark *works* as valid time (Fig. 6).

The primary identifier of PROJECT, namely *Name*, is marked as temporal, violating rule 3. Therefore, we create a stable and non recyclable technical primary identifier Code, while *Name* becomes a secondary identifier (Fig. 6).This schema now meets all the normalization criteria.

CASE Support. The tool checks the properties that a conceptual schema must satisfy. The normalization rules that are violated are reported (Fig. 7).

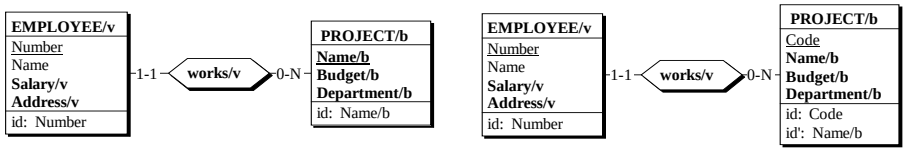


Fig. 6. Normalizing temporal relationship type *works* (left) and making entity primary identifiers non temporal, while preserving the origin uniqueness constraint (rigth).

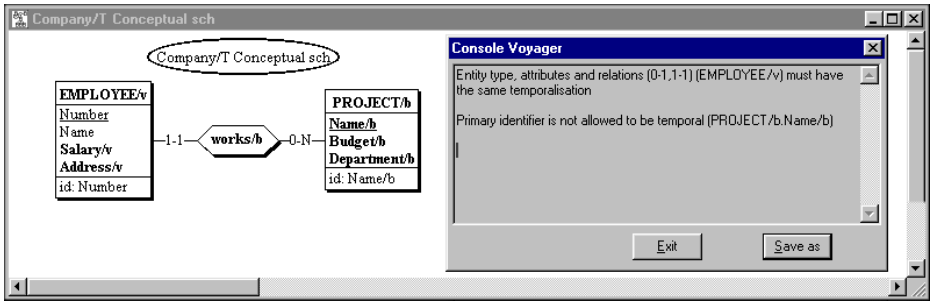


Fig. 7. Temporal normalization tool. The checked schema is in the window. The box on the rigth cites the violated rules.

5.4 Logical Design

The temporal logical design phase consists in translating the conceptual constructs into relational structures and in adding the timestamp columns Vstart, Vend, Tstart and Tend as needed. Regarding the translation process, though sophisticated rules can be designed, we will adopt very simple mapping rules. According to them, an entity type is represented by a table, an attribute by a column, a many-to-many or N-ary relationship type into a table and foreign keys and a one-to-many relationship type by a mere foreign key.

The time tag of a relational construct is inherited from the conceptual object it derives from (Fig. 8). The temporal columns are then added: Vstart and Vend for the valid time and bitemporal tables, and Tstart and Tend for the transaction time and bitemporal tables. The primary keys are defined: (EI,Vstart), (EI,Tstart) and (EI,Vstart,Tstart) for respectively valid time, transaction time and bitemporal tables of entity tables. Similar rules applies for relationship tables. Where needed, the foreign keys are made temporal (Fig. 8).

As far as data management is concerned (through insert, delete, update statements), users can only manage current histories. They work then on a view that has the same configuration as the relational schema but that contains only the current histories, that is to say, all the states of a valid time table, the current states (Tend=∞) of a transaction time table, and the valid states (Tend=∞) of a bitemporal table.

CASE Support. The tool automatically transforms conceptual structures into relational constructions including inherited temporal tags. It adds the timestamp columns and defines the temporal foreign keys (Fig. 8).

5.5 Physical Design

During the temporal physical design, operational and performance issues are considered. Since no specialized technologies can be relied on, we have to stick to pure SQL-92 data structures. Three optimization techniques are proposed.

1. *Table partitioning*. As briefly discussed in Sec. 4, states and columns can be distributed in different tables to improve the execution time of selected operations. The schema of Fig. 9 shows an example of physical schema in which the current states of

each logical table have been duplicated into the specific tables C_EMPLOYEE and C_PROJECT.

2. *System index*. Standard index must be defined in order to support the most common temporal and non-temporal operations. They are described for mono-temporal tables only, to simplify the discussion. A primary key index <EI,Start> supports (1) entity history and single state extraction, (2) projection and coalescing that include the entity primary identifier, (3) FK-to-PK temporal joins. An index on a temporal foreign key <FK,Start> supports PK-to-FK joins and FK selection. Extracting the current state of an entity can be improved by a <EI,End> index, though segregating current states in a specific table will generally be more efficient. Selecting the states that fall in a time interval will profit from an index on <Start,EI>.
3. *Auxiliary structures*. Besides standard index, additional technical tables can be built to accelerate such operations as temporal aggregation, temporal join or projections, as discussed in Sec. 4.

CASE Support. A state data distribution strategy must be chosen. At the present time, three predefined strategies are available, where temporal and non-temporal columns are all grouped in the same table:

1. all the states are in the same table;
2. a table contains the current states and another table the past and invalid states;
3. all the states are gouped in a same table and the current states are duplicated in a specific table (Fig. 9).

The temporal constraints known by the programmers at the logical level must be adapted for each of the three data configurations at the physical level. The expression of the constraints at the physical level remains hidden from the programmers, and those rules will be automatically managed by the triggers of the database.

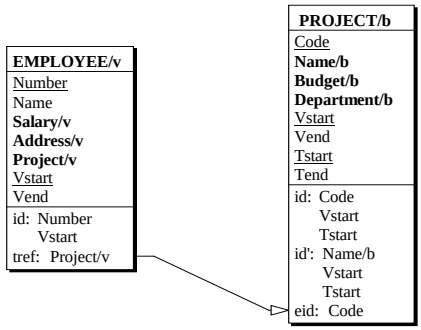


Fig. 8. Temporal logical design. The relationship types have been transformed into foreign keys and the temporal columns have been added. The clause **tref** symbolizes a temporal foreign key (*temporal reference*). The arc indicates the target candidate key, which is the entity identifier (**eid**).

SQL Code Generation. Managing and exploiting temporal data involves a high programming overhead. Therefore, it is important to relieve programmers from the burden of writing this code him/herself. The design of the database must include the writing all the technical procedures, particularly the *triggers*, that manage the tables and keep them in a consistent state.

CASE Support. A tool generates automatically the SQL code that implements the physical schema. This code can be processed by Oracle 8 and includes the definitions of the tables, views, indexes and triggers necessary to manage the temporal data, according to the physical parameters chosen by the developer. The generation of the temporal ODBC drivers, that implement the stratum architecture, still is under development.

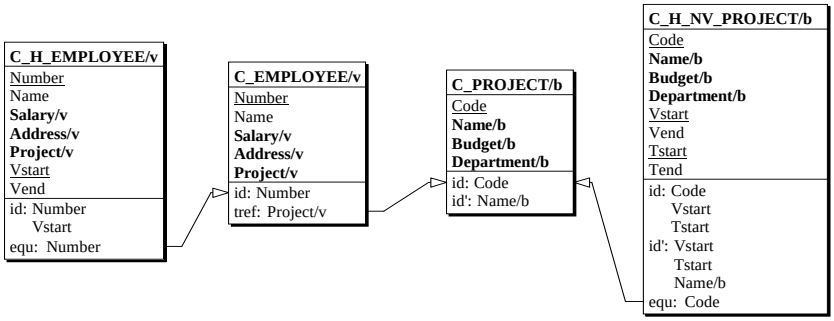


Fig. 9. Physical schema. The clause **equ** symbolizes an *equality* constraint, that combines a foreign key (ref) with an inverse inclusion constraint. For each logical table, a complete history table is maintained, together with a new table that includes the current states.

6 Conclusion

The goal of the TimeStamp project, which started in 1997, was to make, as much as possible, the research results in temporal databases available to practitioners, i.e., *standard* developers and programmers. Considering the richness and the complexity of the concepts of this domain, most of our effort was devoted to simplify them, while retaining enough power and flexibility for making them usable in practical situations.

The simplification has addressed two directions. First, the concepts have been reduced in order to make them easy to understand and to teach. The simplified temporal conceptual model induces a reduced temporal relational logical model, which in turn implies simple and efficient data management techniques. In addition, the methodology we propose is a slight extension of widespread approaches. Secondly, all the burden of developing temporal databases has been taken in charge by a CASE tool and an API has been defined and implemented to make the programming process easier and, more important, more reliable.

Despite this effort, mastering temporal database concepts still is a challenging task, as we experienced when teaching them to practitioners. In particular, understanding bitemporal databases and their dynamic behavior has proved very difficult, and often out of the competence of many ordinary programmers. Since the problem lies in the interpretation of bitemporal data, it cannot be completely solved by merely automating the development processes. Therefore, we consider that education is a major aspect of diffusing temporal database principles, at least as important as developing automated tools and API.

Several questions and points remain unsolved: optimized physical design in the context of the stratum architecture, migration of legacy data to temporal database and coping with schema evolution [4].

7 Credits

Jeff Wijsen kindly reviewed our results and made several important suggestions to improve the models. Babis Theodoulidis and his staff, hosted O. Ramlot during his stay in UMIST, Manchester. Their comments were essential for the definition of the temporal ODBC interface. Ramez Elmasri, who hosted one of our students too, helped him to understand and master physical implementation of temporal databases. Thanks to them.

References

1.BÖHLEN M., *Temporal Database System Implementations*, ACM SIGMOD Record, 24(4):53-60, December, 1995.

2.BÖHLEN M., *Coalescing in Temporal Databases*, Proc. of the 22nd VLDB Conference, Bombay, India, 1996.

3.DB-MAIN, http://www.db-main.be

4.ELMASRI R., WEI H., *Study and Comparison of Schema Versioning and Database Conversion Techniques for Bi-temporal Databases*, Proceedings of the Sixth International Workshop on Temporal Representation and Reasoning (TIME-99), Orlando, Florida, May 1999, IEEE Computer Society.

5.FERG S., *Modeling the Time Dimension in an Entity-Relationship Diagram*, Proceedings of the 4th International Confernce on the Entity-Relationship Approach, pages 280-286, Siver Spring, MD, 1985.

6.GREGERSEN H., JENSEN C. S., MARK L., *Evaluating Temporally Extended ER Models*, Proceedings of the Second CAISE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in System Analysis and Design, 12p, K. Siau, Y. Wand, J. Parsons, editors, Barcelona, Spain, June 1997.

7.GREGERSEN H., MARK L., JENSEN C. S., *Mapping Temporal ER Diagrams to Relational Schemas*, Technical Report TR-39, Aalborg University, Department of Mathematics and Computer Science, December 1998.

8.HAINAUT, J.-L., ROLAND, D., HICK, J.-M., HENRARD, J., ENGLEBERT, V., Database Reverse Engineering: from Requirements to CARE tools, *Journal of Automated Software Engineering*, Vol. 3, No. 2, 1996

9.JENSEN C., SNODGRASS R., *Temporal Data Management*, Technical Report TR-17, TIMECENTER, 1997.

10.LAI V.S., KUILBOER J-P., GUYNES J. L., *Temporal Databases : Model Design and Commercialization Prospects*, DATA BASE, 25(3), 1994.

11.SNODGRASS R., *The TSQL2 Temporal Query Language*, Kluwer Academic Publishers, Massachussetts, USA, 1995.

12.SNODGRASS R., *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann Publishers, USA, 2000.

13.STEINER A., *A Generalisation approach to Temporal Data Models and their Implementations*, Thesis of the Swiss Federal Institute of Technology for the degree of Doctor of Technical Sciences, Zürich, 1998.

14.SVINTERIKOU M., THEODOULIDIS C., *The Temporal Unified Modelling Language*, Department of Computation, UMIST, United Kingdom, October 1997.

15.THEODOULIDIS C.I., LOUCOPOULOS P., WANGLER B., *A Conceptual Modelling Formalism for Temporal Database Applications*, Information Systems, 16(4), 1991.

16.WIJSEN J., *Temporal FDs on Complex objects*, ACM Transactions on Database Systems, Vol 24., No.1,Pages 127-176, March 1999

TimeStamp

Volume 1 - Introduction aux bases de données temporelles

Tutoriels

2. **Introduction pratique aux bases de données temporelles**

Université de Namur (FUNDP)
Laboratoire d'ingénierie des applications de bases de données
Institut d'Informatique
Rue Grandgagnage, 21
B-5000 Namur
http://www.info.fundp.ac.be/libd

TimeStamp project

Understanding
Developing
Processing Temporal databases

Introduction pratique aux bases de données temporelles

Novembre 2002

Préambule

Ce document a pour ambition d'introduire le lecteur aux principaux aspects des bases de données relationnelles représentant les états successifs d'un domaine d'application en évolution. Plutôt que de présenter les principes théoriques de la problématique des données à référence temporelle, nous avons choisi de décrire de manière progressive les concepts et les procédures au travers d'une petite étude de cas constituant un projet complet.

Le texte comporte quatre parties. Dans la première, nous étudierons les structures permettant de représenter les états successifs d'ensembles d'entités. La deuxième partie sera consacrée à la gestion de ces structures de données en réaction aux événements qui affectent le domaine d'application : une entité naît, change d'état à plusieurs reprises, puis meurt. La troisième partie montre comment interroger une base de données temporelle, et décrit plus en détail les trois opérateurs importants de projection, de jointure et d'agrégation. Dans la quatrième partie, on rassemble les pièces développées dans les parties précédentes de manière à constituer un système cohérent implémentant une petite base de données temmporelle ainsi que ses opérateurs de gestion et d'exploitation.

Les composants du projet sont développés sous la forme d'une base de données active en Interbase 5.6, mais ce choix est indépendant des concepts et des principes qui sont présentés dans ce document.

Ce document est un résultat du projet TimeStamp financé par la Région wallonne (Contrat n° 9713563).

Table des matières

1. Introduction

La plupart des bases de données opérationnelles ou d'aide à la décision comportent des *données à référence temporelle*. Le cas le plus fréquent est celui des *données historiques*, qui renseignent sur les états passés du domaine d'application. Bien souvent, le maintien de données historiques est nécessaire pour des raisons légales, l'entreprise étant obligée de conserver durant plusieurs années certaines informations relatives à la comptabilité ou au personnel par exemple. D'autre part, dans le domaine de l'aide à la décision, les données historiques sont souvent interrogées[1] pour livrer des lois d'évolution de certains paramètres de l'organisme, comme des tendances saisonnières dans les ventes, les déplacements de travailleurs, l'évolution d'une maladie dans le temps et dans l'espace, etc.

Malheureusement, il apparaît rapidement que l'introduction du temps dans une base de données entraîne une complexité considérable tant dans la gestion des données que dans leur exploitation. En effet, malgré le caractère en apparence simple et intuitif de la dimension temporelle, sa prise en compte ne tolère pas les solutions improvisées du type "*il suffit d'ajouter une colonne date*".

Le but de ce document est d'initier le lecteur aux principes des structures de données temporelles et de leur traitement en SQL. Pour ce faire, nous allons développer un outil de gestion et d'exploitation d'une petite base de données historiques constituée de deux tables, décrivant respectivement des projets et les employés affectés à ces projets, ainsi que leur évolution au cours du temps. Les propositions que nous allons élaborer sont aisément généralisables à d'autres domaines d'application.

Etant donné l'ampleur et la complexité du domaine, nous n'aborderons que des problèmes simples, et nous renverrons le lecteur à la littérature spécialisée pour les questions plus pointues.

Les composants techniques seront développés en InterBase 5.6[2]. Ils sont aisément généralisables à d'autres SGBD. Cependant, les contraintes propres à Oracle entraînent une architecture des composants quelque peu différente. On consultera sur ce point les résultats du projet TimeStamp.

2. Description de l'application

On considère un domaine d'application constitué d'*employés affectés à des projets*. Ce domaine est représenté par le schéma conceptuel de la Figure 1 (gauche). Il y correspond le schéma relationnel de droite. Ces schémas indiquent que tout employé est identifié par un code, est caractérisé par un nom et un statut, est domicilié à une

1. Sous la forme d'entrepôts de données par exemple.
2. La version 6 est disponible gratuitement sur le site www.inprise.com.

adresse et est affecté à un projet. Un projet est identifié par son intitulé, est caractérisé par un thème et dispose d'un budget.

On admet que l'état de la base de données est celui de la Figure 2. Les données décrivent deux projets et six employés. Cette base de données permet de répondre à des questions relatives, par exemple, aux *thèmes des projets auxquels sont affectés les employés de Genève* ou au *nombre d'employés de statut P affectés au projet BIOTECH*.

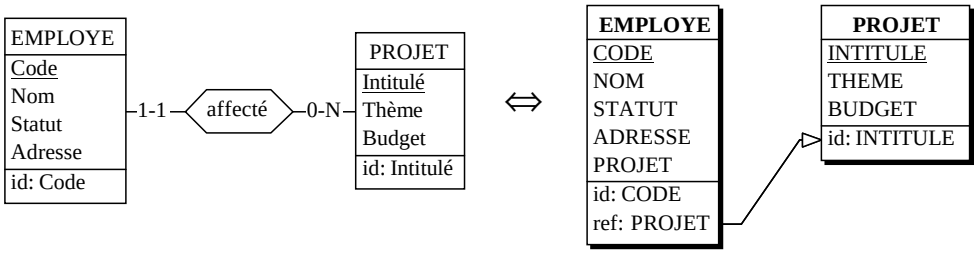


Figure 1 - A gauche, le schéma conceptuel du domaine d'application et à droite sa traduction sous la forme d'un schéma de base de données relationnelle.

Ces données nous renseignent sur l'**état courant** du domaine d'application. Elle ne nous disent rien du passé et de l'évolution des employés et des projets. Des questions essentielles restent sans réponse : *Quand Mercier a-t-il été engagé ? Depuis quand travaille-t-il sur le projet AGRO-2000 ? Combien d'employés travaillaient sur le projet BIOTECH le 1 janvier 2000 ? Quel a été le budget maximum du projet BIOTECH entre 1998 et 2000 ? Quelle a été la carrière des employés qui ont habité au moins un an à Grenoble ?*

PROJET		
INTITULE	**THEME**	**BUDGET**
BIOTECH	Biotechnologie	140.000
AGRO-2000	Amélior. céréales	82.000

EMPLOYE				
CODE	**NOM**	**STATUT**	**ADRESSE**	**PROJET**
C45	Carlier	T	Lille	BIOTECH
G96	Godin	P	Genève	AGRO-2000
D107	Delecourt	P	Genève	BIOTECH
D122	Declercq	P	Charleroi	AGRO-2000
M158	Mercier	P	Paris	BIOTECH
A237	Antoine	P	Grenoble	AGRO-2000

Figure 2 - Etat courant de la base de données. Chaque table décrit l'état actuel d'une population d'entités.

Ce document étudie les composants qu'il serait nécessaire de développer pour que les utilisateurs de cette base de données puissent enregistrer et exploiter les faits rela-

tifs à l'évolution des employés et des projets, c'est-à-dire l'*historique* du domaine d'application. Il répond concrètement aux trois questions suivantes :

I. Comment représenter les données historiques ?

II. Comment gérer des données historiques ?

III. Comment interroger des données historiques ?

La dimension temporelle introduit des concepts et des problèmes nouveaux relativement délicats. Nous serons donc amenés à introduire dans ce chapitre quelques développements spécifiques avant de pouvoir élaborer la solution aux questions posées. En particulier, nous aurons besoins d'algorithmes et de requêtes complexes que nous construirons pas-à-pas de manière à les rendre compréhensibles. Le lecteur aura ainsi la possiblité de les modifier et de les adapter à ses besoins propres.

Partie 1

Comment représenter les données historiques ?

3. Représentation des données historiques

De manière à rendre les matériaux accessibles, nous ferons quelques hypothèses simplificatrices sur lesquelles nous reviendrons brièvement dans la suite. En particulier :

1. La structure du domaine d'application n'évolue pas. En d'autres termes, le schéma de la base de données reste inchangé durant la période considérée du domaine d'application.
2. Nous considérerons un temps abstrait qui se mesure par un entier naturel. Ceci nous évitera de choisir l'unité de temps et d'aborder des problèmes de peu d'intérêt dans un tel document.
3. Nous nous limiterons à la représentation des états passés et présents, en excluant la représentation des états futurs.

3.1 Historique d'un ensemble d'entités

Considérons dans un premier temps la question de l'évolution des projets dans le passé. Le projet BIOTECH, par exemple, n'a pas toujours été dans son état actuel. La Figure 3 montre les événements successifs dont il a été l'objet.

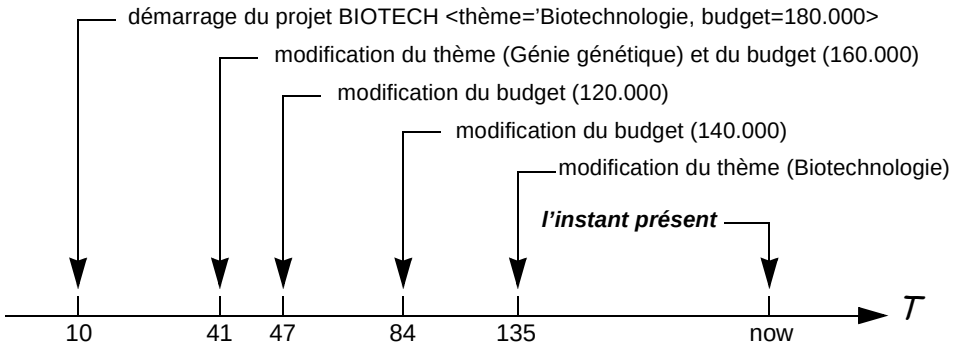


Figure 3 - L'histoire du projet BIOTECH reportée sur l'axe du temps.

Nous choisirons de représenter l'historique d'un projet par la *suite des états successifs* de ce projet, depuis sa création jusqu'à sa disparition. A chaque état sera associée une *période de validité*, définie par son instant initial (debut) et son instant final (fin). Notons que l'instant final est défini comme le premier instant auquel l'état en question a été considéré comme disparu. La période de validité est donc un intervalle *ouvert à droite*, qu'on notera [debut,fin[ou, selon la convention des bases de données temporelles, [debut,fin). Quand on ne connaît pas la fin d'un intervalle, ce qui est le cas des états courants d'un projet, dont on ne sait pas quand ils se termineront, on indiquera le *futur infini* qu'on notera dans ce chapitre par l'entier `9999`.

H_PROJET				
INTITULE	**debut**	**fin**	**THEME**	**BUDGET**
BIOTECH	10	41	Biotechnologie	180.000
BIOTECH	41	47	Génie génétique	160.000
BIOTECH	47	84	Génie génétique	120.000
BIOTECH	84	135	Génie génétique	140.000
BIOTECH	135	9999	Biotechnologie	140.000
AGRO-2000	87	114	Amélior. céréales	65.000
AGRO-2000	114	125	Amélior. céréales	75.000
AGRO-2000	125	9999	Amélior. céréales	82.000
SURVEYOR	65	80	Surv. par satellite	310.000
SURVEYOR	80	101	Surv. par satellite	375.000
SURVEYOR	101	120	Surv. par satellite	345.000

Figure 4 - Historique de l'évolution des projets.

La Figure 4 illustre ces principes. On y apprend par exemple que le projet `BIOTECH` a été créé à l'instant `10`, qu'il est toujours en activité (`fin = 9999`), et qu'il a changé de thème en `41` pour revenir au thème initial en `135`. On y observe aussi l'évolution de son budget. On apprend en outre qu'un troisième projet, `SURVEYOR`, qui a démarré en `65`, est aujourd'hui clôturé.

Nous pouvons faire quelques observations intéressantes.

- A un instant déterminé, une entité est dans un et seul état. On en déduit deux propriétés : (1) deux états d'une même entité sont disjoints[3] et (2) entre les états extrêmes d'une entité, il ne peut y avoir d'états manquants. Si deux états `E1` et `E2` d'une même entité (un même projet) sont tels que `E1.fin = E2.debut`, alors on dira qu'ils sont *jointifs*.
- Pour tout état d'une entité, sauf le dernier, il existe un état jointif suivant du même projet. De même, il existe un état directement jointif précédent, sauf pour le premier. Deux états successifs d'un projet sont *jointifs*.
- Un projet change d'état dès qu'au moins un de ses attributs change de valeur, ou encore à sa création et à sa disparition. Deux états successifs diffèrent donc sur les valeurs d'au moins une colonne d'attribut. On dira que deux états successifs sont *distincts*.
- Tous les instants appartiennent au passé, à l'exception de `9999` qui représente le futur infini. L'instant présent, dont la valeur n'est pas précisée puisqu'elle change en permanence (on la notera `now` lorsqu'il sera nécessaire d'en parler), est supérieur ou égal à toutes les valeurs de `debut`.

La **granularité** du temps est la valeur qu'on assigne à l'unité de temps. C'est aussi la durée de la plus petite période mesurable. Nous ne feront sur la granularité que deux hypothèses :

3. C'est-à-dire que leurs périodes n'ont aucun instant commun.

1. mesurant le temps par des entiers, la granularité est une période de longueur 1;
2. la granularité est choisie de manière que deux événements distincts affectant une même entité sont distants d'au moins une unité sur l'échelle du temps.

3.2 Historiques normalisés

On qualifiera de **normalisé** un historique dans lequel les états d'une même entité sont *disjoints* et *les états jointifs sont distincts* (si deux états sont jointifs, ils sont distincts). Cet historique sera **normalisé et complet** si les états d'une entité sont *disjoints*, *jointifs* (sans trous) et si les *états jointifs sont distincts*. Un historique est **correct** si une entité n'y a, à chaque instant, qu'un seul état. Un historique non normalisé est correct si les états multiples d'une entité à un instant déterminé sont identiques. En d'autres termes, si deux états sont en recouvrement, ils ont les mêmes valeurs pour les colonnes d'attribut. Un historique qui n'est pas correct est considéré comme **corrompu**. En résumé :

Historique d'entités	Définition
Normalisé complet	pour chaque entité : deux états successifs sont jointifs : disjoints, pas de trous (états manquants); deux états jointifs sont distincts;
Normalisé	pour chaque entité : tous les états sont disjoints; deux états jointifs sont distincts; *il peut y exister des trous (états manquants)*;
Correct	pour chaque entité : tous les états non disjoints sont identiques; *il peut y exister des trous (états manquants), des états non disjoints, des états jointifs identiques*;
Corrompu	*il peut y exister des états non disjoints distincts*;

Comme nous le verrons, certaines requêtes de consultation peuvent construire des historiques normalisés incomplets et des historiques non normalisés.

Notons enfin que cette classification ne concerne que les historiques d'entités. Les historiques d'association obéissent à d'autres lois que nous discuterons plus loin.

3.3 Les identifiants

La table H_EMPLOYE décrit les états passés et actuels d'une population d'entités. Nous appellerons *identifiant d'entité* d'une telle table la colonne, ou l'ensemble de colonnes, désignant l'entité à laquelle l'état est associé (ici INTITULE)[4]. Ces colonnes constituent évidemment l'identifiant de la table des états courants (Figure 2). Si cette dernière possède un identifiant primaire et des identifiants secondaires, on parlera, pour la table historique correspondante, de l'*identifiant primaire d'entité* et des *identifiants secondaires d'entité*. Par souci de concision, le terme *identifiant d'entité* désignera l'identifiant primaire d'entité.

Les colonnes INTITULE, THEME et PROJET constituent les *colonnes d'attribut*, c'est-à-dire celles qui représentent les attributs des entités PROJET. Enfin, les colonnes debut et fin forment les *colonnes* ou *bornes temporelles*.

En ce qui concerne les identifiants de la table des états, on admettra les principes suivants :

1. Deux états d'une même entité ont des valeurs de debut distinctes. Il en est de même des valeurs de fin. On déclarera {INTITULE,debut} identifiant primaire de H_PROJET. Pour simplifier, on ne représentera pas explicitement le deuxième identifiant[5], qui est un identifiant secondaire de la table des états.
2. Tout identifiant secondaire d'entité, auquel on ajoute la colonne debut ou la colonne fin, constitue un identifiant secondaire de la table des états.

La gestion cohérente de l'historique d'une population d'entités exige que l'identifiant d'entité jouisse de deux propriétés importantes.

1. Une entité (un projet par exemple) ne peut changer de valeur d'identifiant primaire (INTITULE), à défaut de quoi, il serait impossible de reconstituer son historique. On dira que tout identifiant d'entité doit être **stable** ou non modifiable.
2. D'autre part, l'intitulé d'un projet clôturé ne peut être réutilisé pour représenter un nouveau projet. Si tel était le cas, il y aurait confusion entre les historiques des deux projets. On dira qu'un identifiant d'entité est **non recyclable**.

3.4 Clé étrangère temporelle et intégrité référentielle temporelle

L'évolution des employés se représente d'une manière analogue, comme illustré à la Figure 5. Cet historique introduit un concept nouveau : celui d'*intégrité référentielle temporelle*. L'intégrité référentielle standard, qui traduit dans la Figure 2 le fait qu'*à*

4. Ces propriétés sont valables pour une table représentant des *entités*. Nous ignorerons pour l'instant la question des tables représentant des *associations*, dont certaines caractéristiques peuvent être différentes (voir Section 14.3).
5. Sur base du fait que les algorithmes de gestion garantiront l'unicité des valeurs de cet identifiant.

tout employé correspond un projet, a une interprétation plus complexe pour les données historiques. Il ne suffit plus d'imposer qu'à toute valeur de PROJET d'EMPLOYE doit correspondre une ligne de PROJET de même valeur d'INTITULE, mais encore faut-il que cette propriété soit vraie *pour tout instant passé ou présent de l'historique*.

H_EMPLOYE						
CODE	**debut**	**fin**	**NOM**	**STATUT**	**ADRESSE**	**PROJET**
C45	52	87	Carlier	T	Lille	BIOTECH
C45	87	99	Carlier	T	Lille	AGRO-2000
C45	99	9999	Carlier	T	Lille	BIOTECH
A68	44	65	Albert	P	Mons	BIOTECH
A68	65	95	Albert	P	Mons	SURVEYOR
A68	95	120	Albert	P	Paris	SURVEYOR
G96	12	38	Godin	T	Genève	BIOTECH
G96	38	87	Godin	P	Genève	BIOTECH
G96	87	9999	Godin	P	Genève	AGRO-2000
D107	70	76	Delecourt	T	Grenoble	SURVEYOR
D107	76	97	Delecourt	T	Genève	SURVEYOR
D107	97	111	Delecourt	P	Grenoble	SURVEYOR
D107	111	9999	Delecourt	P	Genève	BIOTECH
D122	47	73	Declercq	P	Mons	BIOTECH
D122	73	120	Declercq	P	Mons	SURVEYOR
D122	120	9999	Declercq	P	Charleroi	AGRO-2000
M158	15	40	Mercier	T	Paris	BIOTECH
M158	40	65	Mercier	P	Paris	BIOTECH
M158	65	108	Mercier	P	Paris	SURVEYOR
M158	108	9999	Mercier	P	Paris	BIOTECH
A237	93	9999	Antoine	P	Grenoble	AGRO-2000
N240	82	93	Nguyen	T	Toulouse	BIOTECH
N240	93	105	Nguyen	T	Grenoble	SURVEYOR
N240	105	131	Nguyen	T	Genève	AGRO-2000

Figure 5 - Historique de l'évolution des employés.

Plus précisément, pour tout état (ligne) E de H_EMPLOYE, il doit exister deux états P1 et P2 (qui peuvent n'en être qu'un seul) de PROJET tels que les deux conditions suivantes sont remplies (Figure 6)[6] :

- P1.INTITULE = P2.INTITULE = E.PROJET
- P1.debut ≤ E.debut et E.fin ≤ P2.fin

6. Ces conditions simplifiées sont valables s'il ne manque pas d'état entre P1 et P2, c'est-à-dire si tous les états d'une entité sont jointifs. Si tel n'est pas le cas, alors il faut ajouter une troisième condition sur l'absence de *trous* entre P1 et P2.

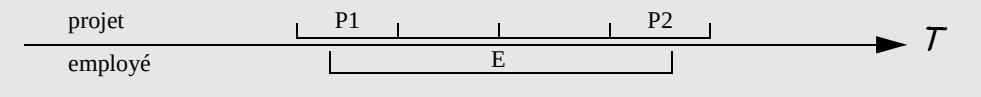


Figure 6 - Intégrité référentielle temporelle.

La Figure 7 représente le schéma relationnel de la base de données historiques. Ce schéma est le siège d'une *contrainte référentielle temporelle*, dérivant de la *clé étrangère temporelle* H_EMPLOYE.(PROJET,debut), notée **tref**[7], vers la table H_PROJET. Les colonnes relatives aux attributs des types d'entités, ou colonnes d'attributs, ont été déclarées obligatoires[8].

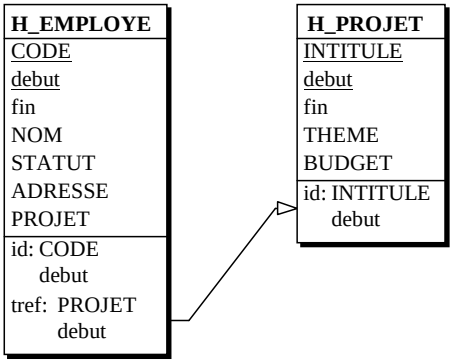


Figure 7 - Schéma de la base de données historiques. H_EMPLOYE.PROJET est une clé étrangère temporelle (**tref**) vers H_PROJET.

3.5 Les tables monotones

La table H_PROJET reprend l'historique des projets, ce qui semble naturel pour garantir l'intégrité référentielle temporelle. Il s'agit d'une condition qu'on peut assouplir. Cette table pourrait être non temporelle, et donc limitée aux états courants, pourvu qu'on n'y supprime jamais aucun projet. Dans ce cas, son identifiant serait INTITULE, et H_EMPLOYE.PROJET serait une clé étrangère ordinaire, non temporelle. On dira d'une table dont on ne supprime jamais aucune ligne et dont l'identifiant primaire est stable qu'elle est *monotone*.

Remarquons que l'intégrité référentielle se contenterait d'une contrainte encore moins stricte : *on peut supprimer une ligne à condition qu'elle ne soit plus référencée.*

7. Dans le projet TimeStamp, on utilisera une autre notation, dans laquelle l'identifiant d'entité est déclaré explicitement.
8. Nous reviendrons sur ce point plus tard (Section 14.2)

3.6 Les deux dimensions temporelles

Nous allons réexaminer le concept d'intervalle temporel de validité, en nous interrogeant sur la signification de ce temps. Toute ligne L de la table H_PROJET représente un état observé P d'une entité PROJET. Il y a donc d'une part **un état P qui existe ou a existé** dans le domaine d'application, entre une date v1 et v2, et d'autre part **une ligne L qui a été enregistrée** dans la table H_PROJET à la date t1 et qui a été supprimée (logiquement, c'est-à-dire considérée comme périmée) à la date t2. Les dates [v1,v2) sont des propriétés du domaine d'application sur lesquelles nous n'avons aucune prise tandis que [t1,t2) ne dépendent que de notre bon vouloir.

Les intervalles [v1,v2) représentent ce qu'on appelle le temps logique ou *valid time*, tandis que les intervalles [t1,t2) indiquent le temps physique ou *transaction time*. Nous représenterons le temps logique par les colonnes debut et fin, et le temps physique par les colonnes enreg et effac (Figure 8).

Si nous ignorons le futur, nous ne pouvons enregistrer que les états observables, passés ou présents. Dans ce cas, on a, pour les états passés et courants considérés comme corrects à l'instant présent (lignes 2 et 3 de la Figure 8) :

$$v1 \leq t1 \text{ et } v2 \leq t2 = 9999$$

La Figure 8 représente un extrait d'une table dans laquelle on aurait enregistré les deux dimensions temporelles de l'évolution des projets. Le temps logique (*valid time*) est représenté par les intervalles [debut,fin) et le temps physique (*transaction time*) par les intervalles [enreg,effac). Cette table est qualifiée de **bitemporelle**, par opposition aux tables telles que H_PROJET et H_EMPLOYE qui sont mono-temporelles.

	HH_PROJET						
	INTITULE	**debut**	**fin**	**enreg**	**effac**	**THEME**	**BUDGET**
	...	...	...	...	...	...	...
1	BIOTECH	10	9999	12	9999	Biotechnologie	180.000
	...	...	...	...	...	...	...

Figure 8 - L'état courant du projet BIOTECH tel qu'on peut l'observer à l'instant présent : on a enregistré en 12 (enreg=12) qu'à partir de 10 (debut=10), le projet BIOTECH est dans l'état <BIOTECH, Biotechnologie,180.000>, et qu'il en est encore ainsi (fin=9999). Cet enregistrement est toujours considéré comme correct (effac=9999).

Partant de la table de la Figure 8, qui représente l'historique du projet BIOTECH tel qu'on peut l'observer en 50 (par exemple) dans la table bitemporelle HH_PROJET, imaginons une série d'événements nouveaux et essayons de les consigner dans cette table. Dans l'intervalle [12,43), on pensait que ce projet était dans l'état <Biotechnologie,180.000> depuis 10 (*ligne 1*). En 43 on a enregistré le fait qu'il avait été remplacé par l'état <Génie génétique,160.000> en 41. On a donc *effacé* l'état précédent (ligne 1, effac=43), on l'a remplacé par un nouvel état ancien (*ligne 2*, fin=41, effec=9999) et on a introduit le nouvel état courant (*ligne 3*). En 50, on considère

que l'historique correct du projet BIOTECH est constitué des lignes 2 et 3. La ligne 1 correspond à un état obsolète, mais nous informe sur ce qu'on croyait correct entre 12 et 43.

	HH_PROJET						
	INTITULE	**debut**	**fin**	**enreg**	**effac**	**THEME**	**BUDGET**
	...	...	...	...	...	...	...
1	BIOTECH	10	9999	12	43	Biotechnologie	180.000
2	BIOTECH	10	41	43	9999	Biotechnologie	180.000
3	BIOTECH	41	9999	43	9999	Génie génétique	160.000
	...	...	...	...	...	...	...

Figure 9 - En 43, on apprend (et on enregistre le fait) que le projet BIOTECH est passé dans l'état <BIOTECH, Génie génétique, 160.000> en 41. La première chose à faire est d'invalider la ligne courante en indiquant qu'elle a été considérée comme correcte jusqu'en 43, mais qu'elle ne l'est plus à partir de ce moment et qu'on l'a donc *effacée* à cette date (effac=43). On la remplace par un état identique, mais qui a été valide entre 10 et 41, qui a été enregistré en 43 et qu'on considère toujours comme étant correct (effac=9999). On enregistre enfin, en 43 (enreg=43) l'état courant (fin=9999), valide depuis 41 (debut=41) et toujours considéré comme correct actuellement, en 50 (fin=9999).

L'historique considéré actuellement comme correct peut être reconstitué aisément par la requête suivante, qui extrait les états passés et courants considérés aujourd'hui comme corrects :

```
select INTITULE,debut,fin,THEME,BUDGET
from   H_PROJET
where  effac = 9999
```

Une table bitemporelle ouvre un espace d'application plus important que celui qu'on vient d'évoquer. C'est ainsi qu'elle permet de représenter non seulement l'évolution des entités, mais aussi les éventuelles corrections des états déjà enregistrés. Nous en reparlerons dans une section ultérieure.

Pour ce qui nous concerne dans l'immédiat, nous limiterons la discussion aux bases de données mono-temporelles. Quant à la question de savoir laquelle des deux dimensions nous adopterons, nous la laisserons dans l'ombre, car la plupart des raisonnements, des requêtes et des procédures en sont indépendants. En particulier, nous continuerons à appeler debut et fin les colonnes temporelles, quelle que soit leur nature. Ce n'est que dans les sections 4 et 5, consacrées à la gestion des données temporelles, que nous distinguerons les deux types de temps.

Partie 2

Comment gérer les données historiques ?

Préambule

On conçoit aisément que les opérations de modification des données, qui d'habitude s'expriment simplement au travers des primitives insert, delete et update, deviennent plus complexes quand il s'agit de conserver la trace des états passés. D'autre part, eu égard aux diverses propriétés particulières d'une base de données historiques, il est important que les opérations de modification de ces données garantissent ces propriétés. Si le SGBD peut prendre en charge certaines d'entre elles, d'autres, plus complexes, réclament des vérifications non triviales, comme l'intégrité référentielle temporelle.

Dans ces conditions, il est indispensable d'**automatiser** autant que possible la gestion des données historiques, et de permettre à l'utilisateur (ou au programmeur) de se limiter aux trois primitives insert, delete et update, appliquées non pas aux états historiques, mais aux états courants, c'est-à-dire aux entités concernées. L'utilisateur/programmeur *créera*, *supprimera* et *modifiera* une *entité*, sans s'inquiéter de la gestion des états historiques. Dans ce but, nous définirons pour chaque table historique une vue (dénommée respectivement PROJET et EMPLOYE) qui reconstitue les états courants des entités, et à laquelle s'adresseront les trois primitives de mise à jour. Des *triggers* prendront en charge le contrôle de la validité de ces opérations et la création des états passés correspondants.

Nous étudierons les structures et règles de comportements respectivement pour la gestion du temps physique (Chapitre 4) et pour celle du temps logique (Chapitre 5).

4. Gestion des données historiques (temps physique ou *transaction time*)

Dans cette section, nous considérerons une gestion entièrement automatique des bornes temporelles debut et fin en fonction de l'état de l'horloge de la machine au moment où les opérations sont effectuées.

Nous pouvons déjà à ce stade rédiger le script de définition de cette petite base de données comme suit (Code 1). On y a ajouté deux vues, PROJET et EMPLOYE, qui présentent les états courants des entités selon la Figure 2. La justification des clauses *default 9999* sera donnée plus tard.

```
create table H_PROJET(
 INTITULE char(12)   not null,
 debut    integer    not null,
 fin      integer default 9999 not null,
 THEME    char(22)   not null,
 BUDGET   decimal(8) not null,
 primary key (INTITULE,debut));

create table H_EMPLOYE(
 CODE     char(5)    not null,
 debut    integer    not null,
```

```
 fin      integer default 9999 not null,
 NOM      char(10)   not null,
 STATUT   char(1)    not null,
 ADRESSE  char(10)   not null,
 PROJET   char(12)   not null,
 primary key (CODE,debut));

create view PROJET(INTITULE,THEME,BUDGET)
as select INTITULE,THEME,BUDGET
   from   H_PROJET
   where  fin = 9999;

create view EMPLOYE(CODE,NOM,STATUT,ADRESSE,PROJET)
as select CODE,NOM,STATUT,ADRESSE,PROJET
   from   H_EMPLOYE
   where  fin = 9999;
```

Code 1 - Définition des tables historiques et des vues des états courants (*transaction time*).

4.1 Représentation du temps physique

Comme nous l'avons déjà signalé, nous remplacerons le temps réel (confondu avec celui de l'horloge de la machine) par une suite d'entiers abstraits[9]. Nous avons défini un générateur de nombres qui garantit la production d'entiers uniques lors de ses invocations successives. Ce générateur, exprimé en SQL InterBase 5.6, simule la valeur courante de l'horloge de l'ordinateur[10], que nous désignerons à l'occasion par le terme now. Dans d'autres SGBD, on utilisera tout dispositif similaire, tel que les *Sequences* d'Oracle ou le type *NuméroAutomatique* d'Access.

```
create generator CURRENT_DATE;
set generator CURRENT_DATE to 15;
```

Code 2 - Simulation d'une horloge physique par un générateur d'entiers démarrant à 15 (InterBase).

Nous développerons les automates de gestion des données historiques pour les trois opérations de modification adressées aux vues des états courants. Pour chacune, nous proposerons une analyse puis le code du ou des *triggers*.

Nous avons développé des triggers associés aux tables de base H_PROJET et H_EMPLOYE. Lorsque le SGBD le permet, ces triggers pourront être attachés aux vues[11] PROJET et EMPLOYE elles-mêmes. Leur contenu sera bien entendu diffé-

9. Une des raisons est la difficulté d'obtenir en SQL une mesure suffisamment précise pour que deux états successifs d'une entité aient des valeurs de debut différentes. Signalons par exemple que le type TIME de certains SGBD a une précision de 1 msec, ce qui peut être insuffisant pour distinguer deux instants dans un système temps-réel.
10. On l'initialise conventionnellement à 15, et on décide que les instants se produisent de 5 en 5. Cette horloge fictive produira donc les instants 15, 20, 25, ...

rent. En outre, il conviendra d'empêcher la modification directe des tables de base par une politique adéquate de privilèges.

Les opérations illégales et les données invalides seront signalées par des *exceptions*, événements provoqués intentionnellement, qui annulent l'opération qui a déclenché le trigger ainsi que les opérations de ce dernier, et qui envoient un message au programme appelant ou, à défaut, à la console. Une exception peut aussi être captée par le trigger lui-même, auquel cas il n'y a pas d'annulation. On retiendra les exceptions suivantes, à titre indicatif[12] :

```
create exception ERR_ID "Erreur: unicité/non recyclabilité";
create exception ERR_FK "Erreur: intégrité référentielle";
create exception ERR_STAB_ID "Erreur: stabilité ID";
create exception ERR_NODIFF  "Erreur: états distincts";
```

Code 3 - Définition des exceptions liées à l'intégrité des données historiques.

4.2 Gestion des opérations de création d'entités

La création d'une entité s'effectue via une requête d'insertion dans la vue des états courants. C'est ainsi qu'on écrira :

```
insert into PROJET values
           ('PETRO','Rech. pétrol.',50000);

insert into EMPLOYE values
           ('B054','Bertier','P','Poitiers','BIOTECH');
```

Cette insertion sera contrôlée par un *trigger before insert* qui assignera aux bornes temporelles les valeurs debut=now et fin=9999, et qui vérifiera que les contraintes propres aux données historiques sont respectées :

1. *unicité de l'identifiant d'entité* : il n'existe pas d'autre projet courant de même valeur d'INTITULE (resp. de CODE);
2. *non recyclabilité de l'identifiant d'entité primaire* : la valeur d'INTITULE (resp. de CODE) n'a encore jamais été attribuée à un autre projet actuellement clôturé.
3. *unicité des identifiants d'entité secondaires*, s'il en existe (en revanche, ils ne doivent pas nécessairement être *non recyclables*)
4. *intégrité référentielle* : pour chaque clé étrangère temporelle (PROJET d'EMPLOYE), on vérifie que la table historique cible possède bien un état courant pour l'entité référencée.

11. Sous la forme de trigger instead of d'Oracle ou de triggers de vues en InterBase.
12. Dans une application réelle, on définira des événements plus informatifs, permettant notamment de distinguer l'origine exacte et la nature précise de l'erreur.

Les conditions 1 et 2 se résument à une seule : il n'y a au moment de l'insertion aucun état, passé ou courant, ayant la valeur proposée d'INTITULE (resp. de CODE).

a) Création d'une entité PROJET

Il faut être attentif au fait que certains *triggers* déclenchés par les autres primitives update et delete vont aussi exécuter des insert into H_PROJET. Il ne faut pas confondre ces insertions secondaires de nature technique avec celles que nous traitons ici, de sorte que le *trigger* que nous allons construire doit rester inactif lors de ces insertions techniques. La distinction entre les deux est simple : les insertions techniques sont caractérisées par fin < 9999 alors que la création d'une entité induit un état dont fin = 9999. Il faut cependant, pour que ces deux cas puissent être distingués, que cette propriété soit observable lors du déclenchement du *trigger*, avant même que le corps de celui-ci s'exécute. Ainsi se justifie la clause default 9999 de la définition de la colonne fin.

Trigger pour l'insert into PROJET

```
create trigger TRG_INSERT_PRO for H_PROJET before insert
as
declare variable N integer;
begin
  if (new.fin=9999) then     /* création d'une entité */
  begin
    select count(*) from H_PROJET
    where INTITULE = new.INTITULE into :N;
    if (N > 0) then
       exception ERR_ID;
    else
       new.debut = gen_id(CURRENT_DATE,5);
  end
end
```

Code 4 - *Trigger* gérant la création d'une entité PROJET (*transaction time*).

b) Création d'une entité EMPLOYE

Le fonctionnement du *trigger* est similaire à celui de H_PROJET, moyennant vérification supplémentaire de l'intégrité référentielle.

Trigger pour l'insert into EMPLOYE

```
create trigger TRG_INSERT_EMP for H_EMPLOYE before insert
as
declare variable N integer;
begin
  if (new.fin=9999) then     /* création d'une entité */
  begin
    select count(*) from H_EMPLOYE
    where CODE = new.CODE into :N;
```

```
    if (N > 0) then
       exception ERR_ID;
    else begin
       select count(*) from H_PROJET
       where INTITULE = new.PROJET and fin = 9999 into :N;
       if (N = 0) then
          exception ERR_FK;
       else new.debut = gen_id(CURRENT_DATE,5);
    end
  end
end
```

Code 5 - *Trigger* gérant la création d'une entité EMPLOYE (*transaction time*).

4.3 Gestion des opérations de modification des valeurs d'attributs

La modification des valeurs d'attributs d'une entité se traduit par un update adressé à la vue correspondant aux états courants des entités. On écrira par exemple :

```
update PROJET
set BUDGET = 150000
where INTITULE='BIOTECH';

update EMPLOYE
set STATUT = 'T', PROJET='BIOTECH'
where CODE='A237';
```

Six propriétés sont à contrôler.

1. *L'entité à modifier doit exister :* elle doit donc avoir un état courant. Cette validation n'est pas à prendre en charge, car elle le sera par la primitive SQL update, qui agit via la vue des états courants.
2. *Stabilité de l'identifiant d'entité* : la valeur d'INTITULE (resp. de CODE) n'est pas modifiée[13]. On considérera toute tentative de modification comme une erreur. Une autre réaction valide aurait pu être d'ignorer toute modification de l'identifiant (via l'instruction new.INTITULE = old.INTITULE).
3. *Unicité des identifiants secondaires d'entité* : en cas de modification d'un tel identifiant, la nouvelle valeur ne peut être celle de l'état courant d'une entité existante.
4. *Le nouvel état doit être différent du précédent* : au moins une colonne d'attribut subit une modification de valeur[14]. On pourrait envisager un autre type de réaction, consistant dans un tel cas à ignorer l'opération.

13. Ici encore, en cas de modification d'un identifiant secondaire, on vérifiera que la nouvelle valeur n'est pas déjà attribuée à un état courant.
14. Rien n'interdit qu'un update réattribue la même valeur à une colonne modifiée.

5. *Intégrité référentielle* : il faut vérifier qu'en cas de modification d'une colonne appartenant à une clé étrangère temporelle, il existe bien un projet actif (ayant un état courant) correspondant à la nouvelle valeur. Remarquons qu'en raison de la stabilité de l'identifiant primaire d'entité, il n'est pas nécessaire de considérer la question de la modification de l'identifiant de l'identifiant référencé, comme cela est nécessaire dans les bases de données non temporelles, via la clause update mode de la clé étrangère.
6. *Autres attributs stables et/ou non recyclables* : à contrôler comme les composants des identifiants primaires.

a) Modification d'une entité PROJET

Nous allons procéder en deux étapes. Avant l'opération, un premier *trigger* actualise l'état courant qui subit (via la vue) l'update en modifiant debut qui prend comme valeur l'instant courant. Après que l'update ait été exécuté, un deuxième *trigger* insère l'état passé directement précédent. Il est nécessaire de procéder en deux étapes, car (1) la mise à jour de l'état courant *avant* son update évite un accès au disque et (2) le nouvel état clôturé est en conflit d'identifiant avec l'ancien état courant et doit donc être inséré après l'update.

Deux remarques.

1. On ne peut demander la valeur de l'instant courant qu'une seule fois pour les deux *triggers*, d'où l'usage de `new.debut` dans le second. En effet chaque consultation de l'horloge renvoie une valeur différente.
2. On vérifie que le *trigger* `TRG_INSERT_PRO` ne se déclenchera pas de manière intempestive lors de l'insertion de l'ancien état courant.

Triggers pour l'update PROJET

```
create trigger TRG_B_UPDATE_PRO for H_PROJET before update
as
begin   /* l'état courant est mis à jour et reste courant */
  if (new.INTITULE <> old.INTITULE) then
     exception ERR_STAB_ID;
  else
     if (new.THEME = old.THEME and new.BUDGET = old.BUDGET) then
        exception ERR_NODIFF;
     else new.debut = gen_id(CURRENT_DATE,5);
end

create trigger TRG_A_UPDATE_PRO for H_PROJET after update
as
begin   /* après l'update, un ancien état courant clôturé est créé */
  insert into H_PROJET values
        (old.INTITULE,old.debut,new.debut,old.THEME,old.BUDGET);
end
```

Code 6 - *Triggers* gérant la modification d'une entité PROJET (*transaction time*).

On notera qu'en cas d'erreur, l'émission d'une exception clôture le *trigger* courant, annule l'opération d'update, et par conséquent empêche le déclenchement du second *trigger*.

b) Modification d'une entité EMPLOYE

La différence avec le contrôle de la table H_PROJET réside dans la validation de l'intégrité référentielle temporelle.

Triggers pour l'update EMPLOYE

```
create trigger TRG_B_UPDATE_EMP for H_EMPLOYE before update
as
declare variable N integer;
begin
  if (new.CODE <> old.CODE) then
     exception ERR_STAB_ID;
  else
     if (new.NOM = old.NOM and new.STATUT = old.STATUT
        and new.ADRESSE = old.ADRESSE
        and new.PROJET = old.PROJET) then
        exception ERR_NODIFF;
     else begin
        if (new.PROJET <> old.PROJET) then begin
           select count(*) from H_PROJET
           where INTITULE = new.PROJET and fin = 9999 into :N;
           if (N = 0) then
               exception ERR_FK;
        end
        new.debut = gen_id(CURRENT_DATE,5);
     end
end

create trigger TRG_A_UPDATE_EMP for H_EMPLOYE after update
as
begin
  insert into H_EMPLOYE values
         (old.CODE,old.debut,new.debut,old.NOM,old.STATUT,
          old.ADRESSE,old.PROJET);
end
```

Code 7 - *Triggers* gérant la modification d'une entité EMPLOYE (*transaction time*).

c) Variante en un seul trigger

Le problème du conflit d'identifiant nous a amené à procéder en deux étapes. Il est possible, avec une petite astuce, de le résoudre en une seule opération, et donc à l'aide d'un seul *trigger* before update. Ce *trigger* travaille comme suit :

1. la valeur old.debut est sauvegardée dans une variable;
2. l'état courant est modifié par un update de manière à donner à debut une valeur *non conflictuelle*, telle que 9999;

3. on insère (insert) l'état courant clôturé (fin=now);
4. on prépare la valeur new.debut = now;
5. on laisse l'update d'origine s'exécuter.

Le corps du trigger ne doit s'exécuter que pour un update de base et non pour l'update technique de l'étape 2. Il suffit de le protéger par la condition new.debut < 9999. Ce procédé permet de ne développer qu'un seul *trigger*, mais induit une opération d'update supplémentaire. La différence de performances dépend de la taille et de la politique de gestion des tampons.

4.4 Gestion des opérations de suppression d'entités

La suppression d'une entité s'effectue via une requête de suppression adressée à la vue des états courants. On écrira donc :

```
delete from PROJET where INTITULE = 'BIOTECH';

delete from EMPLOYE where ADRESSE = 'Grenoble';
```

Une opération de suppression est soumise à deux vérifications.

1. *L'entité à supprimer doit exister :* elle doit donc avoir un état courant. Cette validation n'est pas à prendre en charge, car elle le sera par la primitive SQL update, qui agit via la vue des états courants.
2. *Intégrité référentielle* : en cas de suppression d'une entité référencée (une entité PROJET), il y a lieu de prendre des dispositions pour que l'état final résultant de l'opération soit cohérent.

La seconde propriété suscite immédiatement la question de la politique à adopter lorsque des employés sont encore affectés au projet à supprimer. Traditionnellement, les SGBD relationnels proposent trois types de comportement : blocage (no action ou restrict), propagation (cascade) ou indépendance (set null ou set default). Comment réagir dans le cadre d'une base de données historiques ?

1. Mode **bloquant** : si des états courants de H_EMPLOYE référencent encore l'état courant du projet, on interdit cette suppression; l'application doit donc au préalable supprimer ou transférer ces employés.
2. Mode **propagation** : les états courants de H_EMPLOYE référençant l'état courant du projet sont clôturés; on supprime donc aussi les employés affectés au projet.
3. Mode **indépendance** : la colonne PROJET des états courants des employés affectés au projet est mise à null ou à la valeur default; on détache donc les employés du projet supprimé[15].

15. La mise à null de PROJET correspond à un update de chaque ligne dépendante de H_EMPLOYE. Cet update doit être géré soigneusement comme nous l'avons vu à la Section 4.3. On recommandera donc d'effectuer ces updates sur la vue EMPLOYE.

Le mode *propagation* n'est pas très réaliste dans notre cas, et est donc écarté. Le mode *indépendance* exigerait que la colonne PROJET soit facultative, ce qui n'est pas le cas. Il reste donc le mode bloquant, qui laisse à l'application le choix du sort des employés du projet à supprimer.

Trois remarques.

1. il ne peut s'agir d'un trigger before, qui serait amené à insérer une ligne dont les valeurs de {CODE,debut} seraient identiques à celles de l'état courant à supprimer, provoquant un conflit d'identifiant. Il faut donc attendre que l'état courant soit physiquement supprimé avant d'insérer l'état de clôture.
2. Lorsqu'on supprime un projet qui n'a jamais subi de modification, le trigger réintroduit l'unique état d'un projet, l'état courant (et unique) de ce projet venant d'être physiquement supprimé. Cette situation se distingue de la création d'un nouveau projet par le fait que fin < 9999. Il n'y a donc pas de risque que le trigger `TRG_INSERT_PRO` se déclenche de manière intempestive.
3. Si on peut attacher les triggers aux vues, alors on peut remplacer les opérations de suppression et l'insertion par une simple opération de modification de la colonne fin.

a) Suppression d'une entité PROJET

La seule validation concerne la clé étrangère temporelle PROJET d'EMPLOYE.

Trigger pour le delete from PROJET

```
create trigger TRG_DELETE_PRO for H_PROJET after delete
as
declare variable N integer;
declare variable NOW integer;
begin
  select count(*) from H_EMPLOYE
  where PROJET = old.INTITULE and fin = 9999 into :N;
  if (N > 0)
     then exception ERR_FK;
     else begin
        NOW = gen_id(CURRENT_DATE,5);
        insert into H_PROJET values
        (old.INTITULE, old.debut, :NOW,old.THEME, old.BUDGET);
     end
end
```

Code 8 - *Trigger* gérant la suppression d'une entité PROJET (*transaction time*).

b) Suppression d'une entité EMPLOYE

Le problème est ici beaucoup plus simple puisqu'il n'existe aucune précondition à vérifier.

Trigger pour le delete from EMPLOYE

```
create trigger TRG_DELETE_EMP for H_EMPLOYE after delete
as
declare variable NOW integer;
begin
  NOW = gen_id(CURRENT_DATE,5);
  insert into H_EMPLOYE values(old.CODE,old.debut,:NOW,
              old.NOM,old.STATUT,old.ADRESSE,old.PROJET);
end
```

Code 9 - *Trigger* gérant la suppression d'une entité EMPLOYE (*transaction time*).

5. Gestion des données historiques (temps logique ou *valid time*)

Nous allons à présent réétudier la question du contrôle des opérations de mise à jour des données dans l'hypothèse où le temps considéré n'est pas celui de la machine, mais celui du domaine d'application, soit le temps logique ou *valid time*. Dans ce cas, les valeurs de debut et de fin ne sont plus automatiquement fixées par la machine mais doivent être précisées par l'utilisateur.

H_PROJET				
INTITULE	**debut**	**fin**	**THEME**	**BUDGET**
AGRO-2000	87	114	...	65.000
AGRO-2000	114	125	...	75.000
AGRO-2000	125	9999	...	82.000

⇓

H_PROJET				
INTITULE	**debut**	**fin**	**THEME**	**BUDGET**
AGRO-2000	87	114	...	65.000
AGRO-2000	114	125	...	75.000
AGRO-2000	125	132	...	82.000
AGRO-2000	132	9999	...	86.000

Figure 10 - Une modification du budget d'AGRO-2000 intervient en 132 (temps logique).

Considérons à titre d'exemple qu'une modification du budget du projet AGRO-2000 survienne à l'instant 132 (Figure 10). Cet événement sera enregistré dans la base de données à un instant généralement postérieur à 132, soit par exemple en 140. La valeur de debut du nouvel état sera 140 si on considère le temps physique (*transaction time*) et 132 si on considère le temps logique (*valid time*). Si on désire représenter le temps logique, on doit donc fournir à la machine la valeur de debut, au même titre que toutes les valeurs des colonnes d'attributs.

Au moment de l'enregistrement d'un nouveau projet, nous devrons donc fournir une valeur pour chacune des colonnes INTITULE, debut, THEME et BUDGET.

5.1 Peut-on modifier le passé ?

Une telle question n'avait pas de sens lorsqu'on considérait le temps physique, puisque la gestion de ce dernier n'était pas du ressort de l'utilisateur. La question est au contraire pertinente pour le temps logique. Ce dernier étant une donnée comme les autres, il peut en effet sembler justifié qu'elle puisse subir des modifications ultérieurement à son enregistrement.

Ainsi, on pourrait s'apercevoir que la modification mentionnée ci-dessus a en fait eu lieu plus tôt qu'on ne l'avait cru, soit en 127 et non en 132. Dans ce cas, on doit corriger les deux derniers états, qui deviennent respectivement <AGRO-2000,125,**127**, ...> et <AGRO-2000,**127**,9999, ...> (Figure 11).

H_PROJET				
INTITULE	**debut**	**fin**	**THEME**	**BUDGET**
AGRO-2000	87	114	...	65.000
AGRO-2000	114	125	...	75.000
AGRO-2000	125	132	...	82.000
AGRO-2000	132	9999	...	86.000

⇓

H_PROJET				
INTITULE	**debut**	**fin**	**THEME**	**BUDGET**
AGRO-2000	87	114	...	65.000
AGRO-2000	114	125	...	75.000
AGRO-2000	125	**127**	...	82.000
AGRO-2000	**127**	9999	...	86.000

Figure 11 - On corrige une erreur : la modification de budget n'est pas intervenue en 132 mais en **127**.

Les choses ne sont malheureusement pas aussi simples. Admettons que l'erreur ne soit pas *127 au lieu de 132* mais *122 au lieu de 132*. Si on reporte ces modifications dans la base de données, l'état courant devient <AGRO-2000,**122**,9999, ...> au lieu de <AGRO-2000,**132**,9999, ...>.

Mais alors, un problème se pose pour l'état précédent <AGRO-2000,125,132, ...>, qui est entièrement inclus dans l'état courant corrigé : cet état doit donc disparaître, comme s'il n'avait jamais existé[16] (Figure 12) ! En toute généralité, les modifications de correction d'un historique peuvent se traduire par l'introduction de nouveaux états, la suppression d'états existants, leur fusion ou leur éclatement.

16. On perd donc une information utile. Les bases de données bitemporelles résoudront ce problème.

On conçoit aisément que le concept de **correction** des états passés pose des problèmes nouveaux qu'on ne peut résoudre dans le cadre limité de ce projet. C'est pour cette raison que nous excluerons les *modifications de correction* pour nous limiter aux *modifications d'évolution*, comme nous l'avons fait pour le temps physique.

H_PROJET				
INTITULE	**debut**	**fin**	**THEME**	**BUDGET**
AGRO-2000	87	114	...	65.000
AGRO-2000	114	125	...	75.000
AGRO-2000	125	132	...	82.000
AGRO-2000	132	9999	...	86.000

⇓

H_PROJET				
INTITULE	**debut**	**fin**	**THEME**	**BUDGET**
AGRO-2000	87	114	...	65.000
AGRO-2000	114	**122**	...	75.000
AGRO-2000	**122**	9999	...	86.000

Figure 12 - Erreur : la modification de budget n'est pas intervenue en 132 mais en **122**. Que devient l'état <AGRO-2000,125,132,...> ? Dans le scénario ci-dessus, il disparaît.

A priori, nous voudrions permettre à l'utilisateur d'effectuer les mêmes opérations que pour les historiques à temps physique : *créer une entité*, *modifier les valeurs d'attribut d'une entité* et *supprimer une entité*, à la différence près qu'il lui faudra en outre préciser explicitement dans chaque cas l'instant de l'événement. La nature même du temps considéré ne nous permet cependant pas de pousser cette logique jusqu'au bout : en effet, la suppression d'une entité doit spécifier l'instant auquel cette entité a disparu, ce que le *delete* ne permet pas. La suppression d'une entité correspondant à sa clôture, nous considérerons cette suppression comme la simple modification (update) de la colonne fin de l'actuel état courant.

La définition des tables peut rester celle qui a été proposée en Code 1, mais les vues donnant les états courants doivent à présent inclure les colonnes debut (pour la création et la modification) et fin (pour la suppression), ce qui correspond au Code 10[17].

```
create view PROJET(INTITULE,debut,fin,THEME,BUDGET)
as select INTITULE,debut,fin,THEME,BUDGET
   from   H_PROJET
   where  fin = 9999;

create view EMPLOYE(CODE,debut,fin,NOM,STATUT,ADRESSE,PROJET)
as select CODE,debut,fin,NOM,STATUT,ADRESSE,PROJET
   from   H_EMPLOYE
```

17. On évitera soigneusement la clause with check option. Pourquoi ?

```
where  fin = 9999;
```

Code 10 - Définition des vues des états courants (*valid time*).

5.2 Gestion des opérations de création d'entités

La commande de création d'une entité se présente comme suit :

```
insert into PROJET (INTITULE,debut,THEME,BUDGET)
            values ('PETRO', 140, 'Rech. pétrol.', 50000);
```

Son effet est d'introduire dans la table H_PROJET une ligne reprenant ces valeurs accompagnées de fin=9999.

La valeur fin=9999 étant définie automatiquement par la clause default, le *trigger* a uniquement pour but de contrôler la validité des valeurs introduites. On y vérifiera les mêmes propriétés que pour le temps physique, auxquelles s'ajoute la validité de l'instant spécifié via debut :

1. *unicité de l'identifiant d'entité* : il n'existe pas d'autre projet courant de même valeur d'INTITULE (resp. de CODE);
2. *non recyclabilité de l'identifiant primaire d'entité* : la valeur d'INTITULE (resp. de CODE) n'a encore jamais été attribuée à un autre projet actuellement clôturé.
3. *unicité des identifiants secondaires d'entité*, s'il en existe (en revanche, ils ne sont pas soumis à la contrainte de non recyclabilité)
4. *intégrité référentielle* : pour chaque clé étrangère temporelle (PROJET d'EMPLOYE), on vérifie que l'état inséré a une période de validité qui est entièrement comprise dans la période de vie de l'entité référencée. En d'autres termes, la table historique cible possède bien un état courant pour l'entité référencée et un état (passé ou courant) qui contient la valeur de debut[18].
5. *l'instant de l'événement est correct* : en toute généralité, il faudrait vérifier que cet instant n'appartient pas au futur, puisque nous n'avons pas admis de représenter les événements futurs; on doit donc vérifier que debut ≤ now; en pratique, étant donné la difficulté de comparer le temps logique (modélisé par des entiers abstraits) et le temps physique, qui donne l'instant présent, on se contentera de la condition dégénérée debut ≠ 9999.

Ici encore, les deux premières conditions se résument à une seule : il n'y a au moment de l'insertion aucun état, passé ou courant, qui ait la valeur proposée d'INTITULE (resp. de CODE).

Pour le contrôle de la propriété 5, nous définirons une nouvelle exception :

18. Cette condition tient compte des situations où l'état de l'employé qu'on insère existait avant l'état courant du projet référencé (le projet a changé depuis le début de l'état de l'employé qu'on enregistre). Une telle situation ne peut exister en temps physique.

```
create exception ERR_PERIOD  "Erreur: période de validité";
```

Code 11 - Exception d'erreur de période en temps logique.

a) Création d'une entité PROJET

Trigger pour l'insert into PROJET

```
create trigger TRG_INSERT_PRO for H_PROJET before insert
as
declare variable N integer;
begin
  if (new.fin=9999) then     /* création d'une entité */
  begin
    select count(*) from H_PROJET
    where INTITULE = new.INTITULE into :N;
    if (N > 0) then
       exception ERR_ID;
    else
       if (NEW.debut=9999) then
          exception ERR_PERIOD;
  end
end
```

Code 12 - *Trigger* gérant la création d'une entité PROJET (*valid time*).

b) Création d'une entité EMPLOYE

Les vérifications sont celles de H_PROJET, auxquelles on ajoute l'intégrité référentielle.

Trigger pour l'insert into EMPLOYE

```
create trigger TRG_INSERT_EMP for H_EMPLOYE before insert
as
declare variable N integer;
declare variable DM integer;
begin
  if (new.fin=9999) then     /* création d'une entité */
  begin
    select count(*) from H_EMPLOYE
    where CODE = new.CODE into :N;
    if (N > 0) then
       exception ERR_ID;
    else begin
       if (new.debut=9999) then
          exception ERR_PERIOD;
       else begin
          select count(*) from H_PROJET
          where INTITULE = new.PROJET and fin = 9999 into :N;
          select min(debut) from H_PROJET
          where INTITULE = new.PROJET into :DM;
```

```
          if (N = 0 or new.debut < DM) then
             exception ERR_FK;
       end
    end
  end
end
```

Code 13 - *Trigger* gérant la création d'une entité EMPLOYE (*valid time*).

5.3 Gestion des opérations de modification des valeurs d'attributs

Une commande de modification de l'état d'une entité, en l'occurrence un projet, se présenterait comme suit :

```
update PROJET
set    BUDGET = 148000,
       debut = 145
where  INTITULE = 'BIOTECH'
```

Son effet est d'actualiser l'état courant dans la table H_PROJET et d'introduire un état courant clôturé sous la forme d'une ligne reprenant les valeurs anciennes accompagnées (dans le cas de notre exemple) de fin=145.

Comme pour le *transaction time*, et pour les mêmes raisons, on procédera en deux étapes. Le premier *trigger* met à jour l'état courant, ce qui ne demande aucune autre action que la vérification de la validité de l'opération. Le second introduit l'ancien état courant clôturé.

On vérifiera les sept propriétés suivantes.

1. *L'entité à modifier doit exister :* validation prise en charge par la vue des états courants.
2. *Stabilité de l'identifiant primaire d'entité.*
3. *Unicité des identifiants secondaires d'entité.*
4. *Le nouvel état doit être différent du précédent.*
5. *Intégrité référentielle.*
6. *Autres attributs stables et/ou non recyclables.*
7. *l'instant de l'événement est correct* : l'instant de l'événement de modification (nouvelle valeur de debut) est postérieur à celui de l'état courant; comme précédemment, on se contentera de debut ≠ 9999.

Pour des raisons que nous analyserons dans la section consacrée à la suppression d'entités (qui se traduit ausi par un update), les *triggers* de modification de colonnes d'attribut ne doivent s'activer que pour des updates laissant la colonne fin à 9999.

a) Modification d'une entité PROJET

Triggers pour l'update PROJET

```
create trigger TRG_B_UPDATE_PRO for H_PROJET before update
as
begin   /* l'état courant est mis à jour et reste courant */
  if (new.fin = 9999) then
  begin
    if (new.INTITULE <> old.INTITULE) then
       exception ERR_STAB_ID;
    else
       if (new.THEME=old.THEME and new.BUDGET=old.BUDGET) then
          exception ERR_NODIFF;
       else
          if (new.debut <= old.debut or new.debut = 9999) then
            exception ERR_PERIOD;
  end
end

create trigger TRG_A_UPDATE_PRO for H_PROJET after update
as
begin /* un ancien état courant clôturé est créé */
  if (new.fin = 9999) then
    insert into H_PROJET values
    (old.INTITULE,old.debut,new.debut,old.THEME,old.BUDGET);
end
```

Code 14 - *Trigger* gérant la modification d'un PROJET (*valid time*).

b) Modification d'une entité EMPLOYE

Nous ajouterons simplement la validation de l'*intégrité référentielle* lorsque la valeur de la colonne PROJET est modifiée.

Triggers pour l'update EMPLOYE

```
create trigger TRG_B_UPDATE_EMP for H_EMPLOYE before update
as
declare variable N integer;
declare variable DM integer;
begin
  if (new.fin = 9999) then
  begin
    if (new.CODE <> old.CODE) then
       exception ERR_STAB_ID;
    else
       if (new.NOM = old.NOM and new.STATUT = old.STATUT
           and new.ADRESSE = old.ADRESSE
           and new.PROJET = old.PROJET) then
          exception ERR_NODIFF;
       else
```

```
          if (new.debut <= old.debut) then
             exception ERR_PERIOD;
          else
             if (new.PROJET <> old.PROJET) then begin
               select count(*) from H_PROJET
               where INTITULE = new.PROJET
               and fin = 9999 into :N;
               select min(debut) from H_PROJET
               where INTITULE = new.PROJET into :DM;
               if (N = 0 or new.debut < DM) then
                   exception ERR_FK;
             end
  end
end

create trigger TRG_A_UPDATE_EMP for H_EMPLOYE after update
as
begin
  if (new.fin = 9999) then
    insert into H_EMPLOYE values (old.CODE,old.debut,new.debut,
         old.NOM,old.STATUT,old.ADRESSE,old.PROJET);
end
```

Code 15 - *Triggers* gérant la modification d'une entité EMPLOYE (*valid time*).

5.4 Gestion des opérations de suppression d'entités

Comme nous l'avons observé, la suppression d'une entité ne peut pas être demandée par une requête delete mais par un update de la colonne fin. En contrepartie, il est nécessaire d'interdire aux utilisateurs d'exécuter directement une requête delete qui corromprait les données historiques. La suppression d'un PROJET sera donc demandée sous la forme suivante :

```
update PROJET
set    fin = 145
where  INTITULE = 'BIOTECH'
```

L'effet de cette opération est très simple : elle clôture l'état courant selon la valeur de fin transmise par la requête update. Cependant, la requête d'update étant potentiellement plus riche que le simple delete, il faudra vérifier que les paramètres correspondent strictement à une suppression et que l'utilisateur ne tente pas *subrepticement* de modifier d'autres colonnes de l'état courant. On vérifiera donc les propriétés suivantes :

1. *L'entité à supprimer doit exister* (vérifié par construction via le mécanisme de la vue).
2. *Suppression seulement* : seule la colonne fin est modifiée.
3. *L'instant de l'événement est correct* : la valeur de fin est postérieure à celle de

debut de l'état courant et la nouvelle valeur de fin est différente de 9999.

4. *Intégrité référentielle* : en cas de suppression d'une entité référencée (une entité PROJET), il y a lieu de prendre des dispositions pour que l'état final résultant de l'opération soit cohérent.
5. *Pas de delete direct* : la primitive delete ne peut être exécutée en direct.

Afin de distinguer l'update correspondant à une modification de colonnes d'attributs de celui qui traduit un delete, on filtrera la partie action du trigger selon la nouvelle valeur de fin, qui est égale à 9999 pour une modification et qui est inférieure à 9999 pour une suppression.

Deux nouvelles exceptions seront nécessaires :

```
create exception ERR_DEL  "Erreur: delete sans modification
                           d'attributs";

create exception ERR_NO_DEL  "Erreur: delete direct interdit";
```

Code 16 - Exceptions d'erreurs en cas de suppression.

a) Suppression d'une entité PROJET

Triggers pour le delete PROJET

```
create trigger TRG_DELETE_PRO for H_PROJET before update
as
declare variable N integer;
begin
  if (new.fin < 9999) then
  begin
    if (new.INTITULE <> old.INTITULE
        or new.THEME <> old.THEME
        or new.BUDGET <> old.BUDGET) then
       exception ERR_DEL;
    else
       if (new.fin <= old.debut) then
         exception ERR_PERIOD;
       else begin
         select count(*) from H_EMPLOYE
         where PROJET = old.INTITULE and fin = 9999 into :N;
         if (N > 0)
            then exception ERR_FK;
       end
  end
end

create trigger TRG_DELETE_PRO2 for H_PROJET before delete
as
begin
  exception ERR_NO_DEL;
```

```
end
```

Code 17 - *Triggers* gérant la suppression d'une entité PROJET (*valid time*).

b) Suppression d'une entité EMPLOYE

Ce cas des entités EMPLOYE étant plus simple, le trigger se déduit immédiatement :

Triggers pour le delete EMPLOYE

```
create trigger TRG_DELETE_EMP for H_EMPLOYE before update
as
begin   /* l'état courant est mis à jour et reste courant */
  if (new.fin < 9999) then
  begin
    if (new.CODE <> old.CODE
       or new.NOM <> old.NOM or new.STATUT <> old.STATUT
       or new.ADRESSE <> old.ADRESSE
       or new.PROJET <> old.PROJET) then
       exception ERR_DEL;
    else
       if (new.fin <= old.debut) then
         exception ERR_PERIOD;
  end
end

create trigger TRG_DELETE_EMP2 for H_EMPLOYE before delete
as
begin
  exception ERR_NO_DEL;
end
```

Code 18 - *Triggers* gérant la suppression d'une entité EMPLOYE (*valid time*).

Partie 3

Comment interroger des données historiques ?

6. Interrogation d'une base de données historiques

Une base de données historiques peut offrir une réponse à une grande variété de requêtes à connotation temporelle. On conçoit aisément que l'extraction de données, ou consultation, d'un ensemble de tables historiques soit plus complexe que les opérations similaires qui s'adressent à une base de données classique, limitée aux états courants du domaine d'application. Ce sentiment est à la fois vrai et faux. Certaines requêtes auront en SQL une formulation simple et intuitive, alors que d'autres réclameront des expressions SQL beaucoup plus complexes, sans commune mesure avec leur formulation en français courant[19].

6.1 Relations temporelles

Les requêtes de consultation de données temporelles seront souvent basées sur certaines relations existant entre entre deux instants, entre un instant et un intervalle ou entre deux intervalles. Nous décrirons brièvement ces relations, qui spécifient les positions relatives de leurs arguments. Ces relations ne sont pas primitives dans la mesure où certaines d'entre elles sont des combinaisons d'autres relations.

a) relations entre deux instants t1 et t2

1. t1 et t2 sont simultanés : $t1 = t2$
2. t1 et t2 ne sont pas simultanés : $t1 \neq t2$
3. t2 est postérieur à t1 : $t1 < t2$
4. t2 n'est pas antérieur à t1 : $t1 \leq t2$

b) Relations entre un instant t et un intervalle i ≡ [d,f)

5. t est antérieur à i : $t < d$
6. t est postérieur à i : $t \geq f$
7. i commence en t : $t = d$
8. i se termine en t : $t = f-1$
9. i comprend t : $d \leq t < f$
10. i se termine après t : $t < f-1$
11. i commence avant t : $t > d$

c) Relations entre deux intervalles i1 ≡ [d1,f1) et i2 ≡ [d2,f2)

12. i2 est postérieur à i1 : $f1 \leq d2$

19. . . . et qui justifieront l'essentiel des recherches sur les langages des bases de données temporelles [Snodgrass, 1994].

13. i1 et i2 sont disjoints : $(d1 \geq f2) \vee (d2 \geq f1)$
14. i1 et i2 sont non disjoints : $(d1 < f2) \wedge (d2 < f1)$

    Remarque : l'intersection de i1 et i2 non disjoints est [max(d1,d2), min(f1,f2))

15. i1 et i2 sont consécutifs : $f1 = d2$
16. i1 touche i2 à droite : $f1 = d2+1$
17. i1 est antérieur à i2 : $f1 \leq d2$
18. i1 comprend i2 : $(d1 \leq d2) \wedge (f2 \leq f1)$

On notera que certaines relations (8, 10 et 16) ne sont pas indépendantes de l'unité, ou granularité, temporelle.

6.2 Interrogation simple d'une table historique

Nous donnerons quelques exemples représentatifs de requêtes simples s'adressant à une seule table.

Les requêtes non temporelles

Une table historique reprend les états courants et peut donc satisfaire les requêtes classiques propres aux bases de données non temporelles. Exprimées sur les vues des états courants, ces requêtes sont identiques à celles des bases de données non temporelles équivalentes.

1. Quel l'état actuel de l'employé D122 ?

```
select CODE,NOM,STATUT,ADRESSE,PROJET
from   EMPLOYE
where  CODE = 'D122'
```

   ou, exprimée sur la table historique de base,

```
select CODE,NOM,STATUT,ADRESSE,PROJET
from   H_EMPLOYE
where  CODE = 'D122' and fin = 9999
```

2. Quel(s) projet(s) dispose(nt) du budget le plus élevé ?

```
select distinct INTITULE,BUDGET
from   PROJET
where  BUDGET = (select max(BUDGET) from PROJET)
```

Requêtes temporelles à résultat temporel

Une requête temporelle interroge l'ensemble des états des entités. Son résultat peut être temporel ou non. Nous considérerons d'abord les requêtes qui produisent des données historiques.

3. Extraire l'historique des employés entre 90 et 110

```
select CODE,
       case when debut < 90 then 90  else debut end,
       case when fin > 110  then 110 else fin   end,
       NOM,STATUT,ADRESSE,PROJET
from   H_EMPLOYE
where  debut < 110 and 90 < fin
```

   Justification. Un état est retenu si son intervalle n'est pas disjoint de [90,110).

4. Extraire l'historique d'existence (période de vie) des projets

```
select INTITULE,min(debut),max(fin)
from   H_PROJET
group by INTITULE
```

5. Calculer la date de début et la durée de vie de chaque projet clôturé

```
select INTITULE, min(debut), max(fin)-min(debut)
from   H_PROJET
group by INTITULE
having max(fin) < 9999
```

6. Quand le budget de chaque projet a-t-il dépassé 150.000 ?

```
select INTITULE,debut,fin,BUDGET
from   H_PROJET
where BUDGET > 150000
```

   Attention. Cette requête peut produire un historique non normalisé, les états n'étant plus nécessairement jointifs et des états jointifs pouvant être identiques. Il faudra en tenir compte si le résultat fait l'objet d'opérations ultérieures. On fera si nécessaire appel à un opérateur de normalisation.

7. Quel a été (et pendant quelles périodes) le budget maximum de chaque projet ?

```
select INTITULE,debut,fin,BUDGET
from   H_PROJET P
where  BUDGET = (select max(BUDGET)
                 from   H_PROJET
                 where  INTITULE = P.INTITULE)
```

Attention. Le résultat ne sera pas normalisé lorsqu'un projet maintient son budget maximum pendant plusieurs état consécutifs (états jointifs identiques).

Requêtes temporelles à résultat non temporel

Ces requêtes éliminent la dimension temporelle, soit en réduisant l'espace de recherche à un instant déterminé, soit en utilisant une fonction agrégative (statistique) sur les bornes temporelles.

8. Quel était l'état de l'employé D107 à l'instant 100 ?

```
select CODE,NOM,STATUT,ADRESSE,PROJET
from   H_EMPLOYE
where  CODE = 'D107'
and    debut <= 100 and 100 < fin
```

Remarque. Cette requête construit ce qu'on appelle un *instantané* ou *snapshot*. Un instantané est un état qui a existé en tant qu'état courant dans le passé, ou qui existe encore.

9. Quel a été le budget maximum de chaque projet ?

```
select INTITULE, max(BUDGET)
from   H_PROJET
group by INTITULE
```

10. Combien d'employés étaient affectés au projet SURVEYOR en 110 ?

```
select count(distinct CODE)
from   H_EMPLOYE
where  PROJET = 'SURVEYOR'
and    debut <= 110 and 110 < fin
```

La clause distinct est-elle indispensable ? Justifiez.

11. A combien de projets l'employé N240 a-t-il été affecté pendant sa carrière ?

```
select count(distinct PROJET)
from   H_EMPLOYE
where  CODE = 'N240'
```

12. A combien de projets l'employé D107 a-t-il été affecté pendant qu'il était domicilié à Genève ?

```
select count(distinct PROJET)
from   H_EMPLOYE
where  CODE = 'D107' and ADRESSE = 'Genève'
```

13. Quel est (sont) le(s) projet(s) au(x)quel(s) l'employé N240 a été affecté le plus longtemps ?

!! traitement incorrect des états courants. remplacer dans ce cas End par "current_date".

```
select PROJET, sum(fin - debut)
from   H_EMPLOYE
where  CODE = 'N240'
group by PROJET
having sum(fin - debut) >= all (select sum(fin - debut)
                                from   H_EMPLOYE
                                where  CODE = 'N240'
                                group by PROJET)
```

6.3 Les requêtes complexes

Malheureusement, d'autres requêtes, en apparence anodines, vont exiger des expressions en SQL standard qu'on peut qualifier pudiquement de *très complexes*. Nous examinerons et résoudrons trois opérateurs classiques qui réclament le plus grand soin lorsqu'ils sont appliqués à des données temporelles, à savoir,

1. la *projection temporelle* (Sections 7 et 8),
2. la *jointure temporelle* (Section 9).
3. et l'*agrégation temporelle* (Section 10).

Grâce à ces opérateurs, nous pourrons résoudre, à l'occasion en plusieurs étapes, la plupart des questions qui s'adressent à une base de données temporelle, et qui sont pour l'instant hors de notre portée.

Nous y ajouterons des opérateurs de service, dits de normalisation, qui sont destinés à reconditionner des historiques présentant des lacunes ou des défauts :

4. *normalisation temporelle* d'un historique (Section 11).

7. Projection temporelle d'une table historique (version simplifiée)

La projection est l'opérateur le plus simple qu'on puisse appliquer à des données non temporelles. Il consiste à restreindre les lignes d'une sélection à certaines colonnes de la table initiale. Considérant la table PROJET de la Figure 2, sa projection sur les colonnes INTITULE et THEME, opérateur noté sous forme algébrique PROJET[INTITULE,THEME], s'exprime en SQL de la manière suivante :

```
select INTITULE,THEME
from   PROJET
```

Lorsque l'identifiant n'est pas repris parmi les colonnes de la projection, on éliminera les doubles potentiels à l'aide du modifieur distinct :

```
select distinct THEME
from   PROJET
```

Paradoxalement, la projection temporelle est sans doute l'opérateur le plus complexe de ceux que nous rencontrerons. Nous allons en analyser une première version à partir de l'exemple suivant : *donner l'évolution du thème des projets au cours du temps, indépendamment de celle du budget*.

On demande un *historique partiel*, réduit aux colonnes INTITULE et THEME, ce qui correspond à la **projection temporelle** de H_PROJET sur INTITULE et THEME.

Il est important de noter que **l'identifiant d'entité est repris parmi les colonnes de projection**, ce qui simplifiera le raisonnement, tout comme pour la projection d'une table non temporelle comme nous venons de le rappeler. Dans une section ultérieure, nous généraliserons à la projection qui ne reprend pas nécessairement cet identifiant.

7.1 Premières tentatives

Par analogie avec la projection non temporelle, une première idée naïve de requête pourrait être la suivante :

```
select INTITULE,debut,fin,THEME
from   H_PROJET
```

Code 19 - Extraction de l'historique des thèmes des projets - Version naïve.

Cette requête produit le résultat de la Figure 13.

Le résultat ne contient pas d'erreurs, mais n'est évidemment pas celui qu'on attend, certains états consécutifs étant identiques. On pourrait fusionner ces états en *concaténant* les périodes de chaque projet ayant même valeur de THEME, c'est-à-dire en retenant pour chaque groupe de lignes de mêmes valeurs d'INTITULE et de THEME le plus petit début de période et la plus grande fin de période (Code Figure 20) :

```
insert into H_THEME_2
select INTITULE,min(debut),max(fin),THEME
from   H_PROJET
group by INTITULE,THEME
```

Code 20 - Extraction de l'historique des thèmes des projets - Version *améliorée* mais erronée.

H_THEME_1			
INTITULE	**debut**	**fin**	**THEME**
AGRO-2000	87	114	Amélior. céréales
AGRO-2000	114	125	Amélior. céréales
AGRO-2000	125	9999	Amélior. céréales
BIOTECH	10	41	Biotechnologie
BIOTECH	41	47	Génie génétique
BIOTECH	47	84	Génie génétique
BIOTECH	84	135	Génie génétique
BIOTECH	135	9999	Biotechnologie
SURVEYOR	65	80	Surv. par satellite
SURVEYOR	80	101	Surv. par satellite
SURVEYOR	101	120	Surv. par satellite

Figure 13 - Historique des projets réduits aux valeurs de THEME. Version naïve mais maladroite : plusieurs états successifs ont mêmes valeurs !

Le résultat (Figure 14) semble beaucoup plus proche de ce que nous attendions.

H_THEME_2			
INTITULE	**debut**	**fin**	**THEME**
BIOTECH	10	9999	Biotechnologie
BIOTECH	41	135	Génie génétique
AGRO-2000	87	9999	Amélior. céréales
SURVEYOR	65	120	Surv. par satellite

Figure 14 - Historique des projets réduits aux valeurs de THEME. Nouvelle tentative, mais cette fois incorrecte : un état est inclus dans un autre !

Si ce résultat est satisfaisant pour les projets `AGRO-2000` et `SURVEYOR`, il comporte malheureusement une anomalie grave en ce qui concerne le projet `BIOTECH`. En effet, les deux périodes `[10,9999)` et `[41,135)` ne sont pas jointives, ni même disjointes, l'une étant incluse dans l'autre ! La raison de cette situation est aisée à déterminer : il existe pour ce projet deux périodes dont le thème est *Biotechnologie*, séparées par une période correspondant à un autre thème, *Génie génétique*. La requête a erronément fusionné toutes les périodes de même thème, même quand elle n'étaient pas jointives.

Pour résoudre ce problème, il est nécessaire de construire un état de l'historique final d'un projet à partir de chaque *suite maximale d'états consécutifs de même valeur de THEME*, ce qui exige une procédure plus complexe que nous allons développer.

7.2 Projection temporelle - Version prédicative

Tentons de définir ce qu'on entend par *suite maximale d'états consécutifs de même valeur de THEME*.

Une telle suite est délimitée par les deux états P1 et P2 (Figure 15), P1 étant antérieur à P2, relatifs au même projet (INTITULE) et de même valeur de THEME (soit b cette valeur). Nous admettrons bien sûr les suites d'un seul état, ce qui nous fait dire que P1 et P2 peuvent ne former qu'un seul et même état. De cette suite, nous retenons debut de P1 et fin de P2.

On forme les couples de la manière suivante :

```
select P1.INTITULE, P1.THEME, P1.debut, P2.fin
from   H_PROJET P1, H_PROJET P2
where  P1.INTITULE=P2.INTITULE and P1.THEME=P2.THEME
and    P1.debut <= P2.debut
```

Il faut ensuite préciser que P1 est le premier état de la suite, c'est-à-dire qu'il n'est pas précédé d'un état identique :

```
and not exists (select *
                from   H_PROJET P0
                where  P0.INTITULE=P1.INTITULE
                       and P0.THEME=P1.THEME
                and    P0.fin = P1.debut)
```

. . . et que P2 en est le dernier, c'est-à-dire qu'il n'est pas suivi d'un état identique :

```
and not exists (select *
                from   H_PROJET P3
                where  P3.INTITULE=P1.INTITULE
                       and P3.THEME=P1.THEME
                and    P2.fin = P3.debut)
```

Ce n'est cependant pas tout : il faut encore que ces états extrêmes n'englobent pas d'états étrangers, c'est-à-dire d'états (notés P12) compris entre P1 et P2, relatifs au même projet, mais de valeurs différentes de THEME. Nous devons donc ajouter :

```
and not exists (select *
                from   H_PROJET P12
                where  P12.INTITULE=P1.INTITULE
                       and P12.THEME<>P1.THEME
                and    P1.fin <= P12.debut
                and    P12.fin <= P2.debut)
```

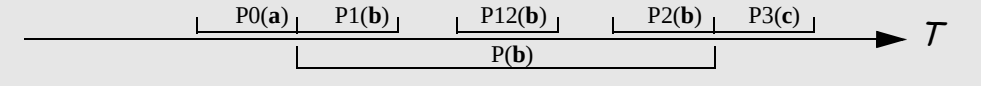


Figure 15 - Les intervalles P1, ..., P12, ..., P2 forment une suite maximale d'états de même valeur (soit b) si P0 et P3, s'ils existent, sont de valeurs différentes (resp. a et

c) et si tous les états, tels que P12, entre P1 et P2 sont de même valeur que P1 (soit b). Ils sont fusionnés pour former un état unique P de même valeur que P1 (soit b).

En rassemblant ces propriétés, nous obtenons la requête complète ci-dessous, qui donne le résultat définitif de la Figure 16.

```
insert into H_THEME
select P1.INTITULE, P1.THEME, P1.debut, P2.fin
from   H_PROJET P1, H_PROJET P2
where  P1.INTITULE=P2.INTITULE and P1.THEME=P2.THEME
and    P1.debut <= P2.debut
and not exists (select *
                from   H_PROJET P0
                where  P0.INTITULE=P1.INTITULE
                       and P0.THEME=P1.THEME
                and    P0.fin = P1.debut)
and not exists (select *
                from   H_PROJET P3
                where  P3.INTITULE=P1.INTITULE
                       and P3.THEME=P1.THEME
                and    P2.fin = P3.debut)
and not exists (select *
                from   H_PROJET P12
                where  P12.INTITULE=P1.INTITULE
                and    P12.THEME<>P1.THEME
                and    P1.fin <= P12.debut
                and    P12.fin <= P2.debut)
```

Code 21 - Extraction de l'historique des thèmes des projets - Version prédicative correcte.

H_THEME			
INTITULE	**debut**	**fin**	**THEME**
BIOTECH	10	41	Biotechnologie
BIOTECH	41	135	Génie génétique
BIOTECH	135	9999	Biotechnologie
AGRO-2000	87	9999	Amélior. céréales
SURVEYOR	65	120	Surv. par satellite

Figure 16 - Historique des projets réduits aux valeurs de THEME. Version correcte.

Cette expression est cependant complexe et s'avérera inefficace pour une table de grande taille[20]. Une solution plus simple et certainement plus efficace consiste à extraire de la table H_PROJET des lignes composées de <INTITULE, debut, fin, THEME>, triées par INTITULE et debut, puis à constituer les états synthétiques à partir des suites de lignes de mêmes valeurs de INTITULE et THEME. C'est cette version procédurale que nous allons examiner dans la Section 7.4.

20. Nous discuterons plus loin de la question des performances (Section 14.8).

Remarques

Les requêtes proposées sont basées sur l'hypothèse d'une table historique initiale *complète* (pas d'états manquants) et *jointive* (pas de recouvrements). En particulier, la prise en compte de l'absence de certains états et de la présence d'états en recouvrement réclamerait une requête plus complexe, telle que celle que nous développerons à la Section 8.

L'expression prédicative est facile à insérer dans tout programme d'application et permet de définir des vues correspondant à des historiques partiels. En revanche, elle sera en général plus coûteuse que la version procédurale.

7.3 L'opérateur de *coalescing*

Cette opération de transformation de suites homogènes en états distincts s'appelle *synthèse* ou *coalescing*. La projection temporelle est basée sur cet opérateur, mais ce dernier pourrait être proposé comme opérateur autonome, applicable à des tables d'origine indéterminée sujettes à des anomalies. Observons cependant que la projection d'une table historique sur toutes ses colonnes réalise un *coalescing*. Il n'est donc pas nécessaire d'isoler cette fonction sous la forme d'un opérateur spécifique.

7.4 Projection temporelle - Version procédurale

Si on accepte de développer une *procédure* de projection plutôt qu'une *requête*, on peut proposer une solution à la fois simple, élégante et efficace. Une telle procédure fonctionne comme suit.

1. On extrait de H_PROJET des lignes constituées des colonnes <INTITULE,debut,fin,THEME> et **ordonnées** par valeurs croissantes de <INTITULE,debut>, ce qui s'obtient par la requête :

   ```
   select INTITULE,debut,fin,THEME
   from   H_PROJET
   order by INTITULE,debut;
   ```

   Le résultat de cette requête est illustré à la Figure 13. Les états d'un même projet se présentent par ordre chronologique, ce qui fait apparaître les suites maximales.

2. La lecture séquentielle de ce résultat permet de reconstituer les états synthétiques par fusion des suites maximales.

Nous construirons une procédure dénommée HPROJECT_PRO_THEME qui range dans la table H_PRO_THEME la projection temporelle de H_PROJET sur [INTITULE,debut,fin,THEME].

La procédure lit successivement les lignes des états triés, et exécute le corps de la boucle pour chacune d'elles. On considère les cas suivants :

```
1. la ligne lue est la première : initialier une suite
2. la ligne lue n'est pas la première : la comparer à
   la suite courante
   2.1 même INTITULE : on reste dans le même projet
       2.1.1 état jointif
          2.1.1.1 même THEME : allonger la période de la
                   suite courante
          2.1.1.2 THEME différent : écrire suite courante,
                                    initialiser une suite
       2.1.2 état manquant : écrire suite courante,
                             écrire suite vide (éventuellement),
                             initialiser une suite
   2.2 INTITULE différent : écrire la suite courante,
                            initialiser une suite
```

Au sortir de la boucle, s'il y a eu au moins une itération, on écrit l'état synthétique formé par la suite courante.

Cette procédure est plus tolérante que la version prédicative car elle accepte les historiques incomplets dans lesquels certains projets peuvent avoir des états intermédiaires manquants. Lorsqu'un tel *trou* est identifié, la procédure peut (instruction en gras, facultative) introduire dans le résultat un état vide, c'est-à-dire une ligne dont THEME = null. Dans ce cas, le résultat est un historique complet.

```
create table H_PRO_THEME(
 INTITULE char(12) not null,
 debut    integer  not null,
 fin      integer  not null,
 THEME    char(22)        );
/* Procedure HPROJECT_PRO_THEME :
           H_PRO_THEME <- H_PROJET[INTITULE,debut,fin,THEME]          */
/* ------------------------------------------------------------------ */
/*    en gras : création d'un état dont THEME = null en cas d'état  manquant     */

create procedure HPROJECT_PRO_THEME
as
declare variable I char(12);          /* dernière ligne extraite */
declare variable d decimal(5);
declare variable f decimal(5);
declare variable T char(22);
declare variable curI char(12);       /* état suite courante   */
declare variable curd decimal(5);
declare variable curf decimal(5);
declare variable curT char(22);
declare variable iter integer; /* indicateur d'itérations : 0=aucune, 1=au moins
une   */
begin
  delete from H_PRO_THEME;
  iter = 0;
  for select INTITULE,debut,fin,THEME
      from   H_PROJET
      order by INTITULE,debut
      into :I, :d, :f, :T
  do
  begin
    if (iter = 0) then      /* premier passage */
```

```
      begin
        curI=I; curd=d; curf=f; curT=T;  /* initial. première suite */
        iter = 1;
      end
    else                    /* passages suivants: une suite est en cours */
      if (curI = I) then     /* on reste dans le même projet      */
         if (curf = d) then   /* état jointif: historique complet */
            if (curT = T) then  /* on reste dans la même suite */
               curf = f;              /* intégrer à la suite courante */
            else                /* on démarre une nouvelle suite du projet */
               begin
                 insert into H_PRO_THEME values
                 (:curI,:curd,:curf,:curT);  /* écrire suite courante */
                 curI=I; curd=d; curf=f; curT=T;/* initial. nouvelle suite*/
               end
         else                   /* état manquant: historique incomplet */
            begin
               insert into H_PRO_THEME values
               (:curI,:curd,:curf,:curT); /* écrire suite courante */
               insert into H_PRO_THEME(INTITULE,debut,fin) values
               (:curI,:curf,:d);          /* écrire état vide */
               curI=I; curd=d; curf=f; curT=T; /* initial. nouvelle suite */
            end
      else                     /* on démarre un nouveau projet */
         begin
            insert into H_PRO_THEME values(:curI,:curd,:curf,:curT);
                                          /* écrire suite courante */
            curI=I; curd=d; curf=f; curT=T;    /* initial. nouvelle suite */
         end
  end
  if (iter = 1) then       /* table source non vide: existe une suite en cours */
    insert into H_PRO_THEME values(:curI,:curd,:curf,:curT);
                           /* écrire dernière suite */
end
```

Code 22 - Extraction de l'historique des thèmes des projets - Version procédurale.

En revanche, la procédure ne traite pas les historiques contenant des états non disjoints (voir à ce sujet la Section 8).

Remarque

On devrait en théorie créer une procédure pour chaque combinaison de colonnes d'attribut susceptible de faire l'objet d'une projection, ce qui ne serait guère réaliste. En pratique, on utilisera une interface avec le serveur SQL offrant une forme quelconque d'SQL dynamique, tel ESQL[21], ODBC[22] ou JDBC[23]. Dans ce cas, les

21. *Embedded SQL*, ou SQL incorporé dans un programme rédigé dans un langage standard (C, PASCAL, COBOL, etc). Dans sa variante DSQL (Dynamic SQL), les instructions SQL peuvent être construites dynamiquement. Dans ce cas, l'instruction est, lors de l'exécution du programme, successivement construite, compilée (préparée), liée aux variables du programmes et exécutée.
22. *Open DataBase Connectivity* : protocole proposé par Microsoft pour tenter de standardiser l'accès à différents moteurs SQL. Les instructions sont transmises au serveur sous la forme de chaînes de caractères, qui peuvent être construites dynamiquement.
23. Protocole similaire à ODBC, mais spécifique aux programmes Java.

requêtes SQL nécessaires seront construites au moment de l'exécution de la procédure, en fonction des besoins de l'opérateur spécifique de projection à exécuter.

Le langage procédural d'InterBase dans lequel notre prototype est développé n'offrant pas cette fonctionnalité, nous aurons à écrire autant de procédures de projection que l'exigera l'utilisateur. La situation est cependant moins grave qu'on pourrait le penser. En effet, la projection temporelle sur **n** colonnes d'attribut (obligatoires) est équivalente à la jointure temporelle des **n** projections temporelles sur chacune de ces colonnes[24].

Attention, cette dernière propriété n'est vraie que pour la variante particulière de projection considérée, qui conserve l'identifiant de la table source. Rappelons qu'une relation (ou *table* dont les lignes sont distinctes) ne peut être décomposée que selon une dépendance *fonctionnelle*[25].

8. Projection temporelle généralisée

Une projection temporelle qui ne reprendrait aucun identifiant de la table historique, et en particulier qui ne reprendrait pas l'identifiant primaire, constitué de l'identifiant d'entité (INTITULE ou CODE) accompagné de la colonne debut, poserait des problèmes plus complexes que ceux auxquels nous avons été confrontés jusqu'ici. Calculons par exemple la projection temporelle de H_EMPLOYE sur la colonne ADRESSE, qui devrait nous donner, pour chaque ville, les périodes durant lesquelles au moins un employé y a habité. La version naïve, qui s'exprimerait sous la forme

```
select ADRESSE,debut,fin
from   H_EMPLOYE
```

produirait un résultat correct, mais particulièrement difficile à interpréter (Figure 17, gauche), au lieu du résultat synthétique attendu (Figure 17, droite).

La transformation du résultat naïf en résultat correct est cependant plus complexe que dans la projection réduite envisagée jusqu'ici. En effet, non seulement le résultat naïf peut-il comporter des états jointifs identiques, des états manquants (des *trous*), mais en outre certains états relatifs à la même adresse *ne sont pas disjoints*. Les requêtes et algorithmes examinés ci-dessus sont donc tout-à-fait insuffisants pour traiter ce type de projection. Il est nécessaire de réexaminer le problème pour prendre en compte ces cas de figure nouveaux.

24. On pourrait aussi construire une procédure par table, regroupant les algorithmes pour chaque colonne de projection, et dont le paramétre est le nom de la colonne de projection.
25. Ou *multivaluée*, ou *de jointure*.

ADRESSE	debut	fin
Charleroi	120	9999
Genève	12	38
Genève	38	87
Genève	87	9999
Genève	76	97
Genève	111	9999
Genève	105	131
Grenoble	70	76
Grenoble	97	111
Grenoble	93	9999
Grenoble	93	105
Lille	52	87
Lille	87	99
Lille	99	9999
Mons	44	65
Mons	65	95
Mons	47	73
Mons	73	120
Paris	95	120
Paris	15	40
Paris	40	65
Paris	65	108
Paris	108	9999
Toulouse	82	93

$\Rightarrow$

ADRESSE	debut	fin
Charleroi	120	9999
Genève	12	9999
Grenoble	70	76
Grenoble	93	9999
Lille	52	9999
Mons	44	120
Paris	15	9999
Toulouse	82	93

Figure 17 - Projection temporelle généralisée : résultat de la requête naïve (à gauche) et résultat synthétique attendu (à droite).

8.1 Projection généralisée - Version prédicative

L'étude précédente n'a considéré que la résolution des états identiques jointifs d'une même entité. Dans une situation réelle (Figure 17 par exemple), on doit envisager qu'un instant quelconque de la vie d'une entité puisse tomber dans un *nombre* quelconque d'états enregistrés : *zéro* (états manquants), *un* (cas normal), *deux* (Section 7) ou ***plusieurs***.

C'est ce dernier cas que nous allons étudier dans cette section. Nous y incluerons évidemment le cas standard des états disjoints mais jointifs. Pour nous résumer, nous allons développer une requête et une procédure qui réduisent, sans perte d'information, un historique de manière telle *qu'il ne s'y trouve plus d'états identiques en recouvrement ou jointifs*.

Un état synthétique (fusionné) est formé à partir d'une suite maximale d'états identiques délimitée par deux états initiaux E1 et E2, non nécessairement distincts, tels que

1. E1 et E2 sont identiques;
2. E1 est antérieur à E2, c'est-à-dire que E1 *a démarré avant* (ou au même instant que) E2 et E2 *se termine après* (ou au même instant que) E1;

3. il n'existe pas d'état identique E0 qui toucherait E1 à gauche, c'est-à-dire qui aurait démarré avant E1 et qui serait non disjoint (ou qui serait jointif) avec E1;
4. il n'existe pas d'état identique E0 qui toucherait E2 à droite, c'est-à-dire qui se terminerait après E2 et qui serait non disjoint (ou qui serait jointif) avec E2;
5. il n'y pas d'instants E3 entre la fin de E1 et le début de E2 qui ne soient repris par un état identique à E1; cette condition en recouvre deux autres : pas d'états étrangers et pas d'états manquants entre E1 et E2;

Il est évident que c'est la dernière condition qui pose un problème de formulation. Nous adopterons l'idée suivante[26]. Supposons que E1 et E2 soient disjoints (Figure 18). S'il existe un état E3 identique à E1, qui se clôture entre la fin de E1 (incluse) et le début de E2 (exclu), alors il doit exister un autre état identique E4 qui chevauche la fin de E3 et donc qui le prolonge vers la droite.

Etant donné la structure d'SQL, on exprimera cette relation d'implication de la manière suivante : *il n'existe pas d'état E3 qui ne soit pas prolongé par un état E4*.

On remarquera quelques propriétés importantes de ces règles.

1. Si E1 et E2 ne sont pas disjoints, alors ils respectent trivialement la condition 4. Tel est le cas en particulier lorsqu'ils sont jointifs ou s'ils sont en recouvrement.
2. Un état isolé E respecte ces règles par construction, puisque E1 = E2 = E.
3. Si E4 ne se clôture pas pendant E2, ou au moment où ce dernier démarre, et donc couvre incomplétement l'espace vide entre E3 et E2, alors la règle concernant E3 s'applique aussi à E4, pour lequel il doit exister un état qui le prolonge, et ainsi de suite.
4. E1 obéit à la définition de E3 (E1 se termine à la fin de E1), ce qui permet de démarrer la règle lorsque E1 et E2 sont disjoints et non jointifs.

Remarquons encore que des doublons seront créés lorsque E1 et E2 ont même intervalle de validité. On veillera donc à ne pas oublier la clause distinct.

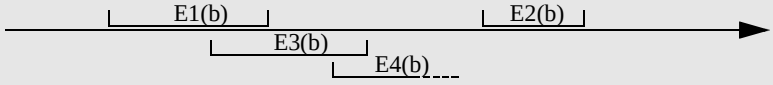


Figure 18 - Pour que l'espace entre E1 et E2 soit complètement couvert, il faut que tout état E3 qui tombe dans l'intervalle entre E1 et E2, sans atteindre E2, soit lui-même prolongé par un état E4. Tous ces états ont mêmes valeurs *b*.

De cette analyse, on tire la requête suivante, réalisant la projection temporelle de H_PROJET sur INTITULE et THEME[27]. (états identiques non disjoints ou jointifs, états manquants).

26. selon [Böhlen 1996]

```
select distinct E1.INTITULE, E1.debut, E2.fin, E1.THEME
from   H_PROJET E1, H_PROJET E2
where  E1.INTITULE=E2.INTITULE and E1.THEME=E2.THEME
and    E1.debut <= E2.debut and E1.fin <= E2.fin
and not exists (select *
                from   H_PROJET E0
                where  E0.INTITULE=E1.INTITULE
                       and E0.THEME=E1.THEME
                and    E0.debut < E1.debut
                and E0.fin >= E1.debut)
and not exists (select *
                from   H_PROJET E0
                where  E0.INTITULE=E1.INTITULE
                       and E0.THEME=E1.THEME
                and    E0.debut <= E2.fin
                and E0.fin > E2.fin)
and not exists (select *
                from   H_PROJET E3
                where  E3.INTITULE=E1.INTITULE
                       and E3.THEME=E1.THEME
                and    E1.fin <= E3.fin
                and    E3.fin <  E2.debut
                and    not exists(select *
                                  from   H_PROJET E4
                                  where  E4.INTITULE=E1.INTITULE
                                         and E4.THEME=E1.THEME
                                  and    E3.fin   <  E4.fin
                                  and    E4.debut <= E3.fin))
```

Code 23 - Requête qui construit la projection de H_PROJET sur INTITULE et THEME. Les états identiques d'une même entité sont fusionnés s'ils sont jointifs ou s'ils se recouvrent (sont non disjoints).

De cet exemple, nous pouvons tirer un canevas général de la projection temporelle sur un ensemble quelconque de colonnes, incluant ou non l'identifiant d'entité, et à partir de tables sources non nécessairement normalisées, c'est-à-dire dont les états d'une entité ne sont pas nécessairement jointifs (états manquants et/ou états en recouvrement). En adoptant les conventions suivantes :

- **TABLE** : désigne la table sur laquelle s'applique la projection**;**
- **ATTRIBUTS(Ei)** : désigne la liste des colonnes d'attribut de la projection pour l'état Ei;
- **EQU_ATTRIBUTS(Ei,Ej)** : désigne la condition d'égalité des états Ei et Ej pour toutes les colonnes d'attribut de la projection.

on dérive la requête générique Code 24.

27. Cette requête est évidemment trop puissante pour l'exemple choisi, puisque celui-ci travaille sur une table source normalisée et complète, et qu'en outre elle inclut l'identifiant d'entité.

```
select distinct ATTRIBUTS(E1), E1.debut, E2.fin
from   TABLE E1, TABLE E2
where  EQU_ATTRIBUTS(E1,E2)
and    E1.debut <= E2.debut and E1.fin <= E2.fin
and not exists (select *
                from   TABLE E0
                where  EQU_ATTRIBUTS(E0,E1)
                and    E0.debut < E1.debut
                and E0.fin >= E1.debut)
and not exists (select *
                from   TABLE E0
                where  EQU_ATTRIBUTS(E0,E1)
                and    E0.debut <= E2.fin
                and E0.fin > E2.fin)
and not exists (select *
                from   TABLE E3
                where  EQU_ATTRIBUTS(E3,E1)
                and    E1.fin <= E3.fin
                and    E3.fin <  E2.debut)
                and    not exists(select *
                                  from   TABLE E4
                                  where  EQU_ATTRIBUTS(E4,E1)
                                  and    E3.fin   <  E4.fin
                                  and    E4.debut <= E3.fin)
```

Code 24 - Canevas de la projection temporelle généralisée - Version prédicative.

A titre d'exemple, la projection de H_EMPLOYE sur ADRESSE (Figure 17) s'obtiendrait par les substitution suivantes :

- **TABLE** ⟶ H_EMPLOYE**;**
- **ATTRIBUTS(E1)** ⟶ E1.ADRESSE;
- **EQU_ATTRIBUTS(Ei,Ej)** ⟶ Ei.ADRESSE = Ej.ADRESSE.

Une solution élégante au problème de la complexité des expressions de projection consiste à définir une **vue** par projection. On proposera par exemple la définition représentée par le Code 25, qu'on appliquera à la projection selon les autres colonnes. Nous avons donné à la vue le nom de la table qui recevait le résultat dans l'approche procédurale. Cette convention permet de minimiser l'impact d'une modification dans le choix de la technique de réalisation des projections.

```
create view H_PRO_THEME(INTITULE,debut,fin,THEME)
as select distinct E1.INTITULE, E1.debut, E2.fin, E1.THEME
   from   H_PROJET E1, H_PROJET E2
   where  E1.INTITULE=E2.INTITULE and E1.THEME=E2.THEME
   etc. (cfr Code 23)
```

Code 25 - Vue exprimant la projection de H_PROJET sur INTITULE et THEME.

8.2 Projection généralisée - Version procédurale

Tout comme pour la projection simplifiée, l'approche procédurale est plus simple et plus efficace que l'approche prédicative. Appliquée une fois encore à la projection de H_PROJET sur INTITULE et THEME, elle fonctionne comme suit.

1. On ordonne les lignes de H_PROJET par valeurs croissantes de <INTITULE,THEME,debut>, de manière à grouper les états identiques et à les présenter par dates de debut croissantes[28]. Ce qui s'obtient par la requête :

```
select INTITULE,debut,fin,THEME
from   H_PROJET
order by INTITULE,THEME,debut;
```

2. Les états identiques apparaissent successivement par ordre chronologique de démarrage, ce qui permet de reconstituer les états synthétiques à partir des suites maximales lors d'une simple lecture séquentielle.

Nous redévelopperons la procédure HPROJECT_PRO_THEME qui range dans la table H_PRO_THEME la projection de H_PROJET sur INTITULE et THEME. Contrairement à la version précédente, les états en recouvrement et les états manquants sont pris en compte. Les états résultants sont donc **disjoints** et **distincts**.

La procédure lit successivement les lignes des états triés, et exécute le corps de la boucle pour chacune d'elles. On considère les cas suivants :

```
1. la ligne lue est la première : initialier une suite
2. la ligne lue n'est pas la première : la comparer à la
   suite courante
   2.1 mêmes valeurs de <INTITULE,THEME>
       et debut antérieur ou égal à fin de la suite courante :
                        allonger la période de la suite courante
   2.2 valeurs différentes de <INTITULE,THEME>
       ou debut postérieur à fin de la suite courante :
                              écrire la suite courante,
                              initialiser une suite
```

Au sortir de la boucle, s'il y eu au moins une itération : on écrit la suite courante.

```
/* Procedure HPROJECT_PRO_THEME: projection temporelle généralisée */
/* ------------------------------------------------------------------------------- */

create procedure HPROJECT_PRO_THEME
as
declare variable I char(12);      /* dernière ligne extraite */
declare variable d decimal(5);
declare variable f decimal(5);
declare variable T char(22);
declare variable curI char(12);   /* état suite courante   */
declare variable curd decimal(5);
declare variable curf decimal(5);
```

28. Le composant debut de la clé de tri n'est pas indispensable, son traitement pouvant être pris en charge par le corps de la procédure.

```
declare variable curT char(22);
declare variable iter integer;    /* indicateur d'itérations : 0=aucune
                                                               1=au moins 1 */
begin
  delete from H_PRO_THEME;
  iter = 0;
  for select INTITULE,debut,fin,THEME
      from   H_PROJET
      order by INTITULE,THEME,debut
      into :I, :d, :f, :T
  do
  begin
    if (iter = 0) then       /* premier passage */
      begin
        curI=I; curd=d; curf=f; curT=T; /* initial. première suite */
        iter = 1;
      end
    else                     /* passages suivants: une suite est en cours */
      if (curI=I and curT=T and d<=curf) then  /*  même suite  */
         begin if (f>curf) then curf=f; end
      else
         begin                                  /* rupture de séquence*/
            insert into H_PRO_THEME values(:curI,:curd,:curf,:curT);
            curI=I; curd=d; curf=f; curT=T;
         end
  end
  if (iter = 1) then       /* écrire dernière suite */
    insert into H_PRO_THEME values(:curI,:curd,:curf,:curT);
end
```

Code 26 - Procédure qui construit la projection de H_PROJET sur INTITULE et THEME : les états identiques d'une même entité sont fusionnés s'ils sont jointifs ou s'ils se recouvrent (sont non disjoints).

Nous pouvons également tirer de cet exemple un canevas général de procédure de projection. Nous adoptons les conventions suivantes :

- **TABLE** : désigne la table sur laquelle s'applique la projection**;**
- **RESULTAT** : désigne la table qui doit contenir le résultat de la projection**;**
- **SQL_COL** : déclaration SQL des colonnes d'attribut de la projection**;**
- **LISTE_COL** : liste des noms des colonnes d'attribut de la projection**;**
- **DECLARE_VAR_LIGNE** : déclaration dans le langage procédural des variables correspondant aux colonnes de projection pour la ligne courante;
- **DECLARE_VAR_SUITE** : déclaration dans le langage procédural des variables correspondant aux colonnes de projection pour la suite courante;
- **:VAR_LIGNE** : liste, selon les conventions SQL, des variables correspondant aux colonnes de projection pour la ligne courante;
- **:VAR_SUITE** : liste, selon les conventions SQL, des variables correspondant aux colonnes de projection pour la suite courante;
- **VAR_SUITE←VAR_LIGNE** : liste, selon le langage procédural, d'assignations des variables de la ligne courante aux variables de la suite courante;

- **VAR_SUITE=VAR_LIGNE** : expression booléenne exprimant l'égalité des variables de la suite courante et des variables de la ligne courante;

La requête générique est représentée par le Code 24.

```
create table RESULTAT(
 SQL_COL,
 debut    integer not null,
 fin      integer not null);


create procedure HPROJECT_XXXXX
as
DECLARE_VAR_LIGNE;            /* dernière ligne extraite */
declare variable d decimal(5);
declare variable f decimal(5);
DECLARE_VAR_SUITE;             /* état suite courante   */
declare variable curd decimal(5);
declare variable curf decimal(5);
declare variable iter integer;       /* indicateur d'itérations : 0=aucune
                                                                  1=au moins 1*/
begin
  delete from RESULTAT;
  iter = 0;
  for select LISTE_COL,debut,fin
      from   TABLE
      order by LISTE_COL,debut
      into :VAR_LIGNE, :d, :f
  do
  begin
    if (iter = 0) then       /* premier passage */
      begin
        VAR_SUITE←VAR_LIGNE; curd=d; curf=f; /* initial. première suite */
        iter = 1;
      end
    else                     /* passages suivants: une suite est en cours */
      if (VAR_SUITE=VAR_LIGNE and d<=curf) then  /*  même suite  */
         if (f>curf) then curf=f;
      else
         begin                                  /* rupture de séquence*/
            insert into RESULTAT values(:VAR_SUITE,:curd,:curf);
            VAR_SUITE←VAR_LIGNE; curd=d; curf=f;
         end
  end
  if (iter = 1) then       /* écrire dernière suite */
    insert into RESULTAT values(:VAR_SUITE,:curd,:curf);
end
```

Code 27 - Canevas de la projection généralisée - Version procédurale.

A titre d'exercice, le lecteur construira la procédure qui calcule la projection de la table H_EMPLOYE sur ADRESSE.

Remarques

1. Les erreurs d'un historique corrompu (dans lequel une entité peut avoir plusieurs états simultanés, voir Section 11.1) sont conservées et ne perturbent pas l'algorithme.

2. De même que nous l'avons fait pour la projection simplifiée, cet algorithme peut être aisément modifié de manière à ajouter des états vides pour chaque état manquant d'une entité. Il suffit de créer un état dont toutes les colonnes d'attributs (sauf l'identifiant d'entité) sont à `null` lorsque l'état qui vient d'être lu appartient à l'entité de la séquence courante mais est postérieur à celle-ci (`curI=I and curf<d`). Bien entendu, cette fonction n'est pertinente que si l'identifiant d'entité est présent parmi les colonnes de projection.
3. Certains SGBD permettent de définir des vues procédurales, c'est-à-dire des vues dont la définition est une procédure qui calcule les lignes successives. Tel est le cas d'InterBase. Il est alors possible de réconcilier les approches prédicatives et procédurales en ce qui concerne l'interface de programmation avec la base de données.

9. Jointure de tables historiques

La jointure entre deux tables non temporelles, telles que EMPLOYE et PROJET (Figure 2), s'exprime comme suit :

```
select CODE,NOM,STATUT,ADRESSE,PROJET,THEME,BUDGET
from   EMPLOYE E,PROJET P
where  E.PROJET = P.INTITULE
```

L'historique de cette jointure au cours du temps s'obtient par une *jointure temporelle*, appliquée aux deux tables H_EMPLOYE et H_PROJET. Nous limiterons la discussion à la jointure de deux historiques, les autres jointures pouvant être effectuées par une succession de jointures binaires.

Ecartons d'emblée l'expression naïve suivante :

```
select CODE,E.debut,E.fin,NOM,...,PROJET,THEME,BUDGET
from   H_EMPLOYE E,H_PROJET P
where  E.PROJET = P.INTITULE
and    E.debut = P.debut;
```

qui fournirait les couples d'états ayant débuté au même instant, correspondant à des événements exceptionnels sans réel intérêt (Figure 19), et comportant des erreurs (il est inexact que le budget du projet SURVEYOR ait été de 310.000 entre 80 et 108).

CODE	debut	fin	NOM	...	PROJET	THEME	BUDGET
D122	47	73	Declercq		BIOTECH	Génie génétique	120.000
A68	65	95	Albert		SURVEYOR	Surv. par satellite	310.000
M158	65	108	Mercier		SURVEYOR	Surv. par satellite	310.000
G96	87	9999	Godin		AGRO-2000	Amélior. céréales	65.000
C45	87	99	Carlier		AGRO-2000	Amélior. céréales	65.000

Figure 19 - Une *fausse* jointure.

9.1 Analyse de la jointure temporelle

Observons d'abord que, de même qu'une jointure entre EMPLOYE et PROJET nous renseigne prioritairement sur les employés (et non sur les projets), une jointure entre H_EMPLOYE et H_PROJET devrait nous informer sur les états historiques des employés, étendus aux informations sur leurs projets durant la période correspondant à chacun de ces états.

Considérons à titre d'exemple l'état suivant extrait de H_EMPLOYE, et dont l'intervalle de validité est [40,65). Cet état référence le projet BIOTECH pendant cette même période.

CODE	debut	fin	NOM	STATUT	ADRESSE	PROJET
M158	40	65	Mercier	P	Paris	BIOTECH

A cette ligne correspondent les trois lignes de la table H_PROJET relatives au projet BIOTECH et dont l'intervalle de validité partage au moins un instant avec l'intervalle [40,65).

INTITULE	debut	fin	THEME	BUDGET
BIOTECH	10	41	Biotechnologie	180.000
BIOTECH	41	47	Génie génétique	160.000
BIOTECH	47	84	Génie génétique	120.000

Le couplage de la ligne de H_EMPLOYE avec ces trois lignes de H_PROJET nous donne le résultat suivant (on notera que les valeurs minimale de debut et maximale de fin ont été ajustées[29]).

CODE	debut	fin	NOM	...	PROJET	THEME	BUDGET
M158	**40**	41	Mercier	...	BIOTECH	Biotechnologie	180.000
M158	41	47	Mercier	...	BIOTECH	Génie génétique	160.000
M158	47	**65**	Mercier	...	BIOTECH	Génie génétique	120.000

La Figure 20 montre les relations temporelles qui associent ces états.

La mise en correspondance de deux lignes est basée sur le recouvrement (non disjonction) de leurs deux intervalles temporels

29. en anglais, *clipping*.

Figure 20 - La jointure de l'état E de H_EMPLOYE qui référence le projet **p** avec les états correspondants P1, P2 et P3 de ce projet dans H_PROJET produit les états EP1, EP2 et EP3.

Nous avons vu (Section 6.1) que les intervalles [d1,f1) et [d2,f2) sont non disjoints si
(d1 < f2) et (d2 < f1)

Leur intervalle commun se calcule comme suit :

[max(d1,d2), min(f1,f2))

Munis de ces règles, nous pouvons à présent préciser le fonctionnement de la jointure temporelle.

1. un état E de H_EMPLOYE et un état P de H_PROJET sont en correspondance si E.PROJET = P.INTITULE, et si leurs périodes de validité sont non disjointes, c'est à dire si

   ```
   (P.debut < E.fin)  et  (E.debut < P.fin)
   ```

   On peut ainsi écrire une première requête qui forme les couples d'états de périodes de validité non disjointes :

   ```
   select CODE,E.debut,E.fin,INTITULE,P.debut,P.fin
   from   H_EMPLOYE E, H_PROJET P
   where  E.PROJET = P.INTITULE
   and    (P.debut < E.fin) and (E.debut < P.fin)
   ```

2. chacun des couples ainsi formés produit une ligne de la jointure dont la période de validité est la suivante

   ```
   [max(P.debut,E.debut),min(P.fin,E.fin))
   ```

En SQL-92 (fonction case-when-else disponible[30]), la jointure temporelle s'exprime donc comme suit.

```
select CODE,INTITULE,
   case when P.debut > E.debut then P.debut else E.debut end,
   case when P.fin  < E.fin  then P.fin  else E.fin end,
   NOM,STATUT,ADRESSE,THEME,BUDGET
from   H_EMPLOYE E, H_PROJET P
```

```
where  E.PROJET = P.INTITULE
and    (P.debut < E.fin)  and  (E.debut < P.fin)
```

Code 28 - Jointure temporelle de H_EMPLOYE et H_PROJET (version 1).

En l'absence de fonction case-when-else, il faudra se résoudre à une formulation plus développée, qui distingue quatre cas de recouvrement des intervalles :

```
select CODE,INTITULE,E.debut,E.fin
       NOM,STATUT,ADRESSE,THEME,BUDGET
from   H_EMPLOYE E,H_PROJET P
where  E.PROJET = P.INTITULE
and    (P.debut <= E.debut)
and    (E.fin   <= P.fin)
   union
select CODE,INTITULE,E.debut,P.fin,
       NOM,STATUT,ADRESSE,THEME,BUDGET
from   H_EMPLOYE E,H_PROJET P
where  E.PROJET = P.INTITULE
and    (E.debut >= P.debut)
and    (P.fin   <  E.fin)
and    (E.debut <  P.fin)
   union
select CODE,INTITULE,P.debut,E.fin,
       NOM,STATUT,ADRESSE,THEME,BUDGET
from   H_EMPLOYE E,H_PROJET P
where  E.PROJET = P.INTITULE
and    (P.debut >  E.debut)
and    (E.fin   <  P.fin)
and    (P.debut <  E.fin)
   union
select CODE,INTITULE,P.debut,P.fin,
       NOM,STATUT,ADRESSE,THEME,BUDGET
from   H_EMPLOYE E,H_PROJET P
where  E.PROJET = P.INTITULE
and    (P.debut >  E.debut)
and    (P.fin   <= E.fin)
```

Code 29 - Jointure temporelle de H_EMPLOYE et H_PROJET (version 2).

Il est tentant de construire une vue qui livre le résultat de la jointure (Code 30). On prévoira une telle vue pour chaque clé étrangère temporelle.

30. En InterBase, l'ajout des fonctions imax et imin à la librairie des *User Defined Functions*, renvoyant respectivement le plus grand et le plus petit des deux entiers passés en arguments, permet de compenser la faiblesse du langage. La requête se traduit alors comme suit :

```
select E.INTITULE,
       imax(P.debut,E.debut) as debut,
       imin(P.fin,E.fin) as fin,
       THEME,BUDGET
from etc.
```

```
create view H_EMPLOYE_PROJET(CODE,INTITULE,debut,fin,
                             NOM,STATUT,ADRESSE,
                             THEME,BUDGET)
as select CODE,INTITULE,
   case when P.debut > E.debut then P.debut else E.debut end,
   case when P.fin   < E.fin   then P.fin   else E.fin end,
          NOM,STATUT,ADRESSE,THEME,BUDGET
   from   H_EMPLOYE E, H_PROJET P
   where  E.PROJET = P.INTITULE
   and    (P.debut < E.fin)  and  (E.debut < P.fin)
```

Code 30 - Vue correspondant à la jointure de H_EMPLOYE et H_PROJET.

Sur base du contenu des tables H_EMPLOYE et H_PROJET, le résultat de cette requête ou de cette vue se présenterait comme suit (Figure 21).

```
CODE INTITULE debut fin  NOM    STATUT ADRESSE   THEME               BUDGET
A237 AGRO-2000   93  114 Antoine   P   Grenoble  Amélior. céréales    65000
A237 AGRO-2000  114  125 Antoine   P   Grenoble  Amélior. céréales    75000
A237 AGRO-2000  125 9999 Antoine   P   Grenoble  Amélior. céréales    82000
A68  BIOTECH     44   47 Albert    P   Mons      Génie génétique     160000
A68  BIOTECH     47   65 Albert    P   Mons      Génie génétique     120000
A68  SURVEYOR    65   80 Albert    P   Mons      Surv. par satellite 310000
A68  SURVEYOR    80   95 Albert    P   Mons      Surv. par satellite 375000
A68  SURVEYOR    95  101 Albert    P   Paris     Surv. par satellite 375000
A68  SURVEYOR   101  120 Albert    P   Paris     Surv. par satellite 345000
C45  AGRO-2000   87   99 Carlier   T   Lille     Amélior. céréales    65000
C45  BIOTECH     52   84 Carlier   T   Lille     Génie génétique     120000
C45  BIOTECH     84   87 Carlier   T   Lille     Génie génétique     140000
C45  BIOTECH     99  135 Carlier   T   Lille     Génie génétique     140000
C45  BIOTECH    135 9999 Carlier   T   Lille     Biotechnologie      140000
D107 BIOTECH    111  135 Delecourt P   Genève    Génie génétique     140000
D107 BIOTECH    135 9999 Delecourt P   Genève    Biotechnologie      140000
D107 SURVEYOR    70   76 Delecourt T   Grenoble  Surv. par satellite 310000
D107 SURVEYOR    76   80 Delecourt T   Genève    Surv. par satellite 310000
D107 SURVEYOR    80   97 Delecourt T   Genève    Surv. par satellite 375000
D107 SURVEYOR    97  101 Delecourt P   Grenoble  Surv. par satellite 375000
D107 SURVEYOR   101  111 Delecourt P   Grenoble  Surv. par satellite 345000
D122 AGRO-2000  120  125 Declercq  P   Charleroi Amélior. céréales    75000
D122 AGRO-2000  125 9999 Declercq  P   Charleroi Amélior. céréales    82000
D122 BIOTECH     47   73 Declercq  P   Mons      Génie génétique     120000
D122 SURVEYOR    73   80 Declercq  P   Mons      Surv. par satellite 310000
D122 SURVEYOR    80  101 Declercq  P   Mons      Surv. par satellite 375000
D122 SURVEYOR   101  120 Declercq  P   Mons      Surv. par satellite 345000
G96  AGRO-2000   87  114 Godin     P   Genève    Amélior. céréales    65000
G96  AGRO-2000  114  125 Godin     P   Genève    Amélior. céréales    75000
G96  AGRO-2000  125 9999 Godin     P   Genève    Amélior. céréales    82000
G96  BIOTECH     12   38 Godin     T   Genève    Biotechnologie      180000
G96  BIOTECH     38   41 Godin     P   Genève    Biotechnologie      180000
G96  BIOTECH     41   47 Godin     P   Genève    Génie génétique     160000
G96  BIOTECH     47   84 Godin     P   Genève    Génie génétique     120000
G96  BIOTECH     84   87 Godin     P   Genève    Génie génétique     140000
M158 BIOTECH     15   40 Mercier   T   Paris     Biotechnologie      180000
M158 BIOTECH     40   41 Mercier   P   Paris     Biotechnologie      180000
M158 BIOTECH     41   47 Mercier   P   Paris     Génie génétique     160000
M158 BIOTECH     47   65 Mercier   P   Paris     Génie génétique     120000
M158 BIOTECH    108  135 Mercier   P   Paris     Génie génétique     140000
M158 BIOTECH    135 9999 Mercier   P   Paris     Biotechnologie      140000
M158 SURVEYOR    65   80 Mercier   P   Paris     Surv. par satellite 310000
M158 SURVEYOR    80  101 Mercier   P   Paris     Surv. par satellite 375000
M158 SURVEYOR   101  108 Mercier   P   Paris     Surv. par satellite 345000
N240 AGRO-2000  105  114 Nguyen    T   Genève    Amélior. céréales    65000
```

```
N240 AGRO-2000  114  125 Nguyen    T   Genève    Amélior. céréales    75000
N240 AGRO-2000  125  131 Nguyen    T   Genève    Amélior. céréales    82000
N240 BIOTECH     82   84 Nguyen    T   Toulouse  Génie génétique     120000
N240 BIOTECH     84   93 Nguyen    T   Toulouse  Génie génétique     140000
N240 SURVEYOR    93  101 Nguyen    T   Grenoble  Surv. par satellite 375000
N240 SURVEYOR   101  105 Nguyen    T   Grenoble  Surv. par satellite 345000
```

Figure 21 - Résultat de la jointure de H_EMPLOYE avec H_PROJET.

Remarques

1. Nous avons basé notre raisonnement sur les propriétés d'un type particulier de jointure, où le couplage (critère de jointure) s'effectue entre un *identifiant* et une *clé étrangère*. En fait, toute jointure temporelle, basée sur un ensemble quelconque de colonnes, peut être calculée sur le modèle des requêtes proposé ci-dessus.
2. L'instantané au temps T (valeur de la variable T) d'une jointure temporelle est obtenu comme suit :

```
select CODE,INTITULE,NOM,STATUT,ADRESSE,THEME,BUDGET
from   H_EMPLOYE E,H_PROJET P
where  E.PROJET = P.INTITULE
and    (P.debut <= :T) and (:T < E.fin)
and    (E.debut <= :T) and (:T < P.fin)
```

Code 31 - Jointure *non temporelle* de H_EMPLOYE et H_PROJET à l'instant T.

10. L'agrégation temporelle

En toute généralité, une requête d'agrégation portant sur une table non temporelle fournit une ou plusieurs grandeurs numériques calculées sur un ensemble de lignes de cette table (Figure 22, gauche), ou pour chaque groupe de lignes constitué selon un critère de groupement (Figure 22, droite).

```
select sum(BUDGET)          select PROJET, count(*)
from   PROJET               from   EMPLOYE
                            group by PROJET
```

Figure 22 - Les deux formes de la requête d'agrégation : *donner la somme des budgets des projets* (gauche); *donner, pour chaque projet, le nombre d'employés* (droite).

On conçoit rapidement l'intérêt de généraliser les requêtes d'agrégation aux données historiques, de manière à obtenir l'évolution temporelle de ces grandeurs.

10.1 Agrégation temporelle : analyse préliminaire

Calculons par exemple **l'évolution du nombre d'employés par projet**. Pour y voir plus clair, nous raisonnerons sur la projection temporelle de H_EMPLOYE sur CODE et PROJET (Figure 23). Cette table, qu'on nommera H_EMP_PRO, indique, pour

chaque employé, ses périodes d'occupation sur chacun des projets sur lesquels il a travaillé, les périodes jointives ayant été fusionnées (par *coalescing*).

H_EMP_PRO			
CODE	**debut**	**fin**	**PROJET**
C45	87	99	AGRO-2000
G96	87	9999	AGRO-2000
A237	93	9999	AGRO-2000
N240	105	131	AGRO-2000
D122	120	9999	AGRO-2000
...	...	...	...

Figure 23 - Projection temporelle de H_EMPLOYE sur CODE et PROJET (limitée au projet AGRO-2000).

Représentons ces périodes sur un diagramme temporel, tel que celui de la Figure 24. On y repère clairement tous les événements liés aux employés et qui affectent le projet AGRO-2000. Leur report sur une même ligne du temps nous fournit un historique unique de ces événements, qui est aisé à calculer (Code 32).

```
create table EVENEMENTS
     (PROJET   char(12)   not null,
      temps    decimal(5) not null);

insert into EVENEMENTS
select PROJET,debut from H_EMP_PROJET
union
select PROJET,fin from H_EMP_PROJET;
```

Code 32 - Calcul de l'historique des événements des employés pour chaque projet (selon ligne du temps de la Figure 24)[31].

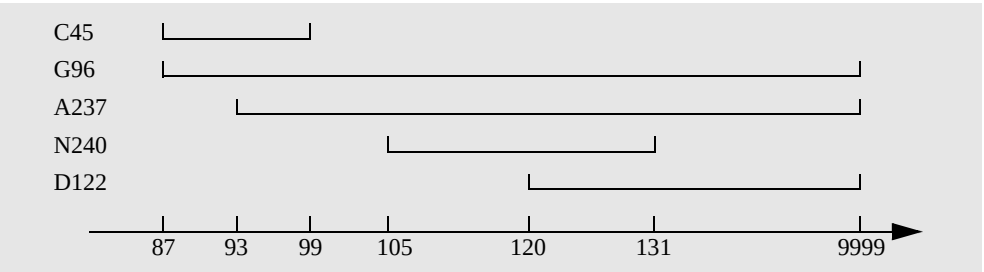


Figure 24 - L'historique des événements relatifs aux employés, et qui affectent le projet AGRO-2000, selon la Figure 23.

31. En InterBase, qui refuse l'*union* dans un *insert*, on utilisera deux *insert* successifs, puis on éliminera les doubles.

L'examen de la Figure 24 nous montre aussi que les couples successifs des événements (<87,93>, <93,99>, etc.) correspondent à des périodes de stabilité, durant lesquelles le nombre d'employés reste constant. Une table de ces états stables (Figure 25) peut être construite via le Code 33.

ETATS_STABLES		
PROJET	**debut**	**fin**
AGRO-2000	87	93
AGRO-2000	93	99
AGRO-2000	99	105
AGRO-2000	105	120
AGRO-2000	120	131
AGRO-2000	131	9999
...	...	...

Figure 25 - Les états stables des projets quant au nombre d'employés.

```
create table ETATS_STABLES
     (PROJET   char(12)   not null,
      debut    decimal(5) not null,
      fin      decimal(5) not null);

insert into ETATS_STABLES(PROJET,debut,fin)
select V1.PROJET, V1.temps, V2.temps
from   EVENEMENTS V1, EVENEMENTS V2
where  V1.PROJET = V2.PROJET
and    V2.temps = (select min(temps)
                   from   EVENEMENTS
                   where  PROJET = V1.PROJET
                   and    temps  > V1.temps);
```

Code 33 - Calcul des périodes constituées des couples d'événements successifs de l'historique de la Figure 24 - Version prédicative

Il reste ensuite à calculer, pour chacune de ces périodes de stabilité, le nombre d'états de H_EMP_PRO qui la contiennent. Ceci s'obtient par une agrégation classique sur {PROJET,debut} de la jointure entre H_EMP_PRO et ETATS_STABLES (Code 34).

```
select PROJET,debut,fin,count(*)
from   H_EMP_PRO P, ETATS_STABLES S
where  P.PROJET = S.PROJET
and    P.debut <= S.debut
and    S.debut < P.fin;
group by PROJET,debut,fin
```

Code 34 - Calcul de l'agrégation temporelle donnant le *nombre d'employés pour chaque état stable de chaque projet*. On notera que le critère d'agrégation inclut aussi fin, SQL2 exigeant que les colonnes citées dans la clause *select* apparaissent aussi dans la clause *group by*.

PROJET	debut	fin	NBRE
AGRO-2000	87	93	2
AGRO-2000	93	99	3
AGRO-2000	99	105	2
AGRO-2000	105	120	3
AGRO-2000	120	131	4
AGRO-2000	131	9999	3
...	...	...	

Figure 26 - Agrégation temporelle : le résulat final.

10.2 Agrégation temporelle : étude approfondie

La discussion précédente nous a permis de comprendre, à partir d'un exemple simple, comment l'agrégation temporelle pouvait se réduire à une agrégation classique. Elle va nous permettre de construire une méthode d'agrégation temporelle à la fois plus générale et plus efficace.

Nous pouvons d'emblée faire quelques observations.

- Le résultat final peut inclure des états identiques jointifs. En effet, il se peut qu'au même instant un employé quitte un projet et qu'un autre y rentre, ce qui entraîne la production de deux états jointifs identiques. Il est donc nécessaire de procéder à un *coalescing final*.
- En revanche, la projection initiale de H_EMPLOYE sur CODE et PROJET n'est pas nécessaire. En travaillant directement sur H_EMPLOYE plutôt que sur H_EMP_PRO, on répertoriera un plus grand nombre d'événements dans EVENEMENTS et donc d'états stables dans ETATS_STABLES. Le seul inconvénient éventuel (à évaluer dans chaque cas) serait un coût plus élevé lorsque le critère d'agrégation est particulièrement stable au regard des autres colonnes de la table, et donc, dans notre cas, lorsque le nombre de lignes de H_EMPLOYE est beaucoup plus grand que celui de H_EMP_PRO.
- Un bref examen de la requête 33 montre que le calcul des états stables (ETATS_STABLES) sera plus rapide pour de grandes tables si on adopte une approche procédurale.
- Le procédé suggéré pour l'évaluation du count n'est pas directement applicable aux autres fonctions agrégatives, telles que sum, ou avg (requête 34 par exemple), qui travaillent sur une propriété spécifique, généralement numérique, des entités interrogées. Nous utiliserons une nouvelle table (ETATS_STABLES_VALUES) qui contiendra le résultat de la jointure calculée par la requête 34, où chaque ligne représente un état stable d'une entité de la table source, accompagné de la valeur (VALEUR) dont on mesure une ou plusieurs

statistiques. Les fonctions agrégatives seront évaluées sur la colonne VALEUR de cette table.

Nous procéderons en quatre phases, que nous développerons sur un exemple différent du précédent : *calculer l'évolution dans le temps de la moyenne des salaires des employés par projet*[32]. La table décrite à la Figure 5 doit comporter une nouvelle colonne, SALAIRE, qui représente le salaire de l'employé durant la période de l'état.

1. Calcul des états stables pour l'ensemble des projets (résultat dans ETATS_STABLES).
2. Calcul des états stables de chaque projet, accompagnés de la valeur dont on désire calculer la statistique (résultat dans ETATS_STABLES_VALUES).
3. Calcul de la statistique (résultat dans STATISTIQUE_BRUTE).
4. Synthèse ou coalescing (résultat dans STATISTIQUE)[33].

Nous verrons que cette décomposition offre une grande souplesse pour résoudre des problèmes plus complexes.

Nous aurons besoin de tables d'accueil des données calculées successivement :

```
create table EVENEMENTS
     (AGREGAT  char(12)   not null,
      temps    decimal(5) not null);

create table ETATS_STABLES
     (AGREGAT  char(12)   not null,
      debut    decimal(5) not null,
      fin      decimal(5) not null);

create table ETATS_STABLES_VALUES
     (AGREGAT  char(12)   not null,
      debut    decimal(5) not null,
      fin      decimal(5) not null,
      VALEUR   decimal(5));

create table STATISTIQUE_BRUTE
     (AGREGAT  char(12)   not null,
      debut    decimal(5) not null,
      fin      decimal(5) not null,
      VALEUR   decimal(5));

create table STATISTIQUE
     (AGREGAT  char(12)   not null,
      debut    decimal(5) not null,
      fin      decimal(5) not null,
```

32. Ce qui correspondrait, pour les états courants, à :
```
select PROJET, avg(SALAIRE)
from   H_EMPLOYE
group by PROJET
```
33. Sauf si l'agrégation est basée sur un calendrier ou sur des intervalles fixes, auquel cas la fusion des états identiques serait malvenue.

```
        VALEUR   decimal(5));
```

a) Phase 1 : états stables globaux

Tout comme dans l'analyse simplifiée, nous établirons d'abord une liste des événements élémentaires puis nous calculerons les états stables pour l'ensemble des projets. Pour des raisons d'efficacité, nous encapsulerons ces opérations dans une procédure (Code 35).

```
create procedure CALC_ETATS_STABLES_EMP_PRO
as
declare variable t decimal(5);
declare variable P char(12);
declare variable firstt decimal(5);
declare variable firstP char(12);
declare variable first_pass integer; /* première itération ?*/
begin
  /* rangement dans EVENEMENTS des événements des projets */
  delete from EVENEMENTS;
  insert into EVENEMENTS
  select PROJET, debut from H_EMPLOYE;
  insert into EVENEMENTS
  select PROJET, fin from H_EMPLOYE;

  /* rangement dans ETATS_STABLES des périodes de stabilité */
  delete from ETATS_STABLES;
  first_pass = 1;
  for select distinct AGREGAT,temps
      from EVENEMENTS
      order by AGREGAT,temps into :P,:t
  do
  begin
    if (first_pass = 1) then begin  /* première itération */
       firstt=t; firstP=P; first_pass=0;
     end
    else begin                      /* itérations suivantes */
       if (P = firstP) then begin      /* meme projet */
          insert into ETATS_STABLES values (:P,:firstt,:t);
          firstt=t;
       end
       else begin                      /* nouveau projet */
          firstt = t; firstP = P;
       end
    end
  end
end;
```

Code 35 - Procédure de calcul des états stables.

On notera que cette procédure dépend de la table source (H_EMPLOYE) et du critère d'agrégation (AGREGAT), bien que celui-ci puisse être constitué de toute colonne du type char(n), où n ≤ 12 (valeur d'ailleurs arbitraire). En revanche, elle ne dépend pas des grandeurs sur lesquelles la statistique est à calculer. Elle conviendra tout aussi bien pour compter les employés des projets que pour calculer la moyenne des salaires.

La première phase consiste donc à appeler cette procédure :

```
execute procedure CALC_ETATS_STABLES_EMP_PRO;
```

b) Phase 2 : états stables individuels valués

Nous allons à présent introduire dans la table ETATS_STABLES_VALUES une instance de chaque ligne de ETATS_STABLE pour chaque employé dont un état dans H_EMPLOYE inclut la période de cette ligne. Au passage, on associe à cette instance les valeurs à agréger. Si on se limite aux fonctions agrégatives à argument numérique, une seule forme de table sera nécessaire quelles que soient la fonction demandée et son argument. La procédure 36 réalise cette instanciation.

```
create procedure CALC_ETATS_STABLES_EMP_PRO_SAL
as
begin
   delete from ETATS_STABLES_VALUES;
   insert into ETATS_STABLES_VALUES
   select S.AGREGAT, S.debut, S.fin, E.SALAIRE
   from   ETATS_STABLES S, H_EMPLOYE E
   where  S.AGREGAT = E.PROJET
   and    S.debut >= E.debut
   and    S.debut <  E.fin;
end;
```

Code 36 - Construction des états stables individuels valués.

Cette deuxième phase consiste à invoquer cette procédure.

```
execute procedure CALC_ETATS_STABLES_EMP_PRO_SAL;
```

c) Phase 3 : agrégation et calcul statistique

Grâce aux phases préparatoires, l'agrégation proprement dite est réalisable à l'aide d'une requête classique (Code 37). Etant donné son caractère de généralité, nous en avons fait une procédure.

```
create procedure CALC_MOYENNE_ETATS_STABLES
as
begin
   insert into STATISTIQUE_BRUTE
```

```
    select AGREGAT, debut, fin, avg(VALEUR)
    from ETATS_STABLES_VALUES
    group by AGREGAT,debut,fin;
end;
```

Code 37 - Calcul de l'évolution temporelle du salaire mensuel moyen de chaque projet.

On appellera donc cette procédure :

```
execute procedure CALC_MOYENNE_ETATS_STABLES;
```

d) Phase 4 : coalescing

Plus encore que pour l'évaluation simplifiée (Section 10.1), on trouvera dans la table STATISTIQUE_BRUTE des suites d'états jointifs identiques. Ces suites étant indésirables (pas de calendrier ni d'intervalles fixes), on les fusionnera par une procédure de coalescing HPROJECT_STAT_VALEUR, qu'on rédigera sur le modèle de HPROJECT_PRO_THEME (Code 26). Cette procédure range dans STATISTIQUE la projection de STATISTIQUE_BRUTE sur <AGREGAT,VALEUR>.

Le coalescing est demandé comme suit :

```
execute procedure HPROJECT_STAT_VALEUR;
```

e) Agrégation : procédure complète

Pour résumer, on rappelle le script qui réalise le calcul de l'évolution temporelle de la moyenne des salaires mensuels des projets :

```
execute procedure CALC_ETATS_STABLES_EMP_PRO;
execute procedure CALC_ETATS_STABLES_EMP_PRO_SAL;
execute procedure CALC_MOYENNE_ETATS_STABLES;
execute procedure HPROJECT_STAT_VALEUR;
```

Code 38 - Script de calcul de l'agrégation.

10.3 Extensions

On observe que les requêtes agrégatives non temporelles proposées par SQL2 comportent des clauses que nous n'avons pas considérées dans cette analyse. Ainsi, nous avons ignoré les clauses `where` et `having`, ainsi que des variantes plus complexes des autres clauses. D'autre part, il convient de réintroduire l'évaluation de la fonction `count` dans cette procédure. Enfin, nous évoquerons l'application des fonctions agrégatives aux colonnes temporelles debut et fin.

a) select

L'étude ci-dessus concerne des requêtes qui produisent simplement le critère d'agrégation (p.ex. PROJET) et une statistique. Moyennant modification des procédures d'extraction (phase 2) et d'agrégation (phase 3), il est aisé d'inclure dans la clause select d'autres résultats tels que les suivants :

```
select PROJET, count(*),
       sum(SALAIRE), max(SALAIRE) - min(SALAIRE)
```

D'autre part, on peut reconstituer le résultat d'une telle requête par une jointure temporelle des résultats relatifs aux statistiques individuelles.

b) from

Nous avons raisonné sur le traitement d'une table source. Or, en toute généralité, celle-ci pourrait résulter d'une jointure. On observe que deux phases font référence à la table source, on propose donc de ne pas inclure cette jointure dans les procédures de ces phases, mais de travailler directement sur une table unique, résultant éventuellement d'une jointure. Deux techniques sont applicables : table réelle ou vue.

1. Le calcul porte sur une **table dérivée** dans laquelle on a rangé préalablement le résultat de la jointure. La jointure n'est évaluée qu'une seule fois et son résultat est traité deux fois.
2. Le calcul porte sur une **vue** qui livre le résultat de la jointure. Cette technique évite la création d'une table supplémentaire.

Les procédures CALC_ETATS_STABLES_EMP_PRO et CALC_ETATS_STABLES_EMP_PRO_SAL sont à réécrire en conséquence.

c) where

Une agrégation qui porte sur un sous-ensemble de lignes de la table source exigera un filtrage sur cette dernière. La question est assez similaire à celle de la jointure : on observe que deux phases font référence à la table source, ce qui suggère d'écarter l'idée d'inclure ce filtrage dans les procédures de ces phases au profit d'un travail direct sur les seules lignes concernées. Deux techniques donc : table réelle ou vue.

1. Le calcul porte sur une table dérivée dans laquelle on a rangé préalablement les lignes à traiter. Cette technique est efficace lorsqu'on ne travaille que sur un petit sous-ensemble (un accès à la table complète + deux accès à la table dérivée).
2. Le calcul porte sur une vue qui ne livre que les lignes sélectionnées. Cette technique évite la création d'une table supplémentaire, mais est moins efficace lorsque la proportion des lignes à traiter est importante (deux accès à la table complète).

Les procédures CALC_ETATS_STABLES_EMP_PRO et CALC_ETATS_STABLES_EMP_PRO_SAL sont à réécrire en conséquence.

d) having

Une agrégation portant sur un sous-ensemble seulement des groupes réclamera un filtrage de ceux-ci. Idéalement, c'est dans la procédure `CALC_MOYENNE_ETATS_STABLES` qu'on placera ce filtrage, exprimé par une clause `having` classique. Attention cependant : les informations citées dans cette clause doivent être présentes dans la table `ETATS_STABLES_VALUES`, qu'on étendra si nécessaire par une ou plusieurs colonnes supplémentaires.

e) count (*) et count(distinct)

La version généralisée de l'agrégation temporelle couvre les statistiques de comptage `count(*)`. En effet, il suffit, dans la procédure de la phase 2, d'assigner à la colonne `VALEUR` une valeur quelconque non `null` (entier 1 par exemple), et de calculer la statistique `count(*)` ou `sum(VALEUR)` dans la procédure de la phase 3.

Quand à la statistique `count(distinct <colonne>)`, elle se traitera d'une manière similaire.

f) agrégation de grandeurs temporelles

Les procédures développées jusqu'ici calculent des statistiques sur des colonnes d'attribut (à l'exception de `count(*)`), et il en est de même des critères d'agrégation. Rien n'empêche cependant de traiter les grandeurs temporelles debut et fin, ainsi que toute valeur qui en dérive, telles que fin - debut. La Section 6.2, bien que consacrée à des requêtes simples, en contient plusieurs illustrations.

11. Normalisation d'une table historique

La plupart des raisonnements que nous avons développés dans ce projet sont basés sur une forme assez stricte de l'historique des états d'une population d'entités, dénommée **historique normalisé complet** (Section 3.2). En particulier, nous avons considéré que les états d'une entité étaient jointifs, c'est-à-dire que ces états sont disjoints et leur séquence est sans *trous* (pas d'états manquants). L'une des conséquences de ces propriétés est que les colonnes (identifiant d'entité, debut) et (identifiant d'entité, fin) constituent deux identifiants de la table historique.

Nous avons cependant observé que certaines opérations produisaient des historiques incomplets ou non normalisés (Figure 17), soit à cause d'**états manquants**, soit par la présence d'**états identiques non disjoints**. Nous examinerons dans cette section quelques requêtes qui permettent d'évaluer le caractère normalisé d'une table historique. Cependant, nous reviendrons d'abord sur une forme particulière d'historique, l'*historique corrompu*.

11.1 Historique corrompu

Il s'agit d'un historique qui viole le principe qui veut qu'une entité ne puisse être dans plus d'un état à tout instant. Dans un historique corrompu, il existe au moins *deux états* ***distincts*** *et* ***non disjoints*** *de la* ***même entité***.

On repèrera ces **erreurs** par la requête Code 39. On notera qu'il est impossible de corriger automatiquement ce type d'erreur pour lequel un arbitrage est nécessaire. D'autre part, les mécanismes de gestion que nous avons développés garantissent que les historiques en seront exempts. A l'exception de la présente section, ce chapitre est basé sur l'hypothèse d'historique non corrompu.

```
select E1.INTITULE,E1.debut,E1.fin,E2.debut,E2.fin
from   H_PROJET E1, H_PROJET E2
where  E1.INTITULE=E2.INTITULE
and    not(E1.THEME=E2.THEME and E1.BUDGET=E2.BUDGET)
and    E1.debut < E2.fin and E2.debut < E1.fin
```

Code 39 - Recherche des états erronés dans H_PROJET, c'est-à-dire des états simultanés mais distincts pour une même entité[34].

11.2 Recherche des états manquants

Un état E_m est **manquant** s'il existe deux états E_1,E_2 de l'entité E tels que :

- E_2 a démarré après la fin de E_1, c'est-à-dire tel que

  $E_1.fin < E_2.debut$

- et il n'existe pas d'état E_3 de E qui aurait un ou des instants tombant dans l'intervalle entre E_1 et E_2, ou encore non disjoint de $[E_1.fin, E_2.debut)$, c'est-à-dire tel que

  $E_3.debut < E_2.debut$ et $E_1.fin < E_3.fin$

L'état manquant E_m a pour intervalle

$[E_1.fin, E_2.debut)$.

En fonction de ces règles, le repérage des états manquants est réalisé par la requête suivante, qu'on suppose destinée à traiter une table H_PROJET incomplète[35].

34. Comment pourrait-on éliminer les doublons (la clause distinct ne suffit pas) ?
35. Remarque à destination des amateurs d'ésotérisme : cette requête extrait de la table H_PROJET des données qui ne s'y trouvent pas !

```
select E1.INTITULE, E1.fin, E2.debut
from   H_PROJET E1, H_PROJET E2
where  E1.INTITULE=E2.INTITULE
and    E1.fin < E2.debut
and not exists (select *
                from   H_PROJET E3
                where  E3.INTITULE=E1.INTITULE
                and    E3.debut < E2.debut
                and    E1.fin < E3.fin)
```

Code 40 - Recherche des états manquants dans l'historique H_PROJET.

On dérive aisément de cette requête un mécanisme qui complète un historique partiel en y réintroduisant les états manquants. Il est cependant à noter qu'on ne peut retrouver les éventuels états manquants extrêmes de l'historique d'une entité, c'est-à-dire le premier et le dernier état.

Important. Cette règle et cette requête sont valables même si l'historique contient des états non disjoints. En effet, il suffit qu'il existe un état non disjoint du *trou* supposé pour que les deux états E_1 et E_2 soient rejetés.

11.3 La projection comme opérateur de *coalescing*

La projection généralisée décrite à la Section 8 peut servir au *coalescing* d'un historique non normalisé, c'est-à-dire à la réduction des états identiques jointifs ou non disjoints. En effet, la projection effectue automatiquement cette réduction pour les colonnes de projection. Il suffit donc de projeter la table sur l'ensemble de ses colonnes d'attribut pour éliminer les anomalies mentionnées.

Partie 4

Compléments

12. Variantes d'historiques

Cette étude n'aura évidemment pas épuisé la question de la représentation de données historiques, loin s'en faut. Nous voudrions cependant évoquer deux autres techniques de représentation de l'historique d'évolution d'une population d'entités sous la forme de tables relationnelles : l'historique des attributs et l'historique des événements.

12.1 Historique des attributs

L'existence des opérateurs de projection et de jointure nous permet d'envisager une variante intéressante de représentation de données historiques concernant des entités, communément appelée **historique des attributs**. Considérons les deux tables de la Figure 27 qui ont été obtenues par l'application des procédures `HPROJECT_PRO_THEME` et `HPROJECT_PRO_BUDGET` (similaire à la première mais adaptée à la colonne BUDGET).

La jointure de ces deux tables selon la requête 41 reconstitue la table H_PROJET d'origine. Cette forme éclatée présente par rapport à la table H_PROJET des avantages qui sont à confronter à ses inconvénients :

H_PRO_THEME			
INTITULE	**debut**	**fin**	**THEME**
BIOTECH	10	41	Biotechnologie
BIOTECH	41	135	Génie génétique
BIOTECH	135	9999	Biotechnologie
AGRO-2000	87	9999	Amélior. céréales
SURVEYOR	65	120	Surv. par satellite

H_PRO_BUDGET			
INTITULE	**debut**	**fin**	**BUDGET**
BIOTECH	10	41	180.000
BIOTECH	41	47	160.000
BIOTECH	47	84	120.000
BIOTECH	84	9999	140.000
AGRO-2000	87	114	65.000
AGRO-2000	114	125	75.000
AGRO-2000	125	9999	82.000
SURVEYOR	65	80	310.000
SURVEYOR	80	101	375.000
SURVEYOR	101	120	345.000

Figure 27 - Eclatement selon chaque colonne d'attribut de l'historique H_PROJET.

avantages : lors d'une modification d'un attribut, seule la table qui représente l'historique de cet attribut est touchée; en outre, les valeurs qui n'ont pas changé ne sont pas dupliquées; l'extraction d'historiques partiels correspondant à une projection sur une colonne d'attribut ne nécessite plus de *coalescing*;

inconvénients : la reconstitution d'un état complet nécessite une ou plusieurs jointures; la modification de plus d'une colonne d'attribut entraîne des modifications dans autant de tables.

```
select E.INTITULE,
   case when P.debut > E.debut then P.debut else E.debut end,
   case when P.fin   < E.fin   then P.fin   else E.fin end,
       THEME,BUDGET
from   H_PRO_THEME E, H_PRO_BUDGET P
where  E.INTITULE = P.INTITULE
and    (P.debut < E.fin)  and  (E.debut < P.fin)
order by INTITULE, 2;
```

Code 41 - Reconstitution par jointure de l'historique complet H_PROJET à partir des historiques des attributs H_PRO_THEME et H_PRO_BUDGET.

Cette analyse n'est cependant valable que si certaines conditions sont satisfaites. Nous savons déjà que, selon la théorie relationnelle, la décomposition de H_PROJET en H_PRO_THEME et H_PRO_BUDGET est licite, puisque la décomposition est effectuée selon le déterminant d'une dépendance fonctionnelle (soit INTITULE, debut $\longrightarrow$ THEME). Mais il faut encore imposer qu'à chaque instant, toute entité qui a un état dans une des tables en a également un dans chacune des autres tables. A défaut de quoi la jointure ignorera ces états. Nous pouvons traduire cette contrainte de plusieurs manières :

1. soit on impose que tous les attributs du type d'entités représenté soient obligatoires; il s'agit de l'hypothèse simplificatrice que nous avons adoptée dans ce projet;
2. soit on permet que certains attributs soient facultatifs, mais quand un tel attribut est absent pour une entité, alors un état accompagné de la valeur `null` est introduit dans l'historique correspondant; l'historique d'un attribut reste donc complet et jointif;
3. soit on permet que certains attributs soient facultatifs, et on admet que les états correspondants soient manquants dans l'historique de cet attribut (historique non jointif ou *à trous*), mais alors on utilise une variante plus élaborée de l'opérateur de jointure qui considère que les états manquants correspondent à des valeurs `null`, comme dans le cas précédent[36]; notons que cette solution ne permet pas de reconstruire un état extrême (initial ou final) dont *tous les attributs (sauf l'identifiant d'entité), seraient absents*, une situation certes marginale mais techniquement possible.

12.2 Historique des événements

Il existe une forme duale de celle que nous avons analysée (représentation des *états*), à savoir la représentation des *événements* ou *transitions d'états*. En termes de données, on y enregistrera les opérations de création, de suppression et de modification d'entités plutôt que les états résultant de ces événements.

36. On considère donc une sorte de **jointure externe temporelle**.

Afin d'uniformiser le format, on conviendra de reprendre chaque colonne d'attribut et de lui donner la valeur suivante :

- **create** : celle de l'attribut au moment de la création de l'entité;
- **delete** : celle de l'attribut au moment de la suppression de l'entité;
- **update** : celle de l'attribut après modification; si l'attribut n'est pas concerné par cette modification, on conserve la valeur de la colonne correspondante.

La table de la Figure 28 représente les événements qui ont produit les états de l'historique H_PROJET (Figure 4).

On notera que cette forme d'historique contiendra un enregistrement supplémentaire pour chaque entité clôturée par rapport à l'historique des états. En effet, le dernier état d'une telle entité nous informe sur deux événements : la dernière modification d'attributs (début, THEME, BUDGET) et la suppression de l'entité (fin). On notera aussi qu'elle n'utilise pas le concept de *futur infini* (9999).

Les deux formes ayant même contenu informationnel, il est possible de construire chacune d'elles à partir de l'autre. Les requêtes ou les procédures de transformation sont assez aisées à développer.

MODIF_PROJET				
INTITULE	**datem**	**oper**	**THEME**	**BUDGET**
BIOTECH	10	create	Biotechnologie	180.000
BIOTECH	41	update	Génie génétique	160.000
BIOTECH	47	update	Génie génétique	120.000
BIOTECH	84	update	Génie génétique	140.000
BIOTECH	135	update	Biotechnologie	140.000
AGRO-2000	87	create	Amélior. céréales	65.000
AGRO-2000	114	update	Amélior. céréales	75.000
AGRO-2000	125	update	Amélior. céréales	82.000
SURVEYOR	65	create	Surv. par satellite	310.000
SURVEYOR	80	update	Surv. par satellite	375.000
SURVEYOR	101	update	Surv. par satellite	345.000
SURVEYOR	120	delete	Surv. par satellite	345.000

Figure 28 - Historique des événements.

La requête Code 42 calcule les événements à partir du contenu d'une table H_PROJET normalisée. Un événement de type `create` est détecté par la présence d'un état de l'entité qui n'a pas de précédent. Un événement `delete` correspond à un état qui n'est suivi d'aucun autre état et dont fin est fini (fin < 9999). Un événement de type `update` correspond à tout état qui a un précédent.

```
select INTITULE,debut,
       cast("create" as char(6)) as oper,THEME,BUDGET
from   H_PROJET HP
where not exists(select * from  H_PROJET
                 where  INTITULE = HP.INTITULE
```

```
                and    fin = HP.debut)
  union
select INTITULE,fin,
       cast("delete" as char(6)) as oper,THEME,BUDGET
from   H_PROJET HP
where not exists(select * from  H_PROJET
                 where  INTITULE = HP.INTITULE
                 and    debut = HP.fin)
and   fin <> 9999
  union
select INTITULE,debut,
       cast("update" as char(6)) as oper,THEME,BUDGET
from   H_PROJET HP
where  exists(   select * from  H_PROJET
                 where  INTITULE = HP.INTITULE
                 and    fin = HP.debut);
```

Code 42 - Requête de conversion de l'historique des états H_PROJET en historique des événements.

Pour le calcul de H_PROJET à partir de la table des événements MODIF_PROJET, on peut proposer la requête Code 43. Elle considère deux sources d'états : d'abord les couples d'événements consécutifs (dont le premier est forcément un create ou un update) qui forment les états passés, ensuite les événements create ou update qui ne sont suivis d'aucun autre événement, et qui forment les états courants (fin=9999). On remarquera que les événements delete ne sont pas explicitement cités. Ils interviennent en revanche comme second événement de la première source des états passés.

```
select P1.INTITULE,P1.datem as debut,P2.datem as fin,
       P1.THEME,P1.BUDGET
from   MODIF_PROJET P1, MODIF_PROJET P2
where  P1.modif in ('create','update')
and    P1.INTITULE = P2.INTITULE
and    P2.datem = (select min(datem)
                   from   MODIF_PROJET
                   where  INTITULE = P1.INTITULE
                   and    datem > P1.datem)
  union
select P1.INTITULE,P1.datem as debut,
       cast(9999 as integer) as fin,P1.THEME,P1.BUDGET
from   MODIF_PROJET P1
where  P1.modif in ('create','update')
and    not exists (select *
                   from   MODIF_PROJET
                   where  INTITULE = P1.INTITULE
                   and    datem > P1.datem);
```

Code 43 - Requête de conversion de l'historique des événements MODIF_PROJET en historique des états.

Ces requêtes sont aisément transformées en vues. Comme nous l'avons vu pour la projection et la normalisation, nous pourrions développer des procédures effectuant ces conversions de manière certainement plus efficace. Cet exercice est laissé à l'initiative du lecteur.

13. Historique d'associations

Nous n'avons considéré jusqu'ici que l'évolution d'une population d'entités. En ce qui concerne les liens entre les entités, nous n'avons admis que des *types d'associations fonctionnels* (binaires *un-à-plusieurs* ou *un-à-un*), qui se traduisent par des clés étrangères appartenant aux tables d'entités (telles que EMPLOYE.PROJET).

Les **types d'associations non fonctionnels** (N-aires ou binaires *plusieurs-à-plusieurs*, ou encore binaire avec attributs) introduisent des structures nouvelles que nous examinerons brièvement.

Considérons un domaine d'application dans lequel *des employés lisent des journaux* mis à leur disposition par l'entreprise. On dispose d'une table (historique) des clients et d'une table (historique) des journaux. Le fait que *tel employé lit tel journal* est enregistré dans la table LECTURE, elle aussi historique. La base de données pourrait correspondre à la Figure 29.

H_EMPLOYE				
CODE	**debut**	**fin**	**NOM**	**STATUT**
C45	87	114	Carlier	T
C45	114	129	Carlier	P
C45	129	150	Carlier	T
A68	105	132	Albert	P
A68	132	9999	Albert	T

H_JOURNAL				
TITRE	**debut**	**fin**	**PER**	**NBRE**
Le Monde	90	9999	Q	5
Le Soir	95	127	Q	2
Le Soir	127	9999	Q	4
L'Echo	80	110	H	3
L'Echo	110	130	M	3

H_LECTURE			
CODE	**TITRE**	**debut**	**fin**
C45	Le Monde	92	120
C45	L'Echo	87	125
C45	Le Monde	140	150
A68	Le Soir	105	9999
A68	L'Echo	110	120
A68	Le Monde	87	114
A68	L'Echo	125	130

Figure 29 - Historiques d'entités et historique d'associations.

On y observe par exemple que l'employé C45 (Carlier) a demandé à lire Le Monde entre 92 et 120, puis entre 140 et 150. En outre, ce même employé à lu L'Echo entre

87 et 125. Il a donc pris le temps, entre 92 et 120, de lire deux journaux, alors qu'il n'en a lu aucun entre 125 et 140.

On peut mettre en évidence deux caractéristiques importantes de ce type de table :

1. une association peut être *intermittente*; elle peut en effet cesser d'exister pour réapparaître plus tard; on pourra aussi considérer que, dans un tel cas, il s'agit d'associations distinctes;
2. alors qu'une entité n'a qu'un seul état à un instant déterminé, *plusieurs associations* impliquant cette entité peuvent exister à ce même instant.

A titre d'information, on a tiré de cette base de données historique la version habituelle des états courants (Figure 30).

H_EMPLOYE		
CODE	**NOM**	**STATUT**
A68	Albert	T

H_JOURNAL		
TITRE	**PER**	**NBRE**
Le Monde	Q	5
Le Soir	Q	4

LECTURE	
CODE	**TITRE**
A68	Le Soir
A68	Le Monde

Figure 30 - Les états courants de la base de données de la Figure 29.

Il semble évident qu'à un instant déterminé, quel que soit l'employé, il n'existe pas plus d'une association entre celui-ci et un journal quelconque. La situation contraire n'aurait en effet aucun sens, du moins dans une table de base[37]. Les deux lignes (C45, Le Monde, 92, 120) et (C45, Le Monde, 105, 135) ne peuvent donc coexister dans la table H_LECTURE. On exclut aussi la possibilité d'enregistrer deux états jointifs identiques tels que (C45, Le Monde, 92, 105) et (C45, Le Monde, 105, 120), ceux-ci devant être fusionnés avant enregistrement.

On en déduit que deux apparitions successives d'une même association sont non jointives; cette association doit avoir réellement disparu entretemps.

On déclarera **normalisée**, une table historique d'associations (sans attributs) dans laquelle deux états d'une même associations ont des intervalles disjoints et non jointifs. La notion d'historique *corrompu* n'est pas pertinent lorsque le type d'association n'a pas d'attributs.

L'identifiant primaire de la table H_LECTURE est (CODE,TITRE,debut). Une table d'associations se distingue des tables représentant l'historique d'entités selon les aspects suivants.

37. Une table qui dériverait d'une chaîne de requêtes pourrait contenir des associations identiques non disjointes dans le temps.

1. Le concept d'identifiant d'entité est remplacé par celui d'*identifiant d'association* (CODE,TITRE). Cet identifiant n'est pas soumis aux contraintes de stabilité et de non recyclabilité.
2. La gestion des associations ne requiert que deux opérations : insert et delete.
3. On ne peut plus parler de l'historique d'une association sous la forme d'états successifs : une association déterminée (*tel employé lit tel journal*) a une période de validité qui indique simplement quand cette association a été établie (debut) et quand elle s'est terminée (fin). Rien n'empêche qu'une autre association soit établie plus tard entre ce même employé et ce même journal, ce qu'on considérera comme une association distincte de la précédente.
4. Un historique d'associations est normalisé si deux associations identiques (ayant mêmes valeurs de l'identifiant d'association) ont des périodes de validité disjointes et non jointives.

Les opérateurs de normalisation proposés à la Section 11 ne s'appliquent pas tels quels aux historiques d'associations. En particulier, la reconstruction des états manquants n'aurait aucun sens : elle reviendrait à introduire des associations qui n'existent pas ou n'ont jamais existé !

Type d'associations doté d'attributs

Nous n'avons considéré que les types d'associations sans attributs. L'ajout d'attributs induit d'autres propriétés que nous n'examinerons pas dans ce projet. En particulier, une association peut changer d'état suite à la modification de valeur d'un de ses attributs, ce qui nous autorise à parler des *états successifs* d'une association.

14. Suggestions d'extension

Le projet que nous avons développé n'a pas épuisé, loin s'en faut, le domaine des données temporelles. Nous citerons quelques questions et extensions qui mériteraient un traitement particulier. A charge pour le lecteur d'approfondir celles qui éveillent son intérêt.

14.1 Granularité temporelle

Nous avons, à de rares exceptions près, fait l'hypothèse que le temps se mesurait avec une unité suffisamment petite pour que les événements subis par une entité se produisent à des instants différents. Cette condition, qui garantit l'unicité des identifiants, peut ne pas être satisfaite dans certains contextes.

Considérons par exemple un historique à temps physique géré par une machine dont l'horloge livre le temps courant (now) au 1/1000ème de seconde[38]. Une telle

38. On admet que le temps physique est lu une fois pour toutes en début de transaction, et que cette valeur est conservée et utilisée durant toute la transaction.

granularité est insuffisante pour garantir que deux transactions quelconques démarrent à des instants différents. Le risque est donc grand que plusieurs événements affectant la même entité, déclenchés par des transactions ayant démarré presque simultanément (dans le même millième de seconde), produisent des états distincts ayant malheureusement la même valeur de debut.

Le problème se complique encore dans une machine multi-processeurs, dans laquelle plusieurs transactions peuvent démarrer strictement en même temps. Dans un système distribué également, plusieurs sites peuvent démarrer des transactions affectant la même entité au même moment.

Ce phénomène peut aussi se produire en temps logique si on admet d'enregistrer plusieurs événements affectant la même entité et se produisant durant la même unité de temps. Par exemple, si le temps est mesuré en jours, on ne peut enregistrer plus d'un changement d'état par jour pour chaque entité.

La solution réside dans l'utilisation d'un serveur de temps qui garantisse que toutes les valeurs délivrées sont distinctes. Notre générateur de temps (Code 2) satisfait à cette exigence dans un contexte mono-utilisateur. On peut cependant généraliser cette idée de la manière suivante, applicable tant au temps physique qu'au temps logique.

1. Les colonnes temporelles enregistrent des instants selon une ligne du temps abstraite formée des entiers naturels.
2. Le serveur de temps est non partageable, c'est à dire qu'il ne peut délivrer de valeur qu'à un seul client à la fois.
3. Le serveur dispose d'un générateur de nombres entiers distincts, comme celui que nous avons utilisé dans cette étude.
4. Lors de chaque demande d'une valeur de temps par un client, le serveur tire un nombre entier n de son générateur, lui associe le temps réel t (donné par l'horloge ou le client selon la nature du temps), stocke le couple <n,t> dans une table TIME(N,T) et livre au client le nombre n. L'identifiant de la table TIME est N, mais en général T n'en constitue pas un.
5. La table TIME permet d'établir la correspondance nécessaire entre le temps réel propre aux utilisateurs et le temps abstrait enregistré dans les historiques.

14.2 Les colonnes facultatives

Les colonnes des tables considérées dans ce projet sont par défaut obligatoires, ce qui a simplifié certains raisonnements et certaines requêtes. En particulier, toute projection temporelle incluant l'identifiant d'entité produit un historique complet (pas d'états manquants). Si on accepte que des colonnes soient facultatives, que signifie un état dont les colonnes de projection (hors identifiants) sont toutes à null ? On peut admettre qu'un tel état n'existe pas, ce qui conduit à des historiques incomplets. A l'inverse, l'opérateur de jointure temporelle doit aussi être reconsidéré, de manière à conserver les états d'un argument sans correspondance dans l'autre argument, soit une *jointure temporelle externe*.

14.3 Historiques d'entités et d'associations

Nous avons très sommairement abordé la représentation des tables d'associations (Section 13), dont les propriétés différaient de celles des tables d'entités. Il serait utile de développer les règles, les requêtes et les procédures propres aux bases de données historiques comportant des tables des deux types.

14.4 Les colonnes d'attribut sans dimension temporelle

Dans notre projet, l'évolution de chaque colonne d'attribut (hors identifiant d'entité) était soigneusement consigné dans l'historique. Dans les situations réelles, on ne désire pas toujours enregistrer l'évolution de toutes ces colonnes. Il existe donc des colonnes *historisées*, dont on conserve l'historique, et des colonnes *non historisées*, dont on ne conserve que l'état courant.

On suggère au lecteur d'adapter le modèle de données historiques ainsi que les requêtes et procédures de manière à prendre en compte les tables mixtes comportant des colonnes des deux types.

14.5 La prise en compte du futur : les bases de données temporelles

Nous n'avons représenté que les états passés et courants des entités (et des associations). Il serait intéressant d'admettre également l'enregistrement de l'évolution future du domaine d'application. On pourrait ainsi consigner aujourd'hui que le budget de tel projet *passera à 185.000* ***dans deux mois***, ou que tel employé *changera de statut* ***dans 15 jours***. Une telle base de données sera qualifiée de **temporelle**, au sens large du terme. Cette extension pose quelques problèmes intéressants. Par exemple, on observe que l'état courant d'une entité peut changer sans que les données subissent aucune modification : il suffit que l'instant présent (now), qui par nature est en évolution constante, corresponde à la valeur de debut d'un état considéré jusqu'ici comme futur.

On propose d'étendre le modèle, les requêtes et les procédures de manière à prendre en compte les états futurs.

14.6 Répartition des données historiques

Dans cette étude, nous avons adopté un mode de stockage selon lequel tous les états, passés et courants, d'une population d'entités ainsi que toutes les colonnes étaient stockés dans une même table. Bien d'autres modes de répartition sont possibles, offrant chacun des avantages et des inconvénients quant à l'espace occupé et au temps d'exécution des requêtes de consultation et de gestion. La Section 12.1 suggérait un premier écart par rapport à l'approche mono-table. D'une manière générale,

on admettra deux dimensions de répartition, conduisant à une gamme de stratégies distinctes.

1. *Répartition horizontale selon les états*. En ignorant les états futurs, trois configurations sont proposées :
   - tous les états sont stockés dans une même table (formule de cette étude);
   - les états passés et les états courants sont stockés dans deux tables distinctes;
   - tous les états sont stockés dans une même table mais les états courants sont dupliqués dans une seconde table.

   La prise en compte des états futurs réduit les possibilités à la première seulement. En effet, à chaque instant, des états enregistrés comme futurs deviennent courants. Il est irréaliste de migrer ou de dupliquer automatiquement et en temps réel ces états d'une table à l'autre comme proposé dans les deux dernières configurations.
2. *Répartition verticale selon les colonnes*. Les colonnes peuvent être réparties en plusieurs tables. Plusieurs configurations types :
   - toutes les colonnes sont réunies dans une même table (formule de cette étude); cette forme comporte généralement des redondances, puisque les données qui n'évoluent pas peuvent se retrouver dans plusieurs états;
   - les colonnes non historisées sont regoupées dans une seule table (ne contenant que des états courants) tandis que les colonnes historisées sont regroupées dans une table séparée, accompagnées de l'identifiant d'entité et des colonnes temporelles; cette forme comporte moins de redondances que la précédente puisque les données non historisées ne sont pas dupliquées;
   - par rapport à la configuration précédente, on peut stocker les états propres à chaque colonne historisée dans une table autonome, comme proposé dans la Section 12.1; cette forme est toujours normalisée au sens de la théorie relationnelle pour autant que la table des état courant soit au départ normalisée.

14.7 Base de données bitemporelle

Jusqu'ici, nous avons considéré qu'une base de données historiques (ou temporelles) ne possédait qu'une seule dimension temporelle, soit logique, soit physique. Comme nous l'avons évoqué à la Section 3.6, on peut envisager que les données soient dimensionnées selon ces deux types de temps, ce qu'on appelle des **données bitemporelles**. Cette extension est d'autant plus intéressante qu'elle permet de prendre en compte non seulement l'**évolution naturelle** des entités et associations du domaine d'application (ce que nous avons considéré dans ce projet), mais aussi les **corrections** des états passés, présents et futurs (et l'historique de ces corrections), c'est-à-dire la rectification d'erreurs détectées, soit dans les valeurs de colonnes d'attributs,

soit dans les intervalles de temps logique[39] (Section 5.1). On peut ainsi demander l'historique d'une entité *telle qu'on le pensait correct* à une certaine date du passé, lequel historique peut être différent aujourd'hui, suite à la correction d'erreurs.

On propose d'étendre le modèle, les requêtes et les procédures de manière à prendre en compte la double dimension temporelle.

14.8 La question des performances

La question de savoir si les SGBD standards permettent de gérer et d'exploiter des données temporelles de manière efficace reste ouverte. De nombreux auteurs ont proposé durant les années 80 et 90 diverses techniques spécifiques d'organisation et d'indexation des données temporelles répondant à des critères de performances, surtout en consultation. Il n'est pas certain que les SGBD actuels soient à ce point inefficaces, tant en gestion qu'en consultation, lorsqu'il sont utilisés pour stocker des données temporelles. Un choix judicieux d'index, des techniques de clustering et un tampon de taille adéquate peuvent fournir des temps de gestion et d'exécution tout à fait raisonnables, même pour des requêtes réputées de grande complexité (aux sens ordinaire et mathématique du terme).

Examinons brièvement la requête de projection généralisée (Code 23), qui réalise l'un des opérateurs les plus utilisés. Sur base du modèle d'évaluation de la jointure par boucles emboîtées, la complexité de cette requête est $O(N^4)$, où N est le nombre de lignes de la table source. On pourrait donc penser que cette requête est irréaliste, même pour des historiques de taille réduite.

Admettons que l'identifiant primaire fasse l'objet d'un index (trié, du type B-tree), comme il est naturel dans toute base de données :

```
create unique index IDX_HPRO_PK
on H_PROJET(INTITULE,debut);
```

Admettons encore que la table H_PROJET décrive l'évolution de P projets, dotés chacun de E états en moyenne. On a $N = PxE$. Calculons le coût en accès physiques au disque de la projection relative à un projet quelconque.

La jointure principale est une auto-jointure qui ne concerne que les états de ce projet (`E1.INTITULE=E2.INTITULE`). L'accès aux états du premier argument s'effectue via l'index primaire, qui est ordonné selon les valeurs de l'identifiant. La lecture de ces états est donc extrêmement rapide, puisque les entrées des états d'un même projet sont consécutives dans l'index : accès à l'entrée d'index du premier état du projet, lecture séquentielle de l'index pour les autres entrées, et pour chaque entrée, accès direct à l'état dans la table H_PROJET. L'auto-jointure principale demande alors de croiser ces états avec eux-mêmes (et non avec toute la table, grâce à la condition `E1.INTITULE=E2.INTITULE`). Or, compte tenu d'une taille de

39. Notons qu'il n'est pas admis de modifier les valeurs des colonnes temporelles de temps physique.

tampon raisonnable, ces états sont encore dans le tampon, et ne nécessitent pas d'accès au disque. Selon ces hypothèses, la jointure principale demande donc ni + ExC accès disques, où ni est le nombre moyen d'accès physiques nécessaires pour retrouver la première entrée dans l'index (typiquement 1 à 3 pour le premier projet et 1 pour les suivants) et C le nombre d'accès physiques requis pour accéder à une ligne dans la table à partir de l'index (typiquement de 0,1 à 1).[40]

Chacune des deux premières conditions not exists correspond à une jointure supplémentaire à effectuer pour chaque couple retenu dans la première jointure. Celles-ci opérant encore une fois sur les états du projet en cours, qui sont toujours dans le tampon, le coût en accès au disque est nul.

L'évaluation de la troisième condition not exists, bien que plus complexe (*double jointure* à effectuer pour chacun des couples retenus dans la première jointure) trouvera toujours ses arguments, soient les états du projet courant, dans le tampon. Le coût en accès disque est encore nul.

Considérant à présent l'ensemble des projets, le **coût en accès au disque** se réduit à celui de la lecture séquentielle de l'index et à un accès direct à chaque ligne de la table.

De manière plus concrète, le coût en accès au disque de la projection généralisée de H_PROJET sur INTITULE et THEME est, moyennant un tampon de taille raisonnable, celui de la requête

```
select *
from   H_PROJET
order by INTITULE,debut
```

Considérons à titre d'exemple que la table décrive 1.000 projets dotés chacun de 100 états en moyenne (soient 100.000 lignes occupant par exemple 8.000 pages[41]), que l'index occupe 800 pages, que la table soit en désordre selon l'identifiant primaire et que l'accès à une page arbitraire coûte 12 msec en moyenne. Dans ce cas, la projection coûtera au maximum, en accès au disque : 800x0,012 + 100.000x0,012, soit **1.210 sec**. ou ±20 min.

Si au contraire la table est triée selon l'identifiant, ou si le tampon peut conserver toutes les pages de la table, ce coût est de **106 sec**, soit le temps de lecture séquentielle de l'index et de la table[42]. Il est intéressant de constater que, selon les hypothèses retenues, la complexité du coût en accès au disque n'est plus que de O(N).

40. Si la table est en désordre par rapport à l'identifiant le coût est de 1 accès physique par ligne. Il peut être largement inférieur si les lignes sont raisonnablement ordonnées selon cet identifiant (cas des *clustered index* de DB2 d'IBM).
41. On admet que les projets comportent plus d'attributs qu'on en a considérés dans ce chapitre, et donc que les lignes sont plus longues.
42. Le temps d'exécution réel sera sans doute moindre, car les calculs ont été basés sur l'hypothèse d'accès physiques indépendants (aléatoires, soit 12 msec). En pratique, la lecture anticipative (*read ahead*) permet de lire plusieurs pages à la fois (contenu d'une piste par exemple), ce qui peut réduire le temps à 20 ou 10% de la valeur théorique calculée ici. Ce fait ne change rien à nos conclusions.

En revanche, celle du **coût du calcul** est de $O(PxE^3)$. Dans les hypothèses retenues ci-dessus, on peut estimer le nombre d'opérations élémentaires intervenant dans les jointures (extraction d'une ligne et comparaison) à $1.000x100^3 = 10^9$. A raison de 5 µsec par opération élémentaire, on peut estimer le coût de calcul à 5.000 sec, soit 1,4 heures. C'est évidemment ce composant du coût qui devient prioritaire. Malgré la vitesse croissante des processeurs, le coût de calcul deviendra rapidement prohibitif lorsque le nombre de lignes augmente, ce qui justifie l'approche procédurale en régime d'exploitation.

Cette observation étant faite, rien n'interdit de définir d'autres index favorisant les requêtes ou les opérateurs les plus fréquents ou encore de distribuer les données de chaque table historique comme on l'a suggéré dans la Section 14.6.

Le lecteur intéressé pourra encore examiner l'intérêt de maintenir des tables techniques contenant des résultats précalculés, tels que l'intervalle de vie des entités (utile pour vérifier l'intégrité référentielle temporelle), les couples identifiant/clé étrangère (accélération des jointures fréquentes), certaines projections fréquentes, etc.

14.9 Divers

Au cours de l'exposé, nous avons laissé de côté certaines variantes de constructions. Nous en citerons quelques-unes.

a) Procédures de gestion

On dévelopera une batterie de procédures de bases de données en remplacement des *triggers* de gestion. Les programmes effectueront les modifications de données par l'invocation de ces procédures plutôt que par l'exécution des primitives *insert*, *delete* et *update*.

Cette interface pourra être constituée de *Stored procedures* ou d'une bibliothèque de procédures externes, rédigées dans un langage quelconque (C, Pascal ou Java).

b) Gestion et consultation des historiques des attributs et des modifications

Il est possible d'adapter les règles, les requêtes et procédures sur les autres représentations de données historiques décrites à la Section 12, c'est-à-dire l'historique des attributs et l'historique des événements.

c) Triggers sur vues

Les SGBD qui, tels Oracle et InterBase, autorisent les *triggers* associés aux vues permettent de rédiger les mécanismes de gestion d'une manière plus simple et plus naturelle. On reprendra les *triggers* développés dans ce projet pour les associer aux vues des états courants.

Partie 5

Les composants du projet

15. Les composants du projet

Il est temps à présent de faire la synthèse des ressources qui vont nous permettre de définir, gérer et exploiter une petite base de données temporelle. Nous reprendrons successivement les composants relatifs aux structures de données, à la gestion et à la consultation, puis nous les appliquerons à quelques fonctions représentatives. Nous n'avons repris les composants destinés au calcul des agrégations, que le lecteur ajoutera selon ses besoins.

16. Les structures de données

Nous distinguerons les tables de base et les vues des états courants d'une part, et les structures annexes, telles que les tables de résultat, d'autre part. Il conviendra de choisir le type de temps qu'on désire gérer : le temps physique (*transaction time*) ou le temps logique (*valid time*).

16.1 Les structures de base

On dispose des tables suivantes

- H_PROJET : historique des projets; table de base; *Code* 1.
- H_EMPLOYE : historique des employés; table de base; *Code* 1.
- PROJET : états courants des projets; vue; *Code* 1 pour le temps physique et 10 pour le temps logique.
- EMPLOYE : états courants de employés; vue; *Code* 1 pour le temps physique et 10 pour le temps logique.

16.2 Les structures annexes

Nous aurons besoin de diverses vues et/ou tables temporaires destinées à mettre à disposition le résultat de certains opérateurs.

Les projections

Selon qu'on adopte la version prédicative ou la version procédurale des projections, on définira des vues ou des tables correspondant aux résultats.

- H_PRO_THEME, H_PRO_BUDGET : tables ou vues des projections de H_PROJET sur THEME et BUDGET; vues : *Code* 25; tables temporaires : *Code* 22.
- H_EMP_NOM, H_EMP_STATUT, H_EMP_ADRESSE, H_EMP_PROJET : tables des projections de H_EMPLOYE sur NOM, STATUT, ADRESSE et PROJET; vues : *Code* 25; tables temporaires : *Code* 22.

Les jointures

- H_EMPLOYE_PROJET : vue de la jointure de H_EMPLOYE et H_PROJET; *Code* 30.

17. La gestion des données

17.1 Gestion des historiques selon le temps physique (transaction time)

Générateur de temps

- CURRENT_TIME : générateur d'instants selon le temps physique; *Code* 2.

Exceptions

- ERR_ID, ERR_FK, ERR_STAB_ID, ERR_NODIFF : exceptions d'erreurs de données pour le temps physique; *Code* 3.

Triggers de création d'entités

- TRG_INSERT_PRO : trigger de gestion de l'insertion d'une entité projet; *Code* 4.
- TRG_INSERT_EMP : trigger de gestion de l'insertion d'une entité employé; *Code* 5.

Triggers de modification des attributs d'entité

- TRG_B_UPDATE_PRO et TRG_A_UPDATE_PRO : triggers de gestion de la modification d'une entité projet; *Code* 6.
- TRG_B_UPDATE_EMP et TRG_A_UPDATE_EMP : triggers de gestion de la modification d'une entité employé; *Code* 7.

Triggers de suppression d'entité

- TRG_DELETE_PRO : trigger de gestion de la suppression d'une entité projet; *Code* 8.
- TRG_DELETE_EMP : trigger de gestion de la suppression d'une entité employé; *Code* 9.

17.2 Gestion des historiques selon le temps logique (valid time)

Exceptions

- ERR_ID, ERR_FK, ERR_STAB_ID, ERR_NODIFF, ERR_PERIOD, ERR_DEL, ERR_NO_DEL : exceptions d'erreurs de données pour le temps logique; *Code* 3, 11, 16.

Triggers de création d'entités

- TRG_INSERT_PRO : trigger de gestion de l'insertion d'une entité projet; *Code* 12.
- TRG_INSERT_EMP : trigger de gestion de l'insertion d'une entité employé; *Code* 13.

Triggers de modification des attributs d'entité

- TRG_B_UPDATE_PRO et TRG_A_UPDATE_PRO : triggers de gestion de la modification d'une entité projet; *Code* 14.
- TRG_B_UPDATE_EMP et TRG_A_UPDATE_EMP : triggers de gestion de la modification d'une entité employé; *Code* 15.

Triggers de suppression d'entité

- TRG_DELETE_PRO et TRG_DELETE_PRO2 : triggers de gestion de la suppression d'une entité projet; *Code* 17.
- TRG_DELETE_EMP et TRG_DELETE_EMP2 : triggers de gestion de la suppression d'une entité employé; *Code* 18.

18. La consultation des données

En ce qui concerne la **projection**, nous choisirons l'approche prédicative ou l'approche procédurale selon la taille des données et le caractère critique des performances et de leur prédictibilité.

L'approche prédicative se traduit sous la forme des vues mentionnées à la Section 16, qu'on rappelle :

- H_PRO_THEME, H_PRO_BUDGET.
- H_EMP_NOM, H_EMP_STATUT, H_EMP_ADRESSE, H_EMP_PROJET.

Si on choisit l'approche procédurale de la projection, on utilisera les procédures correspondantes :

- HPROJECT_PRO_THEME, HPROJECT_PRO_BUDGET : procédures de projection généralisée de H_PROJET sur THEME et BUDGET; *Code* 26 et 27.
- HPROJECT_EMP_NOM, HPROJECT_PRO_STATUT, HPROJECT_PRO_ADRESSE, HPROJECT_PRO_PROJET ; procédures de projection généralisée de H_EMPLOYE sur THEME, STATUT, ADRESSE et PROJET; *Code* 27.

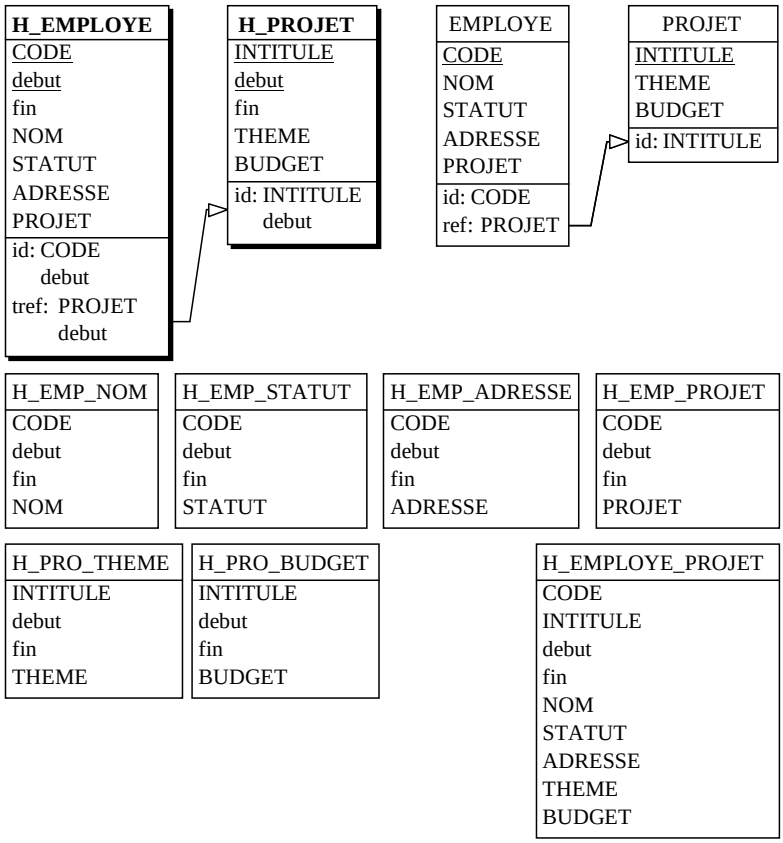


Figure 31 - Tables historiques (H_PROJET, H_EMPLOYE), vues des états courants (PROJET, EMPLOYE), vues (ou tables) des projections (H_PRO_THEME, ..., H_EMP_PROJET) et vue (ou table) de la jointure principale (H_EMPLOYE_PROJET).

Pour la **jointure**, on propose de se limiter au couplage entre H_PROJET et H_EMPLOYE selon la clé étrangère. On peut ainsi utiliser une vue exprimant la jointure.

- H_EMPLOYE_PROJET : vue exprimant la jointure temporelle de H_EMPLOYE et H_PROJET; *Code* 30.

Les composants du projets sont représentés à la Figure 31. pour ce qui concerne les structures de données et aux Figures 32 (temps physique) et 33 (temps logique) pour les composants actifs, *triggers* et exceptions.

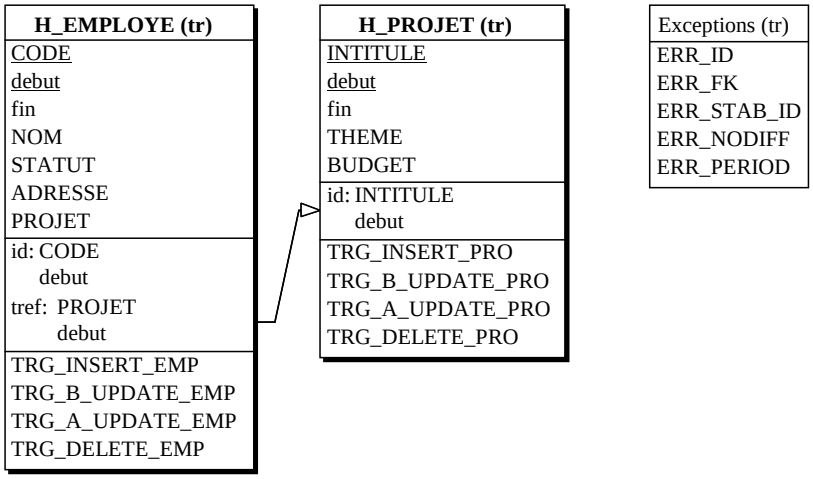


Figure 32 - Tables historiques et composants actifs (triggers et exceptions) pour la gestion du temps physique (*transaction time*).

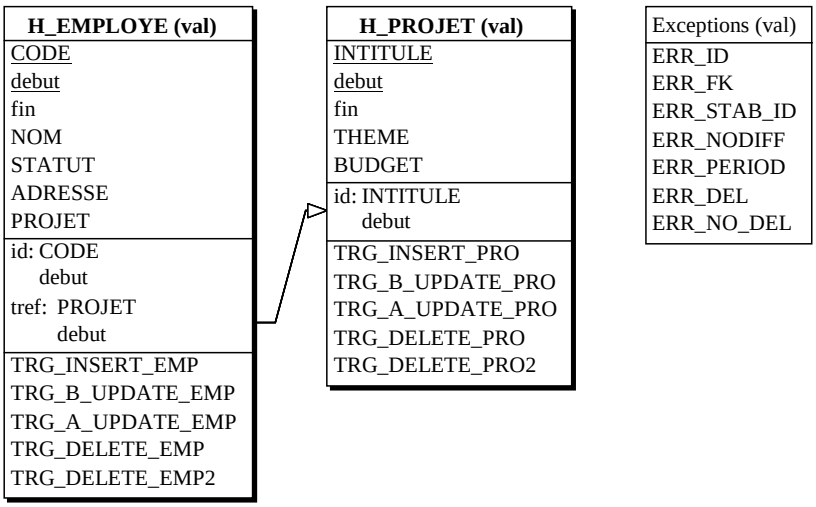


Figure 33 - Tables historiques et composants actifs (triggers et exceptions) pour la gestion du temps logique (*valid time*).

19. Quelques applications représentatives

A titre d'illustration, nous développerons quelques fragments représentatifs d'applications exploitant notre base de données historiques. Ces fragments seront présentés sous la forme de scripts SQL.

Nous construirons un système basé sur le temps logique (*valid time*). Cette décision n'affectera que les scripts d'enregistrement des événements (création, modification et suppression d'entités).

19.1 Création des historiques

a) Création statique

Le script 44 garnit les deux tables H_PROJET et H_EMPLOYE sans faire appel aux mécanismes de gestion automatique. Il permet une expérimentation des requêtes et procédures de manipulation de données. On désactivera (ou on ne créera pas) les *triggers* `TRG_INSERT_PRO` et `TRG_INSERT_EMP` avant d'exécuter ce script.

```
insert into H_PROJET values
       ('BIOTECH',10,41,'Biotechnologie',180000);
insert into H_PROJET values
       ('BIOTECH',41,47,'Génie génétique',160000);
insert into H_PROJET values
       ('BIOTECH',47,84,'Génie génétique',120000);
insert into H_PROJET values
       ('BIOTECH',84,135,'Génie génétique',140000);
insert into H_PROJET values
       ('BIOTECH',135,99999,'Biotechnologie',140000);
insert into H_PROJET values
       ('AGRO-2000',87,114,'Amélior. céréales',65000);
insert into H_PROJET values
       ('AGRO-2000',114,125,'Amélior. céréales',75000);
insert into H_PROJET values
       ('AGRO-2000',125,99999,'Amélior. céréales',82000);
insert into H_PROJET values
       ('SURVEYOR',65,80,'Surv. par satellite',310000);
insert into H_PROJET values
       ('SURVEYOR',80,101,'Surv. par satellite',375000);
insert into H_PROJET values
       ('SURVEYOR',101,120,'Surv. par satellite',345000);

insert into H_EMPLOYE values
       ('C45',52,87,'Carlier','T','Lille','BIOTECH');
insert into H_EMPLOYE values
       ('C45',87,99,'Carlier','T','Lille','AGRO-2000');
insert into H_EMPLOYE values
       ('C45',99,99999,'Carlier','T','Lille','BIOTECH');
insert into H_EMPLOYE values
       ('A68',44,65,'Albert','P','Mons','BIOTECH');
insert into H_EMPLOYE values
       ('A68',65,95,'Albert','P','Mons','SURVEYOR');
insert into H_EMPLOYE values
       ('A68',95,120,'Albert','P','Paris','SURVEYOR');
insert into H_EMPLOYE values
       ('G96',12,38,'Godin','T','Genève','BIOTECH');
insert into H_EMPLOYE values
```

```
       ('G96',38,87,'Godin','P','Genève','BIOTECH');
insert into H_EMPLOYE values
       ('G96',87,99999,'Godin','P','Genève','AGRO-2000');
insert into H_EMPLOYE values
       ('D107',70,76,'Delecourt','T','Grenoble','SURVEYOR');
insert into H_EMPLOYE values
       ('D107',76,97,'Delecourt','T','Genève','SURVEYOR');
insert into H_EMPLOYE values
       ('D107',97,111,'Delecourt','P','Grenoble','SURVEYOR');
insert into H_EMPLOYE values
       ('D107',111,99999,'Delecourt','P','Genève','BIOTECH');
insert into H_EMPLOYE values
       ('D122',47,73,'Declercq','P','Mons','BIOTECH');
insert into H_EMPLOYE values
       ('D122',73,120,'Declercq','P','Mons','SURVEYOR');
insert into H_EMPLOYE values
       ('D122',120,99999,'Declercq','P','Charleroi','AGRO-2000');
insert into H_EMPLOYE values
       ('M158',15,40,'Mercier','T','Paris','BIOTECH');
insert into H_EMPLOYE values
       ('M158',40,65,'Mercier','P','Paris','BIOTECH');
insert into H_EMPLOYE values
       ('M158',65,108,'Mercier','P','Paris','SURVEYOR');
insert into H_EMPLOYE values
       ('M158',108,99999,'Mercier','P','Paris','BIOTECH');
insert into H_EMPLOYE values
       ('A237',93,99999,'Antoine','P','Grenoble','AGRO-2000');
insert into H_EMPLOYE values
       ('N240',82,93,'Nguyen','T','Toulouse','BIOTECH');
insert into H_EMPLOYE values
       ('N240',93,105,'Nguyen','T','Grenoble','SURVEYOR');
insert into H_EMPLOYE values
       ('N240',105,131,'Nguyen','T','Genève','AGRO-2000');
```

Code 44 - Script de création statique des tables historiques en *valid time*. Les triggers *on insert* sont absents ou désactivés.

b) Création dynamique

Les scripts 45 et 46 en revanche *jouent le jeu*, en activant des *triggers* de gestion. On notera que les instructions traduisant les événements doivent être exécutées dans l'ordre des instants de ces événements, qu'ils affectent les projets ou les événements, à défaut de quoi l'intégrité référentielle temporelle risque d'être violée. A titre d'exemple, les *triggers* de gestion ne permettront pas qu'on affecte un employé à un projet qui a été supprimé ou qui n'a pas encore été créé. On donnera les scripts pour le temps physique et pour le temps logique.

Historique à temps physique (*transaction time*)

Etant donné le mécanisme de génération des instants physiques que nous avons adopté, les valeurs des colonnes debut et fin seront différentes de celles des Figures 4 et 5 (multiples de 5) sans que cela nuise aux principes développés dans ce chapitre[43].

```
insert into PROJET values ('BIOTECH','Biotechnologie',180000);
insert into EMPLOYE values ('G96','Godin','T','Genève','BIOTECH');
insert into EMPLOYE values ('M158','Mercier','T','Paris','BIOTECH');
update EMPLOYE set STATUT = 'P' where CODE = 'G96';
```

```
update EMPLOYE set STATUT = 'P' where CODE = 'M158';
update PROJET set THEME = 'Génie génétique',BUDGET = 160000)
       where INTITULE = 'BIOTECH';
insert into EMPLOYE values ('A68','Albert','P','Mons','BIOTECH');
update PROJET set BUDGET = 120000 where INTITULE = 'BIOTECH';
insert into EMPLOYE values ('D122','Declercq','P','Mons','BIOTECH');
insert into EMPLOYE values ('C45','Carlier','T','Lille','BIOTECH');
insert into PROJET values ('SURVEYOR','Surv. par satellite',310000);
update EMPLOYE set PROJET = 'SURVEYOR' where CODE = 'A68';
update EMPLOYE set PROJET = 'SURVEYOR' where CODE = 'M158';
insert into EMPLOYE values
       ('D107','Delecourt','T','Grenoble','SURVEYOR');
update EMPLOYE set PROJET = 'SURVEYOR' where CODE = 'D122';
update EMPLOYE set ADRESSE = 'Genève' where CODE = 'D107';
update PROJET set BUDGET = 375000 where INTITULE = 'SURVEYOR';
insert into EMPLOYE values ('N240','Nguyen','T','Toulouse','BIOTECH');
update PROJET set BUDGET = 140000 where INTITULE = 'BIOTECH';
insert into PROJET values ('AGRO-2000','Amélior. céréales',65000);
update EMPLOYE set PROJET = 'AGRO-2000' where CODE = 'C45';
update EMPLOYE set PROJET = 'AGRO-2000' where CODE = 'G96';
insert into EMPLOYE values
       ('A237','Antoine','P','Grenoble','AGRO-2000');
update EMPLOYE set ADRESSE = 'Grenoble', PROJET = 'SURVEYOR'
       where CODE = 'N240';
update EMPLOYE set ADRESSE = 'Paris' where CODE = 'A68';
update EMPLOYE set STATUT = 'P', ADRESSE = 'Grenoble'
       where CODE = 'D107';
update EMPLOYE set PROJET = 'BIOTECH' where CODE = 'C45';
update PROJET set BUDGET = 345000 where INTITULE = 'SURVEYOR';
update EMPLOYE set ADRESSE = 'Genève', PROJET = 'AGRO-2000'
       where CODE = 'N240';
update EMPLOYE set PROJET = 'BIOTECH' where CODE = 'M158';
update EMPLOYE set ADRESSE = 'Genève', PROJET = 'BIOTECH'
       where CODE = 'D107';
update PROJET set BUDGET = 75000 where INTITULE = 'AGRO-2000';
update EMPLOYE set ADRESSE = 'Charleroi', PROJET = 'AGRO-2000'
       where CODE = 'D122';
delete from EMPLOYE where CODE = 'A68';
delete from PROJET where INTITULE = 'SURVEYOR';
update PROJET set BUDGET = 82000 where INTITULE = 'AGRO-2000';
delete from EMPLOYE where CODE = 'N240';
update PROJET set THEME = 'Biotechnologie' where INTITULE = 'BIOTECH';
```

Code 45 - Script de création dynamique de l'historique H_PROJET en *transaction time*. Les triggers *on insert*, *on update* et *on delete* pour le *transaction time* doivent être actifs.

Historique à temps logique (*valid time*)

```
insert into PROJET(INTITULE,debut,THEME,BUDGET) values
       ('BIOTECH',10,'Biotechnologie',180000);
insert into EMPLOYE(CODE,debut,NOM,STATUT,ADRESSE,PROJET) values
       ('G96',12,'Godin','T','Genève','BIOTECH');
insert into EMPLOYE(CODE,debut,NOM,STATUT,ADRESSE,PROJET) values
```

43. Une autre différence réside dans le fait que le mécanisme qui simule la lecture du temps machine (générateur InterBase), produit des nombres **uniques**. Il apparaît donc que dans les historiques créés sur ce principe, debut est à elle seule une colonne identifiante. Ceci est plausible pour le *transaction time* géré par une machine unique, mais est moins réaliste pour le *valid time*, plusieurs événements pouvant survenir au même instant dans le domaine d'application comme on peut l'observer dans les deux tables historiques (par exemple, trois événements se sont produits en 120). Cette question sera discutée dans la Section 14.1.

```
       ('M158',15,'Mercier','T','Paris','BIOTECH');
update EMPLOYE set debut = 38, STATUT = 'P' where CODE = 'G96';
update EMPLOYE set debut = 40, STATUT = 'P' where CODE = 'M158';
update PROJET
       set debut = 41, THEME = 'Génie génétique',BUDGET = 160000
       where INTITULE = 'BIOTECH';
insert into EMPLOYE(CODE,debut,NOM,STATUT,ADRESSE,PROJET) values
       ('A68',44,'Albert','P','Mons','BIOTECH');
update PROJET set debut = 47, BUDGET = 120000
       where INTITULE = 'BIOTECH';
insert into EMPLOYE(CODE,debut,NOM,STATUT,ADRESSE,PROJET) values
       ('D122',47,'Declercq','P','Mons','BIOTECH');
insert into EMPLOYE(CODE,debut,NOM,STATUT,ADRESSE,PROJET) values
       ('C45',52,'Carlier','T','Lille','BIOTECH');
insert into PROJET(INTITULE,debut,THEME,BUDGET) values
       ('SURVEYOR',65,'Surv. par satellite',310000);
update EMPLOYE set debut = 65, PROJET = 'SURVEYOR'
       where CODE = 'A68';
update EMPLOYE set debut = 65, PROJET = 'SURVEYOR'
       where CODE = 'M158';
insert into EMPLOYE(CODE,debut,NOM,STATUT,ADRESSE,PROJET) values
       ('D107',70,'Delecourt','T','Grenoble','SURVEYOR');
update EMPLOYE set debut = 73, PROJET = 'SURVEYOR'
       where CODE = 'D122';
update EMPLOYE set debut = 76, ADRESSE = 'Genève'
       where CODE = 'D107';
update PROJET set debut = 80, BUDGET = 375000
       where INTITULE = 'SURVEYOR';
insert into EMPLOYE(CODE,debut,NOM,STATUT,ADRESSE,PROJET) values
       ('N240',82,'Nguyen','T','Toulouse','BIOTECH');
update PROJET set debut = 84, BUDGET = 140000
       where INTITULE = 'BIOTECH';
insert into PROJET(INTITULE,debut,THEME,BUDGET) values
       ('AGRO-2000',87,'Amélior. céréales',65000);
update EMPLOYE set debut = 87, PROJET = 'AGRO-2000'
       where CODE = 'C45';
update EMPLOYE set debut = 87, PROJET = 'AGRO-2000'
       where CODE = 'G96';
insert into EMPLOYE(CODE,debut,NOM,STATUT,ADRESSE,PROJET) values
       ('A237',93,'Antoine','P','Grenoble','AGRO-2000');
update EMPLOYE
       set debut = 93, ADRESSE = 'Grenoble', PROJET = 'SURVEYOR'
       where CODE = 'N240';
update EMPLOYE set debut = 95, ADRESSE = 'Paris'
       where CODE = 'A68';
update EMPLOYE
       set debut = 97, STATUT = 'P', ADRESSE = 'Grenoble'
       where CODE = 'D107';
update EMPLOYE set debut = 99, PROJET = 'BIOTECH'
       where CODE = 'C45';
update PROJET set debut = 101, BUDGET = 345000
       where INTITULE = 'SURVEYOR';
update EMPLOYE
       set debut = 105, ADRESSE = 'Genève', PROJET = 'AGRO-2000'
       where CODE = 'N240';
update EMPLOYE set debut = 108, PROJET = 'BIOTECH'
       where CODE = 'M158';
update EMPLOYE
       set debut = 111, ADRESSE = 'Genève', PROJET = 'BIOTECH'
       where CODE = 'D107';
update PROJET set debut = 114, BUDGET = 75000
       where INTITULE = 'AGRO-2000';
update EMPLOYE
       set debut = 120, ADRESSE = 'Charleroi',PROJET = 'AGRO-2000'
       where CODE = 'D122';
update EMPLOYE set fin = 120
       where CODE = 'A68';
```

```
update PROJET set fin = 120
       where INTITULE = 'SURVEYOR';
update PROJET set debut = 125, BUDGET = 82000
       where INTITULE = 'AGRO-2000';
update EMPLOYE set fin = 131
       where CODE = 'N240';
update PROJET set debut = 135, THEME = 'Biotechnologie'
       where INTITULE = 'BIOTECH';
```

Code 46 - Script de création dynamique de l'historique H_PROJET en *valid time*. Les triggers *on insert*, *on update* et *on delete* pour le *valid time* doivent être actifs.

19.2 Consultation des données historiques

1. Etablir l'historique des adresses et des projets des employés.

Le résultat est une table T(CODE,debut,fin,ADRESSE,PROJET) telle que si **<c**,**d**,**f**,**a**,**p>** ∈ T, alors l'employé **c** était domicilié à l'adresse **a** et était affecté au projet **p** durant la période [**d**,**f**).

- *Analyse*

Les données sont extraites de la table H_EMPLOYE par une projection sur <CODE,ADRESSE,PROJET>. Nous disposons déjà des opérateurs élémentaires de projection sur <CODE,ADRESSE> et <CODE,PROJET>. Ou bien nous développons un nouvel opérateur pour la question posée, ou bien nous effectuons la jointure des deux projections élémentaires. C'est cette dernière solution que nous choisissons.

- *Script SQL*

```
/* création de la tables de résultat (T) */
   create table T (CODE,debut,fin,ADRESSE,PROJET)[44];

/* Calcul de la jointure des projections élémentaires   */
   insert into T(CODE,debut,fin,ADRESSE,PROJET)
   select A.CODE,
     case when P.debut > A.debut then P.debut else A.debut end,
     case when P.fin   < A.fin   then P.fin   else A.fin end,
          ADRESSE,PROJET
   from   H_EMP_ADRESSE A, H_EMP_PROJET P
   where  A.CODE = P.CODE
   and    (P.debut < A.fin)  and  (A.debut < P.fin)

/* le résultat se trouve dans T                          */
```

2. Etablir l'historique, pour chaque employé, de ses projets et de ses adresses.

Le résultat est une table T(INTITULE,ADRESSE,debut,fin) telle que si **<i**,**a**,**d**,**f>** ∈ T, alors le projet **i** a eu un ou des employés domiciliés à l'adresse **a** durant la période [**d**,**f**).

44. Domaines de valeurs des colonnes à définir.

• *Analyse*

Les données sont extraites des tables H_PROJET et H_EMPLOYE via une jointure temporelle. Le résultat de cette jointure doit être projeté sur les colonnes INTITULE et ADRESSE, qui ne constituent pas l'identifiant de la jointure. Le résultat de celle-ci n'est donc pas normalisé. On appliquera donc l'opérateur de projection du résultat sur l'ensemble de ses colonnes de manière à produire une table T normalisée (mais éventuellement incomplète).

• *Script SQL*

```
/* création des tables de travail (T0) et de résultat (T) */
   create table T (INTITULE,ADRESSE,debut,fin)44;
   create table T0(INTITULE,ADRESSE,debut,fin)44;

/* création de la procédure de coalescing de T0 dans T    */
/* selon canevas du Code 27 pour INTITULE et ADRESSE   */
   create procedure HPROJECT_INT_ADR
   as etc.;

/* extraction des couples INTITULE,ADRESSE                */
   insert into T0
   select INTITULE,ADRESSE,debut,fin
   from H_EMPLOYE_PROJET;

/* coalescing (projection sur INTITULE,ADRESSE)           */
   execute procedure HPROJECT_INT_ADR;

/* le résultat se trouve dans T                           */
```

3. Etablir l'historique, pour chaque projet, de la charge salariale mensuelle.

Ce problème est semblable à celui de la moyenne des salaires étudié à la Section 10.2. La principale différence est que la statistique est à calculer, non pas pour chaque état stable, par nature de durée variable, mais pour chaque intervalle d'un mois.

Le résultat se trouvera dans la table STATISTIQUE_BRUTE(PROJET, debut,fin,VALEUR) telle que si <**p**,**d**,**f**,**v**> $\in$ STATISTIQUE_BRUTE, alors le projet **p** a dépensé en salaires le montant **v** durant la période mensuelle [**d**,**f**).

• *Analyse*

Par rapport à la méthode proposée à la Section 10.2, la phase 1 est donc à revoir. Chaque état stable a une durée d'un mois. Il est nécessaire de tenir compte de la relation entre le temps réel et le temps abstrait utilisé dans cette étude. Pour simplifier, on fait l'hypothèse que chaque unité correspond à un mois. La procédure CALC_CALENDRIER est chargée d'établir cette version particulière des états stables. L'utilisateur indiquera la période durant laquelle la statistique doit être calculée.

Une nouvelle procédure de calcul de la statistique est définie, effectuant une *somme* cette fois (CALC_SOMME_ETATS_STABLES).

Il est évident qu'il ne faut pas appliquer la phase 4, qui détruirait la périodicité mensuelle.

• *Script SQL*

```
/* création des tables de travail et de résultat     */
create table ETATS_STABLES(AGREGAT,debut,fin);
create table ETATS_STABLES_VALUES(AGREGAT,debut,fin,VALEUR);
create table STATISTIQUE_BRUTE(AGREGAT,debut,fin,VALEUR);

/* Génération des intervalles mensuels de chaque projet */
/*-------------------------------------------------------*/
create procedure CALC_CALENDRIER
                (T1 decimal(5), T2 decimal(5))
as
declare variable PRO char(12);
declare variable mindebut decimal(5);
declare variable maxfin   decimal(5);
declare variable TPS_COUR decimal(5);
declare variable TPS_SUIV decimal(5);
begin
  delete from ETATS_STABLES;
  for select PROJET, min(debut), max(fin)
      from H_EMP_PROJET
      group by PROJET
      order by PROJET into :PRO, :mindebut, :maxfin
  do
  begin
    /* ajustement de l'intervalle du projet courant */
    if (mindebut < T1) then mindebut = T1;
    if (maxfin > T2)   then maxfin   = T2;
    TPS_COUR = mindebut;
    /* génération des périodes mensuelles du projet courant */
    while (TPS_COUR < maxfin) do
    begin
      TPS_SUIV = TPS_COUR + 1;
      insert into ETATS_STABLES
                  values (:TPS_COUR,:TPS_SUIV,:PRO);
      TPS_COUR = TPS_SUIV;
    end
  end
end;

create procedure CALC_SOMME_ETATS_STABLES
as
begin
   insert into STATISTIQUE_BRUTE
   select AGREGAT, debut, fin, sum(VALEUR)
   from ETATS_STABLES_VALUES
   group by AGREGAT,debut,fin;
end;

/*  script de calcul                                    */
execute procedure CALC_CALENDRIER 80, 100;
execute procedure CALC_ETATS_STABLES_EMP_PRO_SAL;
execute procedure CALC_SOMME_ETATS_STABLES;

/* le résultat se trouve dans STATISTIQUE_BRUTE         */
```

- *Extension.*

1. L'unité de temps est inférieure à la période d'agrégation (par exemple le jour par rapport au mois), mais le salaire évolue selon un rythme mensuel. Dans ce cas, une projection temporelle de H_EMPLOYE sur {PROJET,SALAIRE} est

nécessaire pour éliminer les perturbations non mensuelles. La conversion du temps abstrait en mois est alors très simple pour autant que l'unité abstraite soit une division de la période d'agrégation.

2. Toujours dans l'hypothèse d'une unité inférieure à un mois, si le salaire peut changer en cours de la période d'agrégation, il faut convenir d'une formule de calcul du salaire de référence de la période durant laquelle le salaire a changé : minimum, maximum, valeur ancienne, valeur nouvelle, moyenne proportionnelle, etc. La formule se complique si plusieurs modifications peuvent intervenir au cours de la période d'agrégation (cas du salaire annuel par exemple).

20. Bibliographie

Bölhen 196 — Bohlen, M., Snodgrass, R., Soo, M.. Coalescing in temporal databases. In *Proc. of the 22th Int. Conf. on VLDB*, pp. 180-191, 1996.

Date 1999 — Date, C., J., *Introduction to Database Systems* (7th Edition), Longman, 1999

Bettini 2000 — Bettini, C., Jajodia, S., Wang, S., *Time Granularities in Databases, Data Mining, and Temporal Reasoning*, Springer Verlag, 2000

Böhlen 1996 — Böhlen, M., R., Snodgrass, R., T., Soo, M., D., Coalescing in Temporal Databases, in *Proc. VLDB Conf.*, Sept. 1996

Clifford 1997 — Clifford, J., Dyreson, C., E., Isakowitz, T., Jensen C., S., Snodgrass R., T., On the Semantics of "Now" in Databases, *ACM TODS*, 22(2), June 1997, pp. 171-214.

Kline 1995 — Kline, N., Snodgrass, R. Computing temporal aggregates. In *Proc. of the 11th Int. Conf. on Data Engineering*, pp. 222-231, 1995.

Moon 2003 — Moon, B., Vega-Lopez, I., Immanuel, V. Efficient algorithms for large-scale temporal aggregation. *IEEE Trans. on Knowledge and Data Engineering*, 15(3), pp.744-759, 2003.

Ramlot 2000 — Ramlot, O., *Contribution à la mise au point d'un langage d'accès aux bases de données temporelles*, mémoire de fin le maîtrise en informatique, septembre 2000.

Snodgrass 1995 — Snodgrass, R., T. (ed.), et al., *The TSQL2 Temporal Query Language*, Kluwer Academic, 1995

Snodgrass 2000 — Snodgrass, R., T., *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann, 2000

TimeStamp 2000 — Documents du projet TimeStamp, 1997-2000.

Detienne 2001 Detienne, V., Hainaut, J-L., CASE Tool Support for Temporal Database Design, in *Proc. of ER-2001 conference*, LNCS, Springer-Verlag, 2001

Zimànyi 2006 Zimànyi, E., Temporal Aggregates and Temporal Universal Quantification in Standard SQL, in *ACM Sigmod Records*, 35(2), pp. 16-21, June 2006.

Gao 2005 Gao, D., Jensen. C., Snodgrass, R., Soo, M., Join operations in temporal databases, VLDB Journal (2005) 14, pp. 2-29, 2005

TimeStamp

Volume 1 - Introduction aux bases de données temporelles

Exposés

3. **TimeStamp : Aide au développement et à l'exploitation de données à références temporelles (dans le cadre de la réunion d'évaluation DB-Main en 2000)**
4. Gestion des données temporelles (dans le cadre de la journée d'Ingénierie des Applications de Bases de Données)

DB-Main Manual Series

Projet TimeStamp

AIDE AU DÉVELOPPEMENT ET À L'EXPLOITATION DES BASES DE DONNÉES À RÉFÉRENCE TEMPORELLE

VIRGINIE DETIENNE

NOVEMBRE 2000

The University of Namur - LIBD
Institut d'Informatique
Rue Grandgagnage, 21 B-5000 Namur
http://www.info.fundp.ac.be/libd

TimeStamp

Aide au développement et à l'exploitation des bases de données à référence temporelle

Réunion d'évaluation DB-Main

Virginie Detienne
Thomas Lieutenant (1/2)
Didier Roland (1/3)
Denis Zampuniéris (1997-1999)

Namur, le 28 novembre 2000

Contexte (1/2)

Nombreux secteurs industriels et administratifs concernés par la gestion et l'exploitation de *données qui évoluent avec le temps* ou *données à référence temporelle*

- *gestion du personnel, des clients,...*
- *comptabilité d'une entreprise*
- *registre national*
- *dossiers médicaux*
- *cadastre*
- *...*

Contexte (2/2)

- Gestion des données à référence temporelle complexe
  - structuration des données
  - requêtes complexes
  - performance
- Nombreuses recherches mais pas d 'aide significative aux développeurs
  - *pas de méthodologie*
  - *fonctionnalités relatives aux données temporelles dans les SGBD pauvres*
  - *absence de standardisation (définitions,...)*

Acquis

- Programme DB-Main sur l'ingénierie des bases de données (70 PA depuis 1985)
- Outil (méta-)CASE performant et extensible

Objectif du projet

- Traduire les résultats théoriques disponibles en matériaux opérationnels destinés principalement aux développeurs et aux sociétés de services en informatique

Organisation du projet

- Méthodologie de développement de BD temporelles
  - modèle de spécification de BD temporelles
  - méthode de construction de BD temporelles
  - outil Case supportant le modèle et la méthode
- Programmation des applications de BD temporelles
  - opérateurs de manipulation de données destinés aux programmeurs
- Gestion technique des données temporelles
  - techniques physiques d'implémentation de données temporelles qui se basent sur la technologie actuelle (SQL-2)

Complexité du problème (1/10)

- Types de temps
  - Valid Time (période pendant laquelle un fait est effectif dans la réalité)

EMPLOYE/v					
Matricule	**Nom**	**Adresse /v**	**Salaire /v**	**Vstart**	**Vend**
1	DUPONT	Liège	90000	1	5
1	DUPONT	Louvain	90000	5	999

  - Transaction Time (période pendant laquelle une valeur est enregistrée dans la base de données)

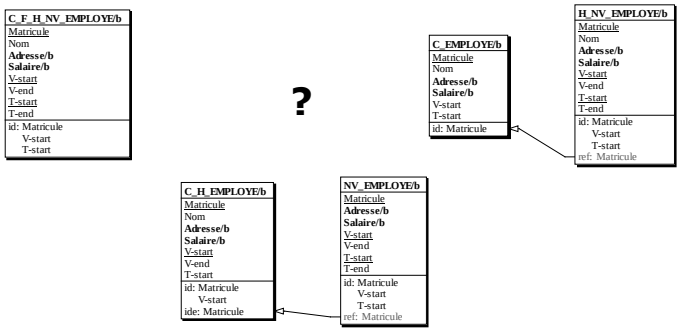
EMPLOYE/t					
Matricule	**Nom**	**Adresse /t**	**Salaire /t**	**Tstart**	**Tend**
1	DUPONT	Liège	90000	3	10
1	DUPONT	Louvain	90000	10	999

  - Bitemporel

EMPLOYE/b							
Matricule	**Nom**	**Adresse /b**	**Salaire /b**	**Vstart**	**Vend**	**Tstart**	**Tend**
1	DUPONT	Liège	90000	1	999	3	10
1	DUPONT	Liège	90000	1	5	10	999
1	DUPONT	Louvain	90000	5	999	10	999

Complexité du problème (2/10)

- Configuration des données
  - Localisation des états courants, futurs, passés et non valides
  - Localisation des attributs temporels



Complexité du problème (3/10)

- Evolution de la base de données

  INSERT, UPDATE et DELETE plus complexes que dans une base de données classique

  - ex : Mise-à-jour dans une table non temporelle

EMPLOYE		
Matricule	**Adresse**	**Projet**
1	Liège	ORGA
2	Bruxelles	CARTO

EMPLOYE		
Matricule	**Adresse**	**Projet**
1	**Namur**	ORGA
2	Bruxelles	CARTO

```
UPDATE EMPLOYE
SET Adresse=Namur
WHERE Matricule=1;
```

Complexité du problème (4/10)

- ex : Mise-à-jour dans une table monotemporelle

H_EMPLOYE				
Matricule	**Adresse /v**	**Projet /v**	**Vstart**	**Vend**
1	Liège	FORET	1	20
1	Liège	ORGA	20	999
2	Bruxelles	CARTO	10	999

H_EMPLOYE				
Matricule	**Adresse /v**	**Projet /v**	**Vstart**	**Vend**
1	Liège	FORET	1	20
1	Liège	ORGA	20	**25**
1	**Namur**	**ORGA**	**25**	**999**
2	Bruxelles	CARTO	10	999

```
UPDATE EMPLOYE
SET Adresse=Namur,
      Vstart=25
WHERE Matricule=1;
```

Complexité du problème (5/10)

Triggers de gestion d'une modification d'un employé (ne gèrent pas les corrections)

```
create trigger TRG_B_UPDATE_EMP for H_EMPLOYE before update
as
declare variable N integer;
begin
  if (new.Vend = 999) then
  begin
    if (new.MATRICULE <> old.MATRICULE ) then
       exception ERR_STAB_ID;
    else
       if (new.ADRESSE = old.ADRESSE
           and new.PROJET = old.PROJET) then
          exception ERR_NODIFF;
       else
          if (new.Vstart <= old.Vstart) then
             exception ERR_PERIOD;
          else
             if (new.PROJET <> old.PROJET) then begin
               select count(*) from H_PROJET
               where INTITULE=new.PROJET and Vend=999 into :N;
               if (N = 0) then
                   exception ERR_FK;
             end
  end
end
```

```
create trigger TRG_A_UPDATE_EMP for H_EMPLOYE after update
as
begin
  if (new.Vend = 999) then
     insert into H_EMPLOYE values(old.MATRICULE,old.Vstart,
       new.Vstart, old.NOM,old.ADRESSE,old.PROJET);
end
```

Complexité du problème (6/10)

- Exploitation de la base de données
  - ex : Projection d'une table non temporelle

Où y a-t-il des employés?

EMPLOYE		
Matricule	**Adresse**	**Projet**
1	Liège	ORGA
2	Bruxelles	CARTO
3	Bruxelles	ORGA
4	Namur	FORET
5	Bruxelles	GEOLO

ADRESSE_EMPLOYE
Adresse
Liège
Bruxelles
Namur

SELECT distinct ADRESSE
FROM H_EMPLOYE;

Complexité du problème (7/10)

- ex : Projection temporelle

Où y a-t-il eu des employés?

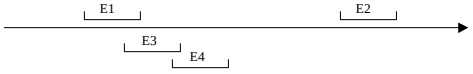
H_EMPLOYE				
Matricule	**Adresse /v**	**Projet /v**	**Vstart**	**Vend**
1	Mons	ORGA	5	10
1	Namur	ORGA	10	18
1	Liège	ORGA	18	999
2	Bruxelles	ORGA	10	14
2	Bruxelles	FORET	14	22
2	Bruxelles	CARTO	22	999
3	Namur	ORGA	5	12
3	Bruxelles	ORGA	12	999
4	Namur	FORET	20	999
5	Bruxelles	GEOLO	2	999

ADRESSE_EMPLOYE		
Adresse /v	**Vstart**	**Vend**
Mons	5	10
Namur	5	18
Namur	20	999
Liège	18	999
Bruxelles	2	999

Complexité du problème (8/10)

Solution prédicative - mauvaises performances

```
select distinct E1.ADRESSE, E1.Vstart, E2.Vend
from   H_EMPLOYE E1, H_EMPLOYE E2
where  E1.ADRESSE=E2.ADRESSE
and    E1.Vstart <= E2.Vstart and E1.Vend <= E2.Vend
and not exists (select *
                from   H_EMPLOYE E0
                where  E0.ADRESSE=E1.ADRESSE
                and    E0.Vstart < E1.Vstart
                and    E0.Vend >= E1.Vstart)
and not exists (select *
                from   H_EMPLOYE E0
                where  E0.ADRESSE=E1.ADRESSE
                and    E0.Vstart <= E2.Vend
                and    E0.Vend > E2.Vend)
and not exists (select *
                from   H_EMPLOYE E3
                where  E3.ADRESSE=E1.ADRESSE
                and    E1.Vend <= E3.Vend
                and    E3.Vend <  E2.Vstart
                and    not exists(select *
                                  from   H_EMPLOYE E4
                                  where  E4.ADRESSE=E1.ADRESSE
                                  and    E3.Vend   <  E4.Vend
                                  and    E4.Vstart <= E3.Vend))
```



Complexité du problème (9/10)

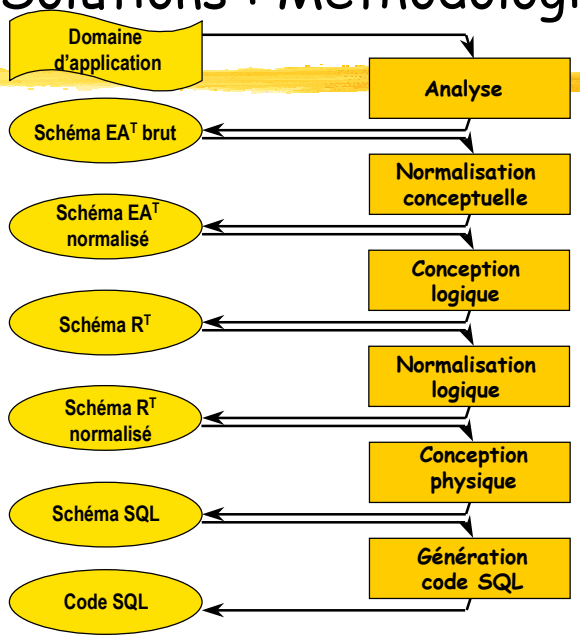
- Solution procédurale - très bonnes performances

```
create procedure HEMPLOYE_ADRESSE
as
declare variable A char(12);      /* dernière ligne extraite */
declare variable d decimal(5);
declare variable f decimal(5);
declare variable curA char(12);   /* état suite courante   */
declare variable curd decimal(5);
declare variable curf decimal(5);
declare variable iter integer;    /* indicateur d'itérations : 0=aucune
                                     1=au moins 1 */
```

Complexité du problème (10/10)

```
begin
  delete from ADRESSE_EMPLOYE;
  iter = 0;
  for select ADRESSE,Vstart,Vend
      from   H_EMPLOYE
      order by ADRESSE,Vstart
      into :A, :d, :f
  do
  begin
    if (iter = 0) then       /* premier passage */
      begin
        curA=A; curd=d; curf=f; /* initial. première suite */
        iter = 1;
      end
    else                     /* passages suivants: une suite est en cours */
      if (curA=A and d<=curf) then  /*  même suite  */
         begin if (f>curf) then curf=f; end
      else
         begin                                  /* rupture de séquence*/
            insert into ADRESSE_EMPLOYE values(:curA,:curd,:curf);
            curA=A; curd=d; curf=f;
         end
  end
  if (iter = 1) then       /* écrire dernière suite */
    insert into ADRESSE_EMPLOYE values(:curA,:curd,:curf);
end
```

Solutions : Méthodologie



Solutions : Outils

- Génération et évolution de la base de données
  - Gestion automatique des différentes étapes
  - Générateur de code SQL avec triggers de gestion de l'aspect temporel
- Exploitation de la base de données
  - Primitives de gestion des données temporelles
  - API du type ODBC
  - Mini-TSQL

Etat du projet

- Méthodologie et gestion automatique de ses différentes étapes
- Générateur de code SQL (Oracle 8)
- Architecture de l'interfaçage
- Mini-TSQL
- Matériaux méthodologiques et pédagogiques

TimeStamp

Volume 1 - Introduction aux bases de données temporelles

Exposés

3. TimeStamp : Aide au développement et à l'exploitation de données à références temporelles (dans le cadre de la réunion d'évaluation DB-Main en 2000)
4. **Gestion des données temporelles (dans le cadre de la journée d'Ingénierie des Applications de Bases de Données)**

DB-Main Manual Series

Projet TimeStamp

GESTION DES DONNÉES TEMPORELLES

VIRGINIE DETIENNE

NOVEMBRE 2000

The University of Namur - LIBD
Institut d'Informatique
Rue Grandgagnage, 21 B-5000 Namur
http://www.info.fundp.ac.be/libd

LIBD

Gestion des données temporelles

Virginie Detienne, Thomas Lieutenant
Projet TimeStamp

Namur, le 2/03/2011

Laboratoire d'Ingénierie des applications de Bases de Données
www.info.fundp.ac.be/libd

LIBD

Contexte

- Nombreux secteurs industriels et administratifs concernés par la gestion des données temporelles
- Technologie la plus utilisée : SQL-92
- SQL-92 : peu de support pour les données temporelles
- Peu de méthodologies et d'outils pour les données temporelles

LIBD

Objectifs

- Méthodologie de conception de bases de données temporelles basée sur SQL-92
- Outil CASE pour la création et la gestion des bases de données temporelles
- API du type ODBC pour la consultation des bases de données temporelles

3

LIBD

Plan

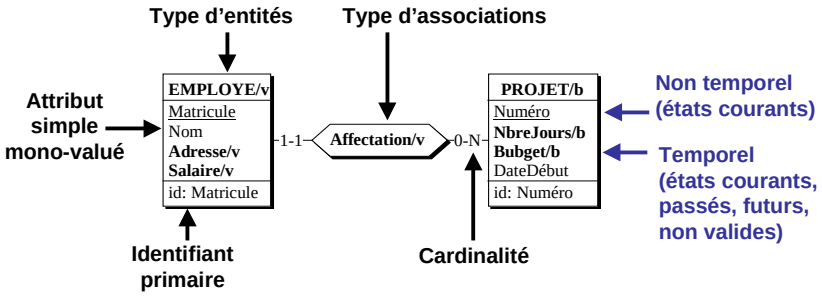
- Définition des modèles
  - Modèle conceptuel temporel
  - Modèle relationnel logique temporel
  - Modèle relationnel physique temporel
- Méthodologie
- Création : Outil CASE
- Gestion : Outil CASE
- Consultation : API

4

LIBD

Modèle conceptuel temporel

- Structures



Valid Time : /v
Transaction Time : /t
Bitemporel : /b
Monotone : /m

5

LIBD

Modèle conceptuel temporel

- Objectifs
  - Assurer la cohérence de la future base de données
  - Limiter sa complexité
  - Rendre son implémentation physique plus facile et plus efficace
  - Exemples :
    - Les attributs temporels d'un type d'entités ont la même dimension temporelle (des attributs temporels et non temporels peuvent coexister);
    - L'identifiant primaire de chaque type d'entités est stable et non recyclable;

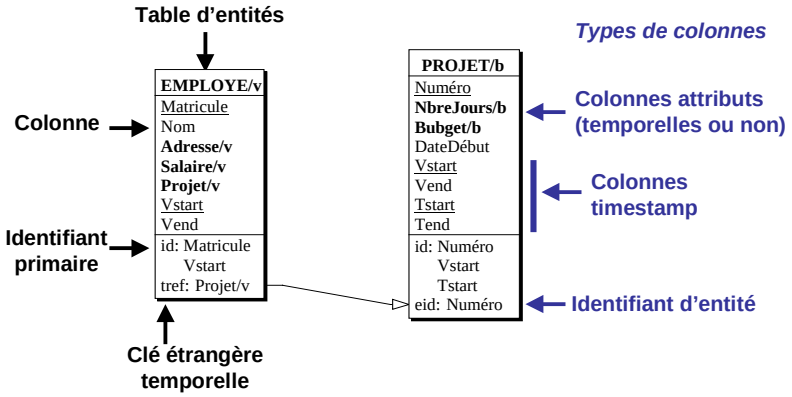
⇒ nécessité d'un outil pour vérifier contraintes

6

LIBD

Modèle relationnel logique temporel

- Structures



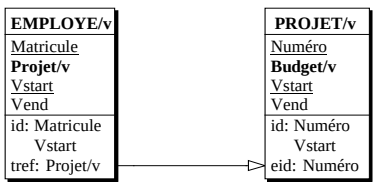
⇒ nécessité d'un outil de conception logique

LIBD

Modèle relationnel logique temporel

- Clés étrangères temporelles
  - Monotemporelle

- P1.Numéro = P2.Numéro = E.Projet
- P1.Vstart ≤ E.Vstart et E.Vend ≤ P2.Vend



LIBD

Modèle relationnel logique temporel

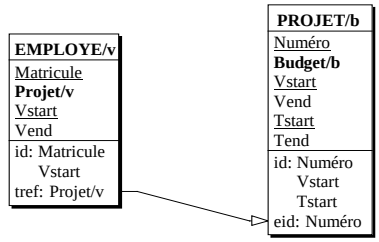
- Bitemporelle

PROJET/b					
Numéro	Budget /b	Vstart	Vend	Tstart	Tend
1	100000	5	999	10	20
1	100000	5	12	20	999
1	150000	12	999	20	30
1	150000	12	20	30	999
1	130000	20	999	30	999
2	160000	8	999	14	999

Etat valide

Etat non valide

EMPLOYE/v			
Matricule	Projet /v	Vstart	Vend
E1	1	8	15
E1	2	15	999
E2	1	10	24



Etats valides

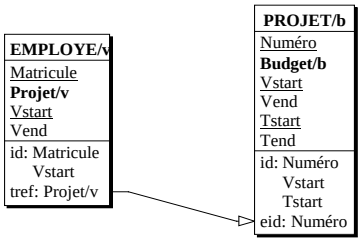
Etats valides +
Etats non valides

LIBD

Modèle relationnel logique temporel

La référence doit se faire entre les états valides

- P1.Numéro = P2.Numéro = E.Projet
- P1.Vstart ≤ E.Vstart et E.Vend ≤ P2.Vend
  et P1.Tend=999 et P2.Tend=999



Etats valides

Etats valides
UNIQUEMENT

LIBD

Modèle relationnel logique temporel

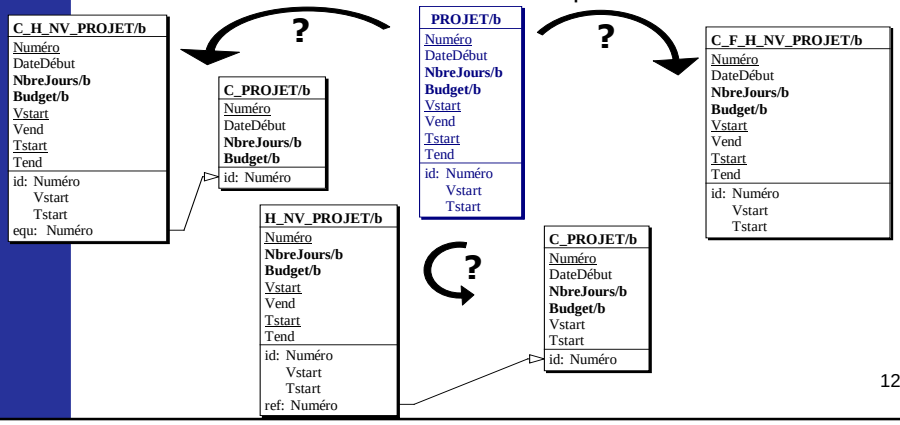
- 25 définitions de contraintes référentielles temporelles

⇒ nécessité d'un outil pour gérer les contraintes référentielles temporelles

LIBD

Modèle relationnel physique temporel

- Distribution de données
  - Localisation des états courants, passés, futurs et invalides
  - Localisation des colonnes temporelles



LIBD

Modèle relationnel physique temporel

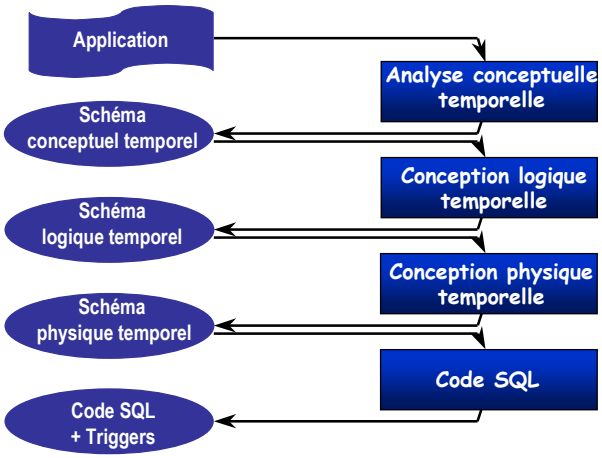
- Gestion automatique des données
  - Base de données logique implémentée comme une base de données active qui gère la cohérence des données et contrôle le mapping logique/physique
  - Pour une base de données contenant 4 tables d'entités, 2 tables d 'associations et 6 clés étrangères monotemporelles et bitemporelles :
    3000 lignes de triggers pour Oracle8

⇒ nécessité d'un outil pour la gestion automatique des données

13

LIBD

Méthodologie



14

LIBD

Gestion : OUTIL Case

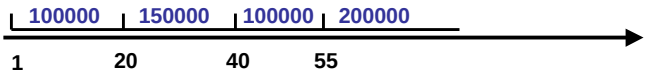
- Triggers
  - INSERT
  - UPDATE
  - DELETE
  - Vérification des contraintes (intégrité référentielle, attributs stables,...)

LIBD

Gestion : OUTIL Case

- Exemple (sans trigger)

  Le budget n 'est pas passé à 100000 à la date 40 mais à la date 15



PROJET/v						
Numéro	NbreJours /v	Budget /v	DateDébut	Client	Vstart	Vend
1	200	100000	25	C1	1	20
1	200	150000	25	C1	20	40
1	200	100000	25	C1	40	55
1	200	200000	25	C1	55	999

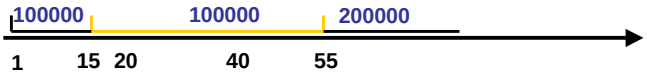
LIBD

Gestion : OUTIL Case

- Exemple (sans trigger)

  Le budget n 'est pas passé à 100000 à la date 40 mais à la date 15

  Première étape



PROJET/v						
Numéro	NbreJours /v	Budget /v	DateDébut	Client	Vstart	Vend
1	200	100000	25	C1	1	15
1	200	150000	25	C1	20	40
1	200	100000	25	C1	15	55
1	200	200000	25	C1	55	999

  - Modifier le Vstart de l'état corrigé (Vstart=15)
  - Modifier le Vend de l'état partiellement recouvert (Vend=15)
  - Supprimer les états entièrement recouverts

17

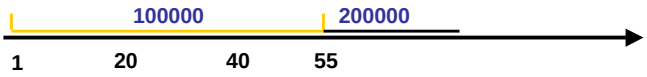
LIBD

Gestion : OUTIL Case

- Exemple (sans trigger)

  Le budget n 'est pas passé à 100000 à la date 40 mais à la date 15

  Seconde étape



PROJET/v						
Numéro	NbreJours /v	Budget /v	DateDébut	Client	Vstart	Vend
1	200	100000	25	C1	1	55
1	200	100000	25	C1	15	55
1	200	200000	25	C1	55	999

  - Réaliser le coalescing

18

LIBD

Gestion : OUTIL Case

- Exemple (avec trigger)

```
UPDATE PROJET
SET Vstart=15
WHERE Numéro=1
AND Vstart=40;
```

PROJET/v

Numéro	NbreJours /v	Budget /v	DateDébut	Client	Vstart	Vend
1	200	100000	25	C1	1	20
1	200	150000	25	C1	20	40
1	200	100000	25	C1	40	55
1	200	200000	25	C1	55	999
2	150	160000	10	C1	5	999

↓ **Géré par un trigger**

PROJET/v

Numéro	NbreJours /v	Budget /v	DateDébut	Client	Vstart	Vend
1	200	100000	25	C1	1	55
1	200	200000	25	C1	55	999
2	150	160000	10	C1	5	999

19

LIBD

Consultation : API

- Problème
  - Opérateurs temporels exprimés en SQL-92 pur complexes et consommateurs de ressources (coalescing en $O(N^4)$)
- Solution
  - API (de type ODBC) pour manipuler les données temporelles
  - Sous-ensemble de TSQL2 = mini-TSQL2
  - Interprétation procédurale des opérateurs temporels

20

LIBD

Consultation API

- Exemple
  - Quels étaient, à la date 25, les salaires des différents employés?
    - Projection
    - Coalescing

EMPLOYE/b
Matricule
Nom
Adresse/b
Salaire/b
Vstart
Vend
Tstart
Tend
id: Matricule Vstart Tstart

  - Sans TSQL
    - +/- 200 lignes de code C/OBDC (quelle fiabilité?)

21

LIBD

Consultation API

- Avec TSQL

```
char matricule[50], salaire [20], output[100];
 sdword *cbmatricule, *cbsalaire ;
 . . .; rc=SQLConnect(hdbc,...); ...
 rc=TSQLExecDirect(hdbc,hstmt,"select Matricule, Salaire
                    from  EMPLOYE
                    where valid(EMPLOYE) contains
                        timepoint '25'",type);
 rc=SQLBindCol(hstmt,1,SQL_C_CHAR, matricule,50,cbname);
 rc=SQLBindCol(hstmt,2,SQL_C_CHAR, salaire,20,cbsalaire );
 do { rc = SQLFetch(hstmt);
     if(rc == SQL_NO_DATA)  break;
     strcpy(output," Matricule : ") ; strcat(output, matricule);
     strcat(output," Salaire : "); strcat(output, salaire );
     MessageBox(output,"TUPLE",MB_OK);
   }while(rc!= SQL_NO_DATA);
 ...; rc=SQLDisconnect(hdbc); ...
```

22