

## Case study 10

---

# Temporal databases - Part 2

**Objective.** In this second part of the study of temporal data, we examine the various ways to query and transform them. We first examine simple temporal and non temporal queries, providing themselves temporal and non temporal results. Then, we extend to the temporal dimension the main families of queries of standard, non temporal, SQL: projection (entity-based and generalized), inner join and outer join, aggregation (count, max, min, average, sum). We also describe the SQLfast temporal library LTemp that offers a series of operators intended to write concise and efficient temporal scripts. The various temporal data models described in part 1 are revisited in order to develop conversion algorithms from one model to each of the other ones. Finally, we address the problem of performance by comparing the various algorithms of the temporal operators. To get realistic execution time measures we apply these algorithms to larger temporal databases. The last section is devoted to a short description of the SQL:2011 standard, that introduces some (but not all) concepts of temporal databases.

**Keywords.** temporal relations, temporal query, temporal projection, coalescing, temporal inner join, temporal outer join, temporal aggregation, stable interval, temporal data model conversion, temporal operator performance, SQL:2011, LTemp library.

## 10.1 Introduction

A temporal database can provide a response to a wide variety of time-related queries. It is easy to understand that extracting, or consulting, data from a set of historical tables is more complex than similar operations on traditional databases, limited to current states of the application domain. This feeling is both true and false. Some temporal queries will have a simple and intuitive formulation in SQL, while others will require far more complex SQL translation than their expression in plain English.

## 10.2 Temporal relations

Temporal queries will often be based on definite relationships between two instants, between an instant and an interval or between two intervals. We will briefly describe these relationships, which specify the relative positions of their arguments. They all derive from Allen's interval algebra which comprises 13 base relations<sup>1</sup>. The three tables below classify the temporal relations and assign them their SQL interpretation. They postulate that instants and intervals share the same time granularity.

Relations between instants <b>t1</b> and <b>t2</b>		
	relation	SQL expression
<i>r1</i>	t1 and t2 simultaneous	$t1 = t2$
<i>r2</i>	t1 and t2 distinct	$t1 \neq t2$
<i>r3</i>	t1 before t2	$t1 < t2$
<i>r3</i>	t1 after t2	$t1 > t2$
<i>r5</i>	t1 not before t2	$t1 \geq t2$
<i>r6</i>	t1 not after t2	$t1 \leq t2$

Relations between instant <b>t</b> and temporal interval <b>I</b> $\equiv [s,e]$		
	relation	SQL expression
<i>r7</i>	t before I	$t < s$
<i>r8</i>	t after I	$t \geq e$
<i>r9</i>	I starts with t	$t = s$
<i>r10</i>	I ends with t	$t = e$
<i>r11</i>	I includes t	$s \leq t \text{ and } t \leq e$
<i>r12</i>	I starts before t	$s < t$
<i>r13</i>	I ends after t	$e > t$

1. [https://en.wikipedia.org/wiki/Allen's\\_interval\\_algebra](https://en.wikipedia.org/wiki/Allen's_interval_algebra)

Relations between intervals $I1 \equiv [s1, e1]$ and $I2 \equiv [s2, e2]$		
	relation	SQL expression
<i>r14</i>	$I1$ before $I2$	$e1 \leq s2$
<i>r15</i>	$I2$ follows $I1$ (aka <i><math>I1</math> meets <math>I2</math></i> )	$s1 = e2$
<i>r16</i>	$I1$ starts $I2$	$s1 = s2$
<i>r17</i>	$I1$ ends $I2$	$e1 = e2$
<i>r18</i>	$I1$ during $I2$	$(s1 \geq s2) \text{ and } (e1 \leq e2)$
<i>r19</i>	$I1$ equals to $I2$	$(s1 = s2) \text{ and } (e1 = e2)$
<i>r20</i>	$I1$ and $I2$ disjoint ( <i>implied by <math>r14</math> and <math>r15</math></i> )	$(e1 \leq s2) \text{ or } (s1 \geq e2)$
<i>r21</i>	$I1$ and $I2$ overlap ( <i>implied by <math>r16</math> to <math>r19</math></i> )	$(e1 > s2) \text{ and } (s1 < e2)$ $\max(s1, s2) < \min(e1, e2)$

### Intersection of intervals

Let us consider intervals  $I1 \equiv [s1, e1]$  and  $I2 \equiv [s2, e2]$ . If they overlap, that is, if  $e1 > s2$  and  $s1 < e2$  (or if  $\max(s1, s2) < \min(e1, e2)$ ), their intersection can be computed as:

$$\text{intersect}(I1, I2) \equiv [\max(s1, s2), \min(e1, e2)]$$

This expression can be generalized to more than two intervals:

$$\text{intersect}(I1, \dots, In) \equiv [\max(s1, \dots, sn), \min(e1, \dots, en)]$$

### Note

It may come as a surprise that we do not suggest in this study to develop a user defined temporal function that checks whether two intervals overlap, something like

$$\text{overlap}(s1, e1, s2, e2)$$

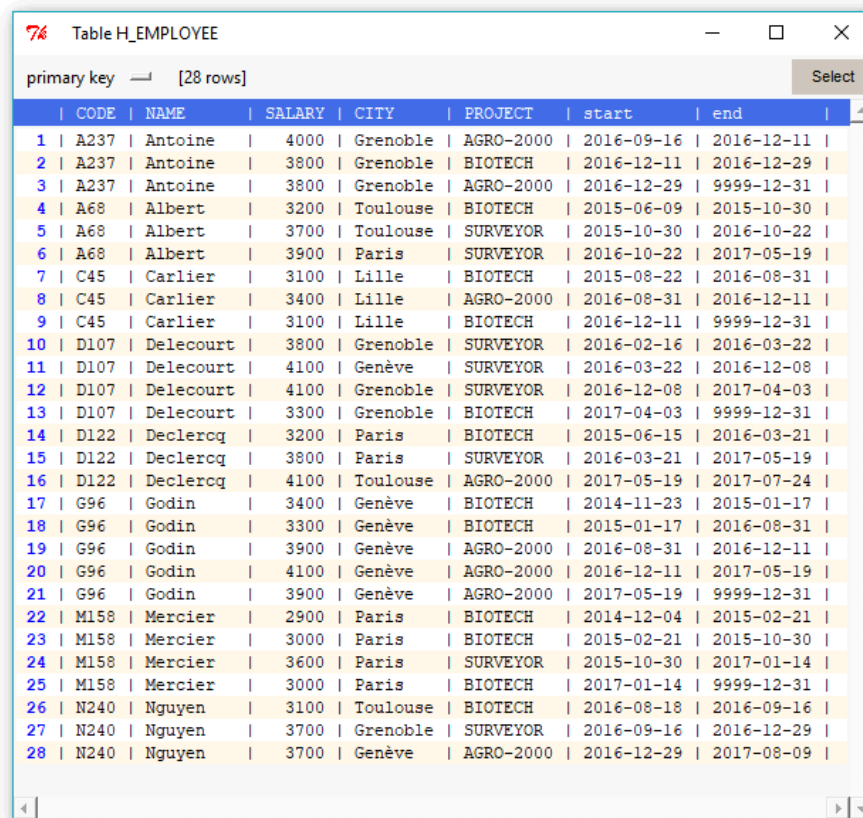
Such a function should simplify many queries and make them easier to write and to understand, and this, especially since several DBMS (SQLite excluded, unfortunately) already include such a function in their core library (the `overlaps()` function of SQL:2011 for example). The answer is related to execution performance of queries. Most of them can be considerably accelerated thanks to indexes defined on temporal columns start and end. If these columns appear explicitly in SQL conditions, the query optimizer is able to take these indexes into account when computing the best execution plan.

So, in the limited context of this case study, the answer is twofold: nice idea for small temporal databases but it will prevent the DBMS to optimize many queries operating on large, complex databases.

Anyway, function `tooverlap`<sup>2</sup> is available in the SQLfast library, but this does not invalidate the above reasoning. Therefore, to use with caution!

### 10.3 The example temporal databases

We will develop and discuss the different classes of queries on the basic temporal model according to which all the states, past and current, are stored in the same tables (*entity-based*, or more generally, *tuple-based* model). Here below, we recall the structure and contents of historical tables `H_EMPLOYEE`, `H_PROJECT` and the views showing the current states of employees and projects.



	CODE	NAME	SALARY	CITY	PROJECT	start	end
1	A237	Antoine	4000	Grenoble	AGRO-2000	2016-09-16	2016-12-11
2	A237	Antoine	3800	Grenoble	BIOTECH	2016-12-11	2016-12-29
3	A237	Antoine	3800	Grenoble	AGRO-2000	2016-12-29	9999-12-31
4	A68	Albert	3200	Toulouse	BIOTECH	2015-06-09	2015-10-30
5	A68	Albert	3700	Toulouse	SURVEYOR	2015-10-30	2016-10-22
6	A68	Albert	3900	Paris	SURVEYOR	2016-10-22	2017-05-19
7	C45	Carlier	3100	Lille	BIOTECH	2015-08-22	2016-08-31
8	C45	Carlier	3400	Lille	AGRO-2000	2016-08-31	2016-12-11
9	C45	Carlier	3100	Lille	BIOTECH	2016-12-11	9999-12-31
10	D107	Delecourt	3800	Grenoble	SURVEYOR	2016-02-16	2016-03-22
11	D107	Delecourt	4100	Genève	SURVEYOR	2016-03-22	2016-12-08
12	D107	Delecourt	4100	Grenoble	SURVEYOR	2016-12-08	2017-04-03
13	D107	Delecourt	3300	Grenoble	BIOTECH	2017-04-03	9999-12-31
14	D122	Declercq	3200	Paris	BIOTECH	2015-06-15	2016-03-21
15	D122	Declercq	3800	Paris	SURVEYOR	2016-03-21	2017-05-19
16	D122	Declercq	4100	Toulouse	AGRO-2000	2017-05-19	2017-07-24
17	G96	Godin	3400	Genève	BIOTECH	2014-11-23	2015-01-17
18	G96	Godin	3300	Genève	BIOTECH	2015-01-17	2016-08-31
19	G96	Godin	3900	Genève	AGRO-2000	2016-08-31	2016-12-11
20	G96	Godin	4100	Genève	AGRO-2000	2016-12-11	2017-05-19
21	G96	Godin	3900	Genève	AGRO-2000	2017-05-19	9999-12-31
22	M158	Mercier	2900	Paris	BIOTECH	2014-12-04	2015-02-21
23	M158	Mercier	3000	Paris	BIOTECH	2015-02-21	2015-10-30
24	M158	Mercier	3600	Paris	SURVEYOR	2015-10-30	2017-01-14
25	M158	Mercier	3000	Paris	BIOTECH	2017-01-14	9999-12-31
26	N240	Nguyen	3100	Toulouse	BIOTECH	2016-08-18	2016-09-16
27	N240	Nguyen	3700	Grenoble	SURVEYOR	2016-09-16	2016-12-29
28	N240	Nguyen	3700	Genève	AGRO-2000	2016-12-29	2017-08-09

2. `tooverlap`, for *temporal overlap*, applicable to closed-open intervals.

7% Table H_PROJECT							-	□	✕
primary key [11 rows]							Select		
	ID	TITLE	THEME	BUDGET	start	end			
1	AGRO-2000	Crop improvement	65000	2016-08-31	2017-04-14				
2	AGRO-2000	Crop improvement	75000	2017-04-14	2017-07-05				
3	AGRO-2000	Crop improvement	82000	2017-07-05	9999-12-31				
4	BIOTECH	Biotechnology	180000	2014-11-18	2015-02-27				
5	BIOTECH	Genetic engineering	160000	2015-02-27	2015-06-15				
6	BIOTECH	Genetic engineering	120000	2015-06-15	2016-08-23				
7	BIOTECH	Genetic engineering	140000	2016-08-23	2017-09-17				
8	BIOTECH	Biotechnology	140000	2017-09-17	9999-12-31				
9	SURVEYOR	Satellite monitoring	310000	2015-10-30	2016-06-25				
10	SURVEYOR	Satellite monitoring	375000	2016-06-25	2016-12-20				
11	SURVEYOR	Satellite monitoring	345000	2016-12-20	2017-05-19				

7% View EMPLOYEE							-	□	✕
CODE [5 rows]							Select		
	CODE	NAME	SALARY	CITY	PROJECT	start	end		
1	A237	Antoine	3800	Grenoble	AGRO-2000	2016-12-29	9999-12-31		
2	C45	Carlier	3100	Lille	BIOTECH	2016-12-11	9999-12-31		
3	D107	Delecourt	3300	Grenoble	BIOTECH	2017-04-03	9999-12-31		
4	G96	Godin	3900	Genève	AGRO-2000	2017-05-19	9999-12-31		
5	M158	Mercier	3000	Paris	BIOTECH	2017-01-14	9999-12-31		

7% View PROJECT							-	□	✕
TITLE [2 rows]							Select		
	ID	TITLE	THEME	BUDGET	start	end			
1	AGRO-2000	Crop improvement	82000	2017-07-05	9999-12-31				
2	BIOTECH	Biotechnology	140000	2017-09-17	9999-12-31				

This database is fine to develop temporal queries and scripts, to analyze their behavior and to observe their effect, but it is far too small to be useful when we evaluate and compare their execution performance. To this goal, we will use larger experimental data sets comprising the history of 100 to 200,000 employees.

## 10.4 Non temporal queries

A history table also shows the current states of the entities and can therefore answer the classic queries specific to non-temporal databases. Expressed on the views of the current states, these queries are identical to those of the equivalent non-temporal databases.

Printed 29/8/19

- What is the current state of employee M158?

```
select CODE, NAME, SALARY, CITY, PROJECT
from   EMPLOYEE
where  CODE = 'M158';
```

**Script 10.1** - Current state of employee M158

CODE	NAME	SALARY	CITY	PROJECT
M158	Mercier	3000	Paris	BIOTECH

- Same query against the historical table.

```
select CODE, NAME, SALARY, CITY, PROJECT
from   H_EMPLOYEE
where  CODE = 'M158' and end = '9999-12-31';
```

**Script 10.2** - Current state of employee M158 (from the historical table).

- Show the project(s) with the highest budget.

```
select distinct TITLE, BUDGET
from   PROJECT
where  BUDGET = (select max(BUDGET) from PROJECT);
```

**Script 10.3** - Project(s) with the highest budget.

TITLE	BUDGET
BIOTECH	140000

## 10.5 Temporal queries with temporal result

A temporal query consults past and current entity states. Its result may or may not be temporal. The state of an entity, or of a set of entities, at a definite time point generally is called a **snapshot**.

We will first consider simple queries that produce temporal data extracted from a single table. In these queries, variable *Tfuture* denotes infinite future, typically 9999-12-31.

- Show the history of employees between 2016-01-01 and 2016-12-31.

The interesting aspect of this query is the *clipping* of the intervals of the first and last state of each employee to adjust them to the interval of the query.

```
select CODE, NAME, SALARY, CITY, PROJECT
       max(start, '2016-01-01') as "Start",
       min(end, '2016-12-31') as "End",
from   H_EMPLOYEE
where  start < '2016-12-31' and '2016-01-01' < end
order by NAME, "Start", "End";
```

**Script 10.4** - History of employees between 2016-01-01 and 2016-12-31

CODE	NAME	SALARY	CITY	PROJECT	Start	End
A68	Albert	3700	Toulouse	SURVEYOR	2016-01-01	2016-10-22
A68	Albert	3900	Paris	SURVEYOR	2016-10-22	2016-12-31
...	...	...	...	...	...	...
N240	Nguyen	3700	Grenoble	SURVEYOR	2016-09-16	2016-12-29
N240	Nguyen	3700	Genève	AGRO-2000	2016-12-29	2016-12-31

- Show the life of the projects

To make data more readable, infinite future is displayed as *null* value.

```
select TITLE, min(start) as Started,
       case when max(end) = '$Tfuture$'
            then null
            else max(end) end as Closed
from   H_PROJECT
group by TITLE
```

**Script 10.5** - Life of the projects

TITLE	Started	Closed
AGRO-2000	2016-08-31	--
BIOTECH	2014-11-18	--
SURVEYOR	2015-10-30	2017-05-19

- Show the start date, end date and life span (duration or age) of each closed project  
This query makes use of temporal function `durationDays(d1,d2)`, which returns the number of days between time points `d1` and `d2` (typically dates). If the arguments denote time or datetime values, the result is rounded to the closest integer.

```
select TITLE, min(start) as "Start", max(end) as "End",
       durationDays(min(start),max(end)) as Age
from   H_PROJECT
group by TITLE
having max(end) < '$Tfuture$';
```

**Script 10.6** - Start date, end date and duration of each closed project

TITLE	Start	End	Age
SURVEYOR	2015-10-30	2017-05-19	567

- We observe that several expressions appear more than once, which may make the query fairly obscure and its execution potentially inefficient (depending on the optimizer). We suggest to rewrite the query with a CTE.

```
with MinMax(TITLE,MinStart,MaxEnd)
as (select TITLE, min(start) as MinStart, max(end) as MaxEnd
    from   H_PROJECT
    group by TITLE
    having MaxEnd < '$Tfuture$')
select TITLE,MinStart as "Start",MaxEnd as "End",
       durationDays(MinStart,MaxEnd) as Duration
from   MinMax;
```

**Script 10.7** - Same with a CTE

- Display the start date and current age of each active project  
The *age* of an active project is computed from its start date to the current date (given by system variable `date`). This is a typical example of non deterministic query: run it tomorrow and it will give another result, though the database has not been updated!



```

with StartAge(TITLE,MinStart)
as (select TITLE, min(start) as MinStart
    from   H_PROJECT
    group by TITLE
    having max(end) = '$Tfuture$')
select TITLE,MinStart as "Start",
       durationDays(MinStart,'$date$') as Age
from   StartAge;

```

**Script 10.8** - Start date and current age of each active project (through a CTE)

TITLE	Start	Age
AGRO-2000	2016-08-31	917
BIOTECH	2014-11-18	1569

- When did the budget of each project exceed 150,000?

```

select TITLE,BUDGET,start,end
from   H_PROJECT
where  BUDGET > 150000;

```

**Script 10.9** - When did the budget of each project exceed 150,000?

*Warning.* Though the result shown below looks good, this query can produce a non-standardized history, as states of some projects may not be consecutive and some consecutive states may be identical (the exact term is *value-equivalent*). Actually, the operation requested is a *projection*. More on this later on. Same remark on the next query.

TITLE	BUDGET	start	end
BIOTECH	180000	2014-11-18	2015-02-27
BIOTECH	160000	2015-02-27	2015-06-15
SURVEYOR	310000	2015-10-30	2016-06-25
SURVEYOR	375000	2016-06-25	2016-12-20
SURVEYOR	345000	2016-12-20	2017-05-19

- What was (and when) the maximum budget of each project?

```

select TITLE, BUDGET, start, end
from   H_PROJECT P
where  BUDGET = (select max(BUDGET)
                  from   H_PROJECT
                  where  TITLE = P.TITLE);

```

**Script 10.10** - What was (and in what periods) the maximum budget of each project?

TITLE	BUDGET	start	end
BIOTECH	180000	2014-11-18	2015-02-27
SURVEYOR	375000	2016-06-25	2016-12-20
AGRO-2000	82000	2017-07-05	9999-12-31

## 10.6 Temporal queries with non temporal result

These queries explore time dependent data (H\_\* tables) but absorbs the time dimension, either by reducing the search space to a specific time point or through aggregation (statistical) functions *on the time boundaries*.

- Show the state of employee D107 on 2017-02-01 (*snapshot*)

```

select CODE, NAME, SALARY, CITY, PROJECT
from   H_EMPLOYEE
where  CODE = 'D107'
and    start <= '2017-02-01' and '2017-02-01' < end;

```

**Script 10.11** - Snapshot of employee D107 on 2017-02-01

CODE	NAME	SALARY	CITY	PROJECT
D107	Delecourt	4100	Grenoble	SURVEYOR

- How many employees were assigned to the SURVEYOR project on 2017-02-01?

```

select count(distinct CODE) as Staff
from   H_EMPLOYEE
where  PROJECT = 'SURVEYOR'
and    start <= '2017-02-01' and '2017-02-01' < end;

```

**Script 10.12** - How many employees were assigned to the SURVEYOR project on 2017-02-01?

```

+-----+
| Staff |
+-----+
| 3     |
+-----+

```

- How many projects has employee N240 been assigned to throughout her career?

```

select count(distinct PROJECT) as Projects
from   H_EMPLOYEE
where  CODE = 'N240';

```

**Script 10.13** - How many projects has the N240 employee been assigned to throughout her career?

```

+-----+
| Projects |
+-----+
| 3       |
+-----+

```

- How many projects was employee M158 assigned to while living in Paris?

```

select count(distinct PROJECT) as Projects
from   H_EMPLOYEE
where  CODE = 'M158' and CITY = 'Paris';

```

**Script 10.14** - How many projects was employee M158 assigned to while living in Paris?

```

+-----+
| Projects |
+-----+
| 2       |
+-----+

```

- Which project(s) has employee N240 been assigned to the longest?

*Note:* SQLite does not allow subquery quantifiers *any* or *all*. In this expression, the subquery selects the highest value within the durations of employee N240 in her projects.

For current states, when the employee still is a member of the staff of the project, the *end* value in the computation of the duration is replaced by the current date. As already noticed, this query is non deterministic.

```

select PROJECT,
       sum(durationDays(start,min(end,'$date$')) as Duration
from   H_EMPLOYEE
where  CODE = 'N240'
group by PROJECT
having Duration =
       (select sum(durationDays(start,min(end,'$date$')) as Dur
from     H_EMPLOYEE
where    CODE = 'N240'
group by PROJECT
order by Dur desc limit 1);

```

**Script 10.15** - Which project(s) has employee N240 been assigned to the longest?

```

+-----+-----+
| PROJECT | Duration |
+-----+-----+
| BIOTECH | 703      |
+-----+-----+

```

## 10.7 Complex temporal queries

The queries developed so far are fairly mundane, once the bases of SQL have been mastered. Unfortunately, and surprisingly, other seemingly harmless queries will require much more complex SQL expressions. We will examine and resolve three conventional operators that require the greatest care when applied to temporal data, namely,

1. temporal projection (Section 10.8),
2. temporal join (Section 10.9) and outer join (Section 10.10).
3. temporal aggregation (Section 10.11).

Thanks to these operators, we will be able to resolve, sometimes in several steps, most of the questions that are addressed to a temporal database, and which are currently beyond our reach.

## 10.8 Temporal projection

The *projection* is the simplest operator that can be applied to non temporal data. It consists in restricting the rows of a selection to certain columns of the initial table. Considering the PROJECT table, its projection on columns TITLE and THEME, therefore discarding BUDGET values, is expressed in SQL as follows:

```
select TITLE, THEME
from PROJECT;
```

**Script 10.16** - SQL expression of the projection of PROJECT on columns TITLE and THEME

If the columns of the *select list* do not include all the components of a unique (or primary) key, then it is recommended to remove duplicates through the *distinct* modifier:

```
select distinct THEME
from PROJET
```

**Script 10.17** - SQL expression of a projection that produces a set of unique values

### 10.8.1 Entity-based temporal projection

We first study a simple variant of projection, in which the target columns include all the components of a unique key. We will call it *entity-based projection*.

Let us apply the pattern of Script 10.16 to table H\_PROJECT, from which we want to display not only the couples of (TITLE,THEME) values of each project but also the intervals in which each couple was valid. Let us try this query:

```
select TITLE, THEME, start, end
from H_PROJECT;
```

**Script 10.18** - First attempt to compute the projection of table H\_PROJECT on columns TITLE and THEME

With this result:

TITLE	THEME	start	end
AGRO-2000	Crop improvement	2016-08-31	2017-04-14
AGRO-2000	Crop improvement	2017-04-14	2017-07-05
AGRO-2000	Crop improvement	2017-07-05	9999-12-31
BIOTECH	Biotechnology	2014-11-18	2015-02-27
BIOTECH	Genetic engineering	2015-02-27	2015-06-15
BIOTECH	Genetic engineering	2015-06-15	2016-08-23
BIOTECH	Genetic engineering	2016-08-23	2017-09-17
BIOTECH	Biotechnology	2017-09-17	9999-12-31
SURVEYOR	Satellite monitoring	2015-10-30	2016-06-25
SURVEYOR	Satellite monitoring	2016-06-25	2016-12-20
SURVEYOR	Satellite monitoring	2016-12-20	2017-05-19

**Figure 10.1** - Projecting H\_PROJECT on TITLE, THEME - First (failing) trial

Though these rows are not incorrect, anyway they represent true facts, the way they display these facts is not what we expected. Let us add the clause `order by start` to this query, and the result, though equivalent, becomes unreadable. The problem is that these rows include several consecutive identical states, generally called *value-equivalent states*. For instance, the first three rows of this result should be merged into this single row:

```
| AGRO-2000 | Crop improvement      | 2016-08-31 | 9999-12-31 |
```

In order to merge these states, let us try this query, that only keep the lowest and highest time limits of consecutive value-equivalent states.

```
insert into H_THEME_2
select TITLE,THEME,min(start) as "Start",max(end) as "End"
from   H_PROJECT
group by TITLE,THEME;
```

**Script 10.19** - Second (still failing) attempt to compute the projection of table H\_PROJECT on columns TITLE and THEME

TITLE	THEME	Start	End
AGRO-2000	Crop improvement	2016-08-31	9999-12-31
BIOTECH	Biotechnology	2014-11-18	9999-12-31
BIOTECH	Genetic engineering	2015-02-27	2017-09-17
SURVEYOR	Satellite monitoring	2015-10-30	2017-05-19

**Figure 10.2** - Projecting H\_PROJECT on TITLE, THEME - Second (failing) trial

At first glance, the result looks much better, until we observe a curious phenomenon: one of the two states of project BIOTECH is embedded within the other one! So, these rows tell that, from 215-02-27 to 2017-09-17, this project was in two distinct states, with two different values of THEME, which is absurd.

Obviously, the correct answer should be the following:

TITLE	THEME	start	end
AGRO-2000	Crop improvement	2016-08-31	9999-12-31
BIOTECH	Biotechnology	2014-11-18	2015-02-27
BIOTECH	Genetic engineering	2015-02-27	2017-09-17
BIOTECH	Biotechnology	2017-09-17	9999-12-31
SURVEYOR	Satellite monitoring	2015-10-30	2017-05-19

**Figure 10.3** - Projecting H\_PROJECT on TITLE, THEME - The correct expected result

When we compare the rows of Figures 10.1 and 10.3, we immediately see that each row of the correct result is formed by merging, or reducing<sup>3</sup>, a series of consecutive value-equivalent rows in Figure 10.1. We call this series a *maximal suite of identical consecutive states* or MSICS.

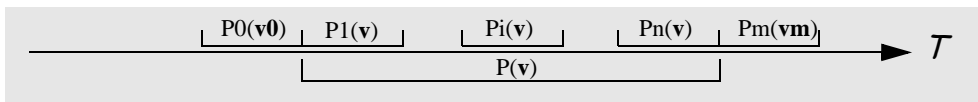
In temporal table **R**, an MSICS comprises *one or several* consecutive states (**s1**, **s2**, ..., **sn**) such that:

- all these states are value-equivalent
- either **s1** is the first state of **R** or its previous state is not value-equivalent; in short, the first state does not follow a value-equivalent state
- either **sn** is the last state of **R** or its next state is not value-equivalent; in short, the last state is not followed by a value-equivalent state.

Practically, if **R** represents project states, an MSICS **P** is defined by two project states (**P1**, **Pn**), such that:

- **P1** precedes **Pn** (unless the MSICS comprises one state only),
- **P1** and **Pn** are value-equivalent; let us call their value **v**,
- there is no state **P0** with value **v** that directly precedes **P1**,
- there is no state **Pm** with value **v** that directly follows **Pn**,
- all the states between **P1** and **Pn** have the same value **v**; or, reversing the condition, no state between **P1** and **Pn** has a value different from **v**.

These conditions are depicted in Figure 10.4.



**Figure 10.4** - Definition of MSICS P comprising states (P1,..., Pn)

The algorithm reduces each MSICS to the unique state (v,s1,en). We express it through the SQL query of Script 10.20

The first two conditions on subquery could be merged into this one:

```
and not exists (select * from H_PROJECT P0
                where (P0.TITLE, P0.THEME) = (P1.TITLE, P1.THEME)
                and   (P0.end = P1.start
                     or Pn.end = P0.start))
```

3. The technical term for this reduction operation is *coalescing*. Since SQL already includes a function called coalesce, we will keep the name *reduction*.

```

select P1.TITLE, P1.THEME, P1.start, Pn.end
from   H_PROJECT P1, H_PROJECT Pn
where  (P1.TITLE, P1.THEME) = (Pn.TITLE, Pn.THEME)
and    P1.start <= Pn.start
and not exists (select * from H_PROJECT P0
                where  (P0.TITLE, P0.THEME) = (P1.TITLE, P1.THEME)
                and    P0.end = P1.start)
and not exists (select * from H_PROJECT Pm
                where  (Pm.TITLE, Pm.THEME) = (P1.TITLE, P1.THEME)
                and    Pn.end = Pm.start)
and not exists (select * from H_PROJECT Pi
                where  Pi.TITLE = P1.TITLE
                and    Pi.THEME <> P1.THEME
                and    P1.end <= Pi.start
                and    Pi.end <= Pn.start);

```

**Script 10.20** - Projecting temporal table H\_PROJECT on columns (TITLE,THEME)

However, performance tests show that this concise form is *five times slower* than the former one (observed with SQLite 3.28). So, we abandon the idea.

This algorithm works fine if the following strict conditions are met:

- The temporal table records the history of a set of *similar entities*, e.g., *employees, projects*, etc. These entities are identified by an entity primary key, e.g., TITLE for projects in H\_PROJECT or CODE for employees in H\_EMPLOYEE.
- The history of each entity is *normalized*, that is, it includes no *gaps*, no *overlapping states*, even though the latter are value-equivalent and no *consecutive value-equivalent states*.
- The projection columns include the primary key of the entities, e.g., TITLE, THEME for H\_PROJECT or CODE, CITY, PROJECT for H\_EMPLOYEE.

Otherwise, it fails to produce a correct result. We will cope with this problem in Section 10.8.3, but, before, we will study ways to represent missing information.

### 10.8.2 Temporal projection with null values

Let us suppose that we do not always know the city in which an employee has been living. The usual way to translate this (absence of) knowledge consists in setting column CITY to *null* (CITY must be declared nullable). Figure 10.5 shows that the city of employee A68 was unknown during interval [2015-10-30, 2016-10-22).

In the projection of H\_EMPLOYEE on CODE, CITY, how do we represent the missing information on employee A68? Do we generate a row telling that the value of CITY is *null* during this interval, or do we merely discard such row? Figure 10.6 shows the state of the projection according to these alternatives.



CODE	NAME	SALARY	CITY	PROJECT	start	end
...	...	...	...	...	...	...
A68	Albert	3200	Toulouse	BIOTECH	2015-06-09	2015-10-30
A68	Albert	3700	--	SURVEYOR	2015-10-30	2016-10-22
A68	Albert	3900	Paris	SURVEYOR	2016-10-22	2017-05-19
...	...	...	...	...	...	...

**Figure 10.5** - Information may be unknown for some period

CODE	CITY	start	end
...	...	...	...
A68	Toulouse	2015-06-09	2015-10-30
A68	--	2015-10-30	2016-10-22
A68	Paris	2016-10-22	2017-05-19
...	...	...	...

CODE	CITY	start	end
...	...	...	...
A68	Toulouse	2015-06-09	2015-10-30
A68	Paris	2016-10-22	2017-05-19
...	...	...	...

**Figure 10.6** - Two ways to represent missing information in a projection

The first representation preserves *null* values, known to induce complexity in data processing. The second representation is more concise and more natural: the best way to represent the absence of information is ... the absence of representation!

On the other hand, rebuilding source table `H_EMPLOYEE` will use different operators, namely *temporal inner join* (Section 10.9) with the first representation and *temporal outer join* (Section 10.10) in the second one.

The algorithm of Script 10.20 generates projections in the first representation. Discarding *null* states is quite easy (Script 10.21).<sup>4</sup>

```
delete from H_CITY where CITY is null;
```

**Script 10.21** - Translating the first representation into the second one

### 10.8.3 Generalized temporal projection

Let us consider the projection of `H_EMPLOYEE` on column `CITY`, which is clearly not a unique key for employees. The result, shown below, should tell us in *which cities* and *when* at least one employee worked (or still is working) on a project:

4. More generally: `delete from H_T where (col1,..., coln) is (null,..., null)`

CITY	start	end
Genève	2016-03-22	2016-12-08
Genève	2016-12-29	2017-08-09
Genève	2016-12-29	9999-12-31
Grenoble	2016-02-16	2016-03-22
Grenoble	2016-09-16	9999-12-31
Grenoble	2016-09-16	9999-12-31
Grenoble	2016-09-16	9999-12-31
Grenoble	2016-09-16	9999-12-31
Grenoble	2016-12-08	9999-12-31
Grenoble	2016-12-08	9999-12-31
Paris	2015-06-15	2017-05-19
Paris	2016-10-22	2017-05-19
Paris	2016-10-22	9999-12-31
Toulouse	2015-06-09	2016-10-22
Toulouse	2016-08-18	2016-09-16
Toulouse	2017-05-19	2017-07-24

What do we observe?

- Many states overlap, which makes the data difficult to interpret.
- This history shows *gaps* (Genève was *inactive* between 2016-12-08 and 2016-12-29). Though such gaps do not necessarily represent anomalies, they are a new phenomenon.
- Worse, states are missing: For instance, employee G96 (Godin) worked on project BIOTECH from 2014-11-23 to 2016-08-31 while living in Genève. No trace of this fact in the result.

Actually, the result we expect is the following:

CITY	start	end
Genève	2014-11-23	9999-12-31
Grenoble	2016-02-16	2016-03-22
Grenoble	2016-09-16	9999-12-31
Lille	2015-08-22	9999-12-31
Paris	2014-12-04	9999-12-31
Toulouse	2015-06-09	2016-10-22
Toulouse	2017-05-19	2017-07-24

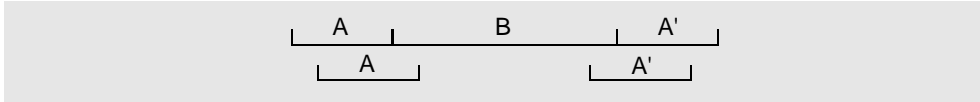
So, we conclude that the algorithm we just developed fails. Obviously, we must build a more general algorithm that solves all the projection patterns. In particular, it must apply to temporal tables which include:

- overlapping value-equivalent states,
- consecutive value-equivalent states,
- gaps.

In addition, it must tolerate incorrect data, according to which an entity is (erroneously) reported to be in more than one state. It must preserve them, leaving their processing to another process.

Before developing this improved algorithm, we define the concept of *interval extension*. Interval  $A$  extends interval  $B$  if their union has no gap and is larger than  $B$ . More precisely,

- interval  $A$  extends interval  $B$  to the left (Figure 10.7, left) either if  $B$  follows  $A$  ( $eA = sB$ ) or if  $A$  starts before  $B$  and  $A$  and  $B$  overlap ( $sA < sB$  and  $eA > sB$ ); in short, if  $sA < sB$  and  $eA \geq sB$ .
- interval  $A'$  extends interval  $B$  to the right (Figure 10.7, right), either if  $A'$  follows  $B$  ( $sA' = eB$ ) or if  $B$  starts before  $A'$  and  $A'$  and  $B$  overlap ( $sA' < eB$  and  $eA' > eB$ ); in short, if  $sA' \leq eB$  and  $eA' > eB$ .



**Figure 10.7** - Two ways for  $A$  (or  $A'$ ) to extend  $B$

The key concept of *maximum suite of identical consecutive states* (MSICS) must be extended to cope with overlapping states. So, we will consider reducing each *maximal suite of identical consecutive or overlapping states* (or MSICOS). The suite  $(S1, \dots, Sn)$  is a MSICOS that must be reduced to  $(v, s1, en)$  if and only if:

1.  $S1$  and  $Sn$  have the same value  $v$
2. either  $S1 = Sn$  or  $S1$  starts before  $Sn$  and does not ends after  $Sn$  ( $s1 \leq sn$  and  $e1 \leq en$ )
3. there is no state  $S0$  with value  $v$  that would extend  $S1$  to the left
4. there is no state  $Sm$  with value  $v$  that would extend  $Sn$  to the right
5. there is no state of the MSICOS between  $S1$  and  $Sn$  with a value different from  $v$
6. there is no missing state (gap) in the MSICOS between  $S1$  and  $Sn$ .

Conditions 1 to 5 are fairly easy to express in SQL: they are simple extensions of those of Script 10.20. However, condition 6 is more complicated since it describes the *absence of non-existent objects*.

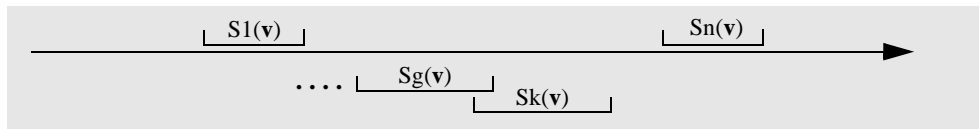
The algorithm we will develop is based on the idea described in [Böhlen,1996]<sup>5</sup>. Let us consider state  $Sg$  with value  $v$ , that ends after  $e1$  and before  $sn$ , that is,  $e1 \leq eg < sn$ . If  $Sg$  is extended to the right by any state  $Sk$  with value  $v$ , then  $Sg$  is *not followed by a gap*. Hence the rule that implements condition 6:

5. Böhlen, M., R., Snodgrass, R., T., Soo, M., D., Coalescing in Temporal Databases, in *Proc. VLDB Conf.*, Sept. 1996

*if there exists a state  $S_g$  with value  $v$  such that  $e1 \leq e_g < s_n$ , this state must be extended to the right by a state  $S_k$  with value  $v$*

or, reversing this rule, which is easier to translate in SQL:

*there is no state  $S_g$  with value  $v$  such that  $e1 \leq e_g < s_n$ , that is not extended to the right by a state  $S_k$  with value  $v$*



**Figure 10.8** - Checking whether a gap exists between  $S_1$  and  $S_n$

The algorithm that implements conditions 1 to 6 is translated into the query of Script 10.22.

```
select distinct E1.CITY, E1.start, En.end
from   H_EMPLOYEE E1, H_EMPLOYEE En
where  E1.CITY = En.CITY
and    E1.start <= En.start and E1.end <= En.end
and not exists (select *
                 from   H_EMPLOYEE E0
                 where  E0.CITY = E1.CITY
                 and    E0.start < E1.start
                 and    E0.end >= E1.start)
and not exists (select *
                 from   H_EMPLOYEE Em
                 where  Em.CITY = E1.CITY
                 and    Em.start <= En.end
                 and    Em.end > En.end)
and not exists (select *
                 from   H_EMPLOYEE Eg
                 where  Eg.CITY = E1.CITY
                 and    E1.end <= Eg.end
                 and    Eg.end < En.start
                 and    not exists(select *
                                   from   H_EMPLOYEE Ek
                                   where  Ek.CITY = E1.CITY
                                   and    Eg.end < Ek.end
                                   and    Ek.start <= Eg.end));
```

**Script 10.22** - Projecting temporal table H\_EMPLOYEE on column CITY

Building this query is a nice intellectual exercise, but the result is not particularly easy to understand. In addition, executing this query on large temporal tables will probably be quite expensive (we will check this later).

Fortunately, we can develop a procedural algorithm that is much simpler and faster. Let us consider this elementary query

```
select CITY,start,end
from   H_EMPLOYEE
order by CITY,start;
```

that produces the following result:

CITY	start	end	
...	...	...	
Genève	...	...	
Grenoble	2016-02-16	2016-03-22	(a)
Grenoble	2016-09-16	2016-12-11	(b)
Grenoble	2016-09-16	2016-12-29	(c)
Grenoble	2016-12-08	2017-04-03	(d)
Grenoble	2016-12-11	2016-12-29	(e)
Grenoble	2016-12-29	9999-12-31	(f)
Grenoble	2017-04-03	9999-12-31	(g)
Lille	...	...	(h)
...	...	...	

If we read these rows sequentially, we get the rows of each city in chronological order, that is, in increasing (actually *not decreasing*) values of the start column. Let us focus on city Grenoble, for which we read the successive rows. We reason as follows:

- When we read row (a), we identify the first state of a MSICOS and we memorize it as its tentative reduction [Grenoble, 2016-02-16, 2016-03-22).
- We read row (b). We observe that its start value (2016-09-16) is greater than the end value of the current reduction (2016-03-22). This means that we encounter a gap that closes the current MSICOS. So, (Grenoble, 2016-02-16, 2016-03-22) is the (trivial) reduction of the current MSICOS and belongs to the projection we are computing. Row (b) becomes the first row of the next MSICOS. We initialize the new current reduction to (Grenoble, 2016-09-16, 2016-12-11).
- We read row (c). Since its start value (2016-09-16) is lower than the end value of the current reduction (2016-12-11), this row extends the latter. So, we modify the current reduction to (Grenoble, 2016-09-16, 2016-12-29). So far, rows (b) and (c) have been reduced.
- Successively reading rows (d), (e), (f) and (g) leads us to the same observation, each one extending the current reduction. At this point, the current MSICOS is formed with rows (b) to (g) and its current reduction is (Grenoble, 2016-09-16, 9999-12-31).
- Reading next row (h) gives us a different value of CITY. This closes the current MSICOS, confirms the current reduction (Grenoble, 2016-09-16, 9999-12-31) and starts a new MSICOS.

When all the source rows have been read, the projection of H\_EMPLOYEE on column CITY looks like:

CITY	start	end
...	...	...
Genève	...	...
Grenoble	2016-02-16	2016-03-22
Grenoble	2016-09-16	9999-12-31
Lille	...	...
...	...	...

Translating this iterative algorithm in SQLfast is straightforward (Script 10.23). Tuple (city, sta, end) represents the new row that has just been read and tuple (curCity, curSta, curEnd) stores the current state of the reduction. The blue section is the core of the procedure. It checks whether the new row extends the reduction or if it closes it.

```
create temp table H_CITY(City char(10),start date,end date);
set first = 1;
for city,sta,end = [select CITY,start,end
                    from   H_EMPLOYEE
                    order by CITY,start];
  if ($first$);
    set curCity,curSta,curEnd = $city$,$sta$,$end$;
    set first = 0;
  else;
    if ('$city$' = '$curCity$' and '$sta$' <= '$curEnd$');
      if ('$end$' > '$curEnd$') set curEnd = $end$;
    else;
      insert into H_CITY
      values ('$curCity$','$curSta$','$curEnd$');
      set curCity,curSta,curEnd = $city$,$sta$,$end$;
    endif;
  endif;
endfor;
if (not $first$)
  insert into H_CITY
  values ('$curCity$','$curSta$','$curEnd$');
```

**Script 10.23** - SQLfast procedure that stores in table H\_CITY the projection of H\_EMPLOYEE on column CITY

As with most scripting languages, iterative scripts written in SQLfast may be slow when running against large temporal tables. For this reason, the SQLfast distribution includes a Python library (**LTemp.py**) offering a series of efficient functions that process temporal data.

From this library, function **project** computes the projection of a source table on a list of columns and stores the result in a target table. The arguments of this function are expressed as a query in an SQL-like mini-language. For example, the projection computed by Scripts 10.22 and 10.23 can be described by expression:

```
select CITY from H_EMPLOYEE into H_CITY
```

A where clause can also be specified:

```
select CITY from H_EMPLOYEE where SALARY > 6000 into H_CITY
```

This function is used as shown in Script 10.24.<sup>6</sup>

```
function status = LTemp:project
{select CITY from H_EMPLOYEE into H_CITY};
if ($status$ = 0) select * from H_CITY;
```

**Script 10.24** - Using function **project** from library LTemp

A simplified version of this function, called **reduce**,<sup>7</sup> modifies the source table itself by reducing its MSICOS. Script 10.25 shows how it can be used to return the projection of H\_EMPLOYEE on column CITY. It is useful notably to reduce intermediate results in complex temporal data processing, for instance when computing aggregates, as we will see later.

```
create temp table H_CITY as
select CITY,start,end from H_EMPLOYEE where SALARY > 6000;
function status = LTemp:reduce {select CITY from H_CITY};
if ($status$ = 0) select * from H_CITY;
```

**Script 10.25** - Using function **reduce** from library LTemp

## 10.9 Temporal join

In non temporal databases, *joining* tables, that is, applying a *join* operator, is the most common way to combine data from several tables. For example, extending the data of current employees with the theme of the project they are working on will be performed by *joining* views EMPLOYEE and PROJECT (Script 10.26).

6. Result *status* can be ignored, so that we can also write:

```
function LTemp:project {select CITY from H_EMPLOYEE into H_CITY}
```

7. Reminder: the standard name of the **reduce** operator in the vocabulary of temporal databases, is **coalesce**.

```

select CODE,NAME,E.PROJECT,THEME
from   EMPLOYEE E,PROJECT P
where  E.PROJECT = P.TITLE;

select CODE,NAME,E.PROJECT,THEME
from   EMPLOYEE E join PROJECT P
      on (E.PROJECT = P.TITLE);

```

**Script 10.26** - Two common forms for *joining* non temporal tables

These forms of *join* produce the same result:

CODE	NAME	PROJECT	THEME
C45	Carlier	BIOTECH	Biotechnology
A237	Antoine	AGRO-2000	Crop improvement
M158	Mercier	BIOTECH	Biotechnology
D107	Delecourt	BIOTECH	Biotechnology
G96	Godin	AGRO-2000	Crop improvement

Just like for the *projection*, studying the *join* operator between temporal tables starts with a naive trial (10.27).

```

select CODE,NAME,E.PROJECT,THEME,E.start,E.end
from   H_EMPLOYEE E,H_PROJECT P
where  E.PROJECT = P.TITLE
and    E.start = P.start;

```

**Script 10.27** - Joining temporal tables H\_EMPLOYEE and H\_PROJECT - First trial

The result below shows that the result is wrong. Since employees at each instant of their life worked on a project, the result must include at least one row for each employee state, which is not the case.

CODE	NAME	PROJECT	THEME	start	end
D122	Declercq	BIOTECH	Genetic engi[...]	2015-06-15	2016-03-21
A68	Albert	SURVEYOR	Satellite mo[...]	2015-10-30	2016-10-22
M158	Mercier	SURVEYOR	Satellite mo[...]	2015-10-30	2017-01-14
C45	Carlier	AGRO-2000	Crop improve[...]	2016-08-31	2016-12-11
G96	Godin	AGRO-2000	Crop improve[...]	2016-08-31	2016-12-11

This query clearly does not work as intended. Rows from both tables are joined on the basis of rare and independent events: when a change of state of an employee accidentally coincides with an unrelated change of state of his project.



Let us solve a simple example of what we expect when joining both tables. First, we consider state [2015-02-21, 2015-10-30] of employee M158 (Mercier, then living in Paris):

CODE	NAME	CITY	PROJECT	start	end
M158	Mercier	Paris	BIOTECH	2015-02-21	2015-10-30

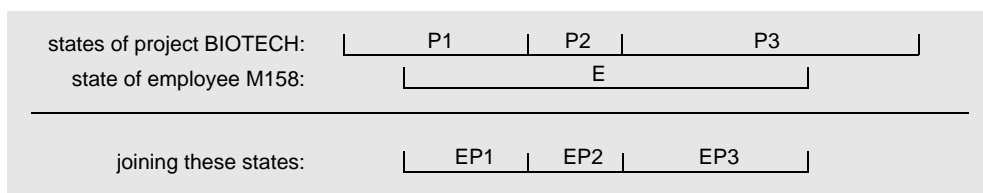
The interval of this state overlap with the interval of several states of table H\_PROJECT. So, we conclude that this state *references*, not one state of H\_PROJECT, as is the rule in non temporal data, but **all the states** of H\_PROJECT whose interval **overlaps** with that of M158 employee state. They are easy to identify:

TITLE	THEME	BUDGET	start	end
BIOTECH	Biotechnology	180000	2014-11-18	2015-02-27
BIOTECH	Genetic eng[...]	160000	2015-02-27	2015-06-15
BIOTECH	Genetic eng[...]	120000	2015-06-15	2016-08-23

Joining these rows produces the expected result:

CODE	NAME	..	PROJECT	..	start	end
M158	Mercier	..	BIOTECH	..	2015-02-21	2015-02-27
M158	Mercier	..	BIOTECH	..	2015-02-27	2015-06-15
M158	Mercier	..	BIOTECH	..	2015-06-15	2015-10-30

The reasoning can be represented graphically as shown in Figure 10.9.



**Figure 10.9** - Joining state E of table E\_EMPLOYEE with overlapping states P1, P2, P3 of table H\_PROJECT

We observe that the intervals of the first and last H\_PROJECT states have been reduced to align them to the interval of H\_EMPLOYEE, an operation called *interval clipping*.

Translating these operations into an SQL query is fairly easy (Script 10.28).

```

select CODE, NAME, PROJECT, THEME,
       max(P.start, E.start) as "start",
       min(P.end, E.end) as "end"
from   H_EMPLOYEE E, H_PROJECT P
where  E.PROJECT = P.TITLE
and    P.start < E.end and E.start < P.end;

```

**Script 10.28** - Temporal join of H\_EMPLOYEE with H\_PROJECT

Script 10.29 expresses the extraction of a snapshot from the join.

```

set T = 2016-01-01;
select CODE, NAME, PROJECT, THEME
from   (select CODE, NAME, PROJECT, THEME,
              max(P.start, E.start) as "Start",
              min(P.end, E.end) as "End"
        from   H_EMPLOYEE E, H_PROJECT P
        where  E.PROJECT = P.TITLE
        and    P.start < E.end and E.start < P.end)
where  "Start" <= '$T$' and '$T$' < "End";

```

**Script 10.29** - Snapshot of the temporal join of H\_EMPLOYEE with H\_PROJECT

CODE	NAME	PROJECT	THEME
G96	Godin	BIOTECH	Genetic engineering
D122	Declercq	BIOTECH	Genetic engineering
C45	Carlier	BIOTECH	Genetic engineering
A68	Albert	SURVEYOR	Satellite monitoring
M158	Mercier	SURVEYOR	Satellite monitoring

A more elegant (and reusable) translation of this snapshot is shown in Script 10.30. It defines the view that expresses the join then extracts the snapshot from this view.

```

create view H_EMP_PROJ (CODE, NAME, PROJECT, THEME, start, end) as
select CODE, NAME, PROJECT, THEME,
       max(P.start, E.start), min(P.end, E.end)
from   H_EMPLOYEE E, H_PROJECT P
where  E.PROJECT = P.TITLE
and    (P.start < E.end) and (E.start < P.end);

select CODE, NAME, PROJECT, THEME
from   H_EMP_PROJ
where  start <= '$T$' and '$T$' < end;

```

**Script 10.30** - View-based snapshot of the temporal join of H\_EMPLOYEE with H\_PROJECT

If this view is used in this query only, expressing it as a Common Table Expression (CTE) could be more appropriate.

## 10.10 Temporal outer join

Let us consider two tables resulting from the projection of table H\_EMPLOYEE:

- table H\_EMP1 comprises the projection of H\_EMPLOYEE on columns CODE, NAME and CITY ,
- table H\_EMP2 comprises the projection of H\_EMPLOYEE on columns CODE, SALARY and PROJECT.

Each column of the source table is included in one of these table, except CODE, the entity primary key, which is included in both.

We could think that tables H\_EMP1 and H\_EMP2 are equivalent to table H\_EMPLOYEE, that is, that joining the projections will produce the exact contents of H\_EMPLOYEE. So, we could get rid of H\_EMPLOYEE and replace it with H\_EMP1 and H\_EMP2, since, if needed, we always could rebuild H\_EMPLOYEE.

This is true under two strict conditions:

- CODE is the entity primary key in each table<sup>8</sup>,
- for each state in H\_EMP1 with interval  $v_1$ , there exists in H\_EMP2 a sequence of contiguous states with the same value of CODE that covers  $v_1$ , and conversely.

Said in other ways,

- column CODE in each of these tables is a temporal foreign key to the other,
- the temporal projections of these tables on CODE are identical.

### The need for outer joins

What would happen if a state is missing in one of the tables, H\_EMP2 for example (see Section 10.8.2)? Simple: some of the matching state(s) in the other table (H\_EMP1) will be lost in the join. This would be the case, for example, if, in some states of the source table, both SALARY and PROJECT are *null* while columns NAME and/or CITY are not *null*. These states in H\_EMPLOYEE would generate states in H\_EMP1 but none in table H\_EMP2, leaving gaps in this table. In joining H\_EMP1 and H\_EMP2 into H\_EMP12, we clearly lose information, so that H\_EMP12 and H\_EMPLOYEE no longer are equivalent.

In *non temporal* databases, this problem can be solved with a variant of the join operator named *outer join*. When joining two tables T1 and T2, a missing row in

---

8. More generally, CODE is a unique key in each snapshot of these tables.

table T2 is replaced by *null* values, therefore preserving the data of T1. Reading the join arguments from left to right, this form is called *left outer join*. If the missing rows are in T1, we name it *right outer join*. If both tables may have missing rows, the operator is a *full outer join*. In the relational model, the ordinary join is more precisely called *inner join*, as opposed to expression *outer join*.

### Temporal outer join: description

To study the outer join operator applied to temporal data, we consider temporal tables PRO and EMP of Figure 10.10. They are reduced versions of tables H\_PROJECT and H\_EMPLOYEE studied so far. Table PRO describes projects 'p1' and 'p2' while table EMP describes the evolution of employees 'e1' and 'e2'. Columns ProID and EmpID are entity primary keys of their respective tables. Column Proj of EMP is a foreign key to table PRO.

table PRO					table EMP				
ProID	Name	Budg	start	end	EmpID	Sal	Proj	start	end
p1	bio	120	2	4	e1	10	p1	1	3
p1	bio	105	6	9	e1	12	p1	3	5
p1	bio	130	10	12	e1	15	p1	6	7
p1	bio	142	12	13	e1	16	p1	8	11
p2	tec	125	3	5	e2	12	p1	3	6
p2	tec	130	5	8	e2	15	p2	7	10
p2	tec	142	10	12	e2	16	p2	11	12

Figure 10.10 - The two source temporal tables to study the outer join operator

The states of these tables are fairly *chaotic*: the history of each table is incomplete, and even incorrect: several states are missing and referential integrity is violated. Some PRO states do not match any EMP state while some EMP states do not match PRO states. The idiosyncrasies of the data are better shown in Figure 10.11. In short, a good basis to study the join of the table on Proj = ProID!

	1	2	3	4	5	6	7	8	9	10	11	12	13
PRO (p1) :													
EMP (e1.p1) :	<---	<---	<---	<---	<---	<---	<---	<---	<---	<---	<---	<---	<---
EMP (e2.p1) :			<---	<---	<---	<---	<---	<---	<---	<---	<---	<---	<---
PRO (p2) :			<---	<---	<---	<---	<---	<---	<---	<---	<---	<---	<---
EMP (e1.p2) :			<---	<---	<---	<---	<---	<---	<---	<---	<---	<---	<---
EMP (e2.p2) :			<---	<---	<---	<---	<---	<---	<---	<---	<---	<---	<---

Figure 10.11 - The time lines of each PRO entity and its associated EMP entities

As a consequence, a standard temporal join (Figure 10.12) will lose data from both tables. Figure 10.13 shows that attempting to recover the source data from their inner join through temporal projection (Figure 10.13) entails the loss of many states. This is particularly clear in the synthetic view of Figure 10.14.

EmpID	Sal	ProID	Name	Budg	Start	End
e1	10	p1	bio	120	2	3
e1	12	p1	bio	120	3	4
e1	15	p1	bio	105	6	7
e1	16	p1	bio	105	8	9
e1	16	p1	bio	130	10	11
e2	12	p1	bio	120	3	4
e2	15	p2	tec	130	7	8
e2	16	p2	tec	142	11	12

**Figure 10.12** - Inner join of PRO and EMP on columns ProID = Proj

table PRO					table EMP				
ProID	Name	Budg	start	end	EmpID	Sal	Proj	start	end
p1	bio	120	2	4	e1	10	p1	2	3
p1	bio	105	6	7	e1	12	p1	3	4
p1	bio	105	8	9	e1	15	p1	6	7
p1	bio	130	10	11	e1	16	p1	8	9
p2	tec	130	7	8	e1	16	p1	10	11
p2	tec	142	11	12	e2	12	p1	3	4
					e2	15	p2	7	8
					e2	16	p2	11	12

**Figure 10.13** - (Unsuccessful) tentative recovery of source data from their inner join

	1	2	3	4	5	6	7	8	9	10	11	12	13
PRO (p1) :		<-->	<-->			<-->		<-->		<-->			
EMP (e1.p1) :		<-->	<-->			<-->		<-->		<-->			
EMP (e2.p1) :			<-->										
PRO (p2) :							<-->				<-->		
EMP (e1.p2) :							<-->				<-->		
EMP (e2.p2) :							<-->				<-->		

**Figure 10.14** - The time lines of each PRO entity and its associated EMP entities derived from the inner join of their tables

Let us now go back to the concept of *outer join* applied to temporal data.

Considering tables PRO and EMP, *in this order*, the *temporal left outer join* preserves each state p of PRO, extended with the values of EmpID and Sal of each row of EMP the interval of which overlaps with the interval of p. When no row matches in EMP, p is extended with *null* values. Figure 10.15 shows the result of the *left outer join* of PRO with EMP. We observe that all states of PRO are preserved, sometimes complemented by *null* values when no matching state has been found in EMP.

ProID	Name	Budg	EmpID	Sal	Start	End
p1	bio	105	--	--	7	8
p1	bio	130	--	--	11	12
p1	bio	142	--	--	12	13
p1	bio	120	e1	10	2	3
p1	bio	120	e1	12	3	4
p1	bio	105	e1	15	6	7
p1	bio	105	e1	16	8	9
p1	bio	130	e1	16	10	11
p1	bio	120	e2	12	3	4
p2	tec	125	--	--	3	5
p2	tec	130	--	--	5	7
p2	tec	142	--	--	10	11
p2	tec	130	e2	15	7	8
p2	tec	142	e2	16	11	12

Figure 10.15 - Left outer join of PRO with EMP on columns ProID = Proj

The *right outer join* of PRO with EMP, that preserves all the states of EMP, is built similarly. The *full outer join* preserves the data of both source tables. Its result is shown in Figures 10.16. It can be computed as the union of the results of the *left* and *right outer joins*.

ProID	Name	Budg	EmpID	Sal	Start	End
p1	bio	105	--	--	7	8
p1	bio	130	--	--	11	12
p1	bio	142	--	--	12	13
p1	--	--	e1	10	1	2
p1	bio	120	e1	10	2	3
p1	bio	120	e1	12	3	4
p1	--	--	e1	12	4	5
p1	bio	105	e1	15	6	7
p1	bio	105	e1	16	8	9
p1	--	--	e1	16	9	10
p1	bio	130	e1	16	10	11
p1	bio	120	e2	12	3	4
p1	--	--	e2	12	4	6
p2	tec	125	--	--	3	5
p2	tec	130	--	--	5	7
p2	tec	142	--	--	10	11
p2	tec	130	e2	15	7	8
p2	--	--	e2	15	8	10
p2	tec	142	e2	16	11	12

Figure 10.16 - Full outer join of PRO with EMP on columns ProID = Proj

### Implementing temporal outer joins

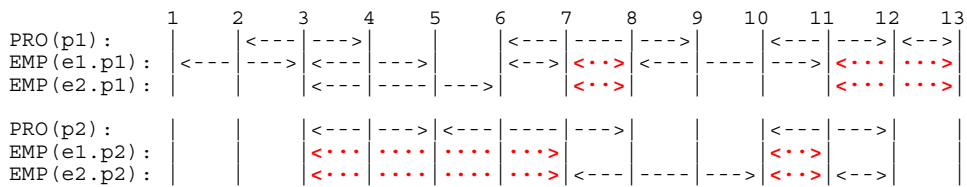
Now, we have to find an algorithm that computes the *left outer join* of PRO and EMP based on the observations we made here above. To this aim, we will apply a well known problem solving strategy: *transforming our problem into another one already solved*.

The examination of EMP shows a set of *missing intervals* wrt of PRO. An interval (t1,t2) is called *missing* in EMP wrt project entity p if:

- (t1,t2) exists in PRO for p,
- there no consecutive states in EMP for which Proj = p and the intervals of which cover (t1,t2).

The missing interval of EMP wrt PRO are represented in red in Figure 10.17. The history of project p1 has no matching states in EMP for intervals [7,8) and [11,13). So, they are the four missing intervals of EMP.

In an *inner join*, these intervals will lead to the loss of the matching intervals in PRO, and in a *left outer join*, they will complement the states of PRO with *null* values.



**Figure 10.17** - Identifying missing intervals in the time lines of each PRO entity and of its associated EMP entities

Let us fill them with *null* values. Figure 10.18 shows the contents of table EMP in which we have inserted, for each missing interval, a row made of *null* values for columns EmpID and Sal.

EmpID	Sal	Proj	start	end
e1	10	p1	1	3
e1	12	p1	3	5
e2	12	p1	3	6
e1	15	p1	6	7
--	--	p1	7	8
e1	16	p1	8	11
--	--	p1	11	13
--	--	p2	3	7
e2	15	p2	7	10
--	--	p2	10	11
e2	16	p2	11	12

**Figure 10.18** - Table EMP complemented with *null* missing intervals wrt PRO

This modified table EMP is just a transient working data set. It is incorrect since it violates the primary key and *not null* constraints. However it exhibits an interesting property: it constitutes a complete history of employees. This means that each state of PRO will match with at least one state in modified EMP. This has an important consequence: the *left outer join* of PRO with EMP can be computed through a simple *inner join*.

The missing states are computed and stored in table EMP\_MISSING by Script 10.31, then added to the source states to constitute table EMP\_COMPLETED (Script 10.32).

```
create table EMP_MISSING as
  select EMPa.Proj, null as "EmpID", null as "Sal",
         EMPa.end as Tstart, EMPb.start as Tend
  from   EMP EMPa, EMP EMPb
  where  EMPa.Proj = EMPb.Proj
  and    EMPa.end < EMPb.start
  and not exists (select *
                  from   EMP EMPc
                  where  EMPc.Proj = EMPa.Proj
                  and    EMPc.start < EMPb.start
                  and    EMPa.end < EMPc.end)

  union

  select EMP.Proj, null, null,
         min(PRO.start) as Tstart, min(EMP.start) as Tend
  from   PRO, EMP
  where  PRO.ProID = EMP.Proj
  group by EMP.Proj
  having Tstart < Tend

  union

  select EMP.Proj, null, null,
         max(EMP.end) as Tstart, max(PRO.end) as Tend
  from   PRO, EMP
  where  PRO.ProID = EMP.Proj
  group by EMP.Proj
  having Tstart < Tend

  union

  select null as EmpID, null as Sal, ProID as Proj,
         min(start) as "start", max(end) as "end"
  from   PRO
  where  ProID not in (select Proj from EMP)
  group by ProID;
```

**Script 10.31** - Generating the missing states of source table EMP

```
create table EMP_COMPLETED as
  select Proj, EmpID, Sal, start as Tstart, end as Tend from EMP
  union
  select Proj, null, null, Tstart, Tend from EMP_MISSING;
```

**Script 10.32** - Table EMP is completed with its missing states

The algorithm that computes the missing intervals is made up of four parts:

- The first part extracts the *internal* missing intervals, that is, those that constitutes gaps in EMP.



- The second and third parts create left and right *external* missing intervals when the history of the current project starts before and/or ends after the series of states of EMP that reference this project. For instance, the third part creates missing state [11,13) of project p1 in EMP.
- The fourth part copes with project entities that are **not** referenced by employees. A *null* state is created in EMP for each value of ProID in PRO that does not appear in column Proj in EMP. By the way, this part also fill **empty** table EMP with a *null* state for each project that will preserve all the states of PRO.

Finally, a simple inner join of PRO with EMP\_COMPLETED produces the desired *left outer join* in table LEFT (Script 10.33). A similar procedure will generate the right outer join in table RIGHT.

```
create table LEFT as
select PRO.ProID, Name, Budg, EmpID, Sal,
       max(PRO.start, EMPC.Tstart) as "Start",
       min(PRO.end, EMPC.Tend) as "End"
from   PRO, EMP_COMPLETED EMPC
where  PRO.ProID = EMPC.Proj
and    (PRO.start < EMPC.Tend) and (EMPC.Tstart < PRO.end);
```

**Script 10.33** - A simple inner join produces the *left outer join* of PRO with EMP

The full outer join is computed as the union of LEFT and RIGHT (Script 10.34).

```
select * from LEFT union select * from RIGHT;
```

**Script 10.34** - Full outer join of PRO and EMP

### Computing outer joins with the LTemp library

The completed version of EMP *wrt* PRO, called above EMP\_COMPLETED, can be more simply (and generally much faster) computed by function **normalize** of temporal library **LTemp**. Its argument is a pseudo-SQL query that specifies the source table to be completed (EMP), the reference table (PRO), the join condition (Proj = ProID) and the target table (EMP\_COMPLETED).

```
function LTemp:normalize {update EMP wrt PRO on Proj = ProID
                        into EMP_COMPLETED};
```

**Script 10.35** - Computing EMP\_COMPLETED through function **normalize** of library **LTemp**

If clause '**wrt PRO**' is missing, only internal missing intervals of EMP are added. If clause '**into EMP\_COMPLETED**' is missing, the missing intervals are inserted into table EMP itself. So, the following form is also valid:

```
function LTemp:normalize {update EMP on Proj};
```

### Reversibility of the outer joins

To convince us that outer joins preserve one or both of their source arguments, just check that the script below produces the exact contents of table EMP:

```
function LTemp:project {select EmpID,Sal,ProID from FULL
                        into NewEMP};
delete from NewEMP where (EmpID,Sal) is (null,null);
```

## 10.11 Temporal aggregation

In general, an aggregation query for a non-temporal table provides a unique quantity calculated on a set of rows of that table (Script 10.36, top), or for each group of rows constituted according to a grouping criterion (Script 10.36, bottom). Their results are shown in Figure 10.19. Let us remind that, according to the SQL standard(s), the *select list* (the data elements between key words **select** and **from**) may comprise *constants*, elements of the *grouping criterion* and *aggregate functions* only, as well as expressions comprising these elements.

```
select sum(BUDGET) as Tbudget
from   PROJECT

select PROJECT, count(*) as Nemp
from   EMPLOYEE
group by PROJECT;
```

**Script 10.36** - Aggregation queries on non temporal data

Tbudget	PROJECT	Nemp
222000	AGRO-2000	2
	BIOTECH	3

**Figure 10.19** - Aggregated data computed on non temporal data

Querying a snapshots is as easy, since it just is a *current state* that existed in the past. The queries of Script 10.37 show the same statistics extracted from the state on January 1st, 2017 and Figure 10.20 shows the result.

```
select sum(BUDGET) as Tbudget
from H_PROJECT
where start <= '2017-01-01' and '2017-01-01' < end;

select PROJECT, count(*) as Nemp
from H_EMPLOYEE
where start <= '2017-01-01' and '2017-01-01' < end
group by PROJECT;
```

**Script 10.37** - Aggregation queries on a snapshot

Tbudget	PROJECT	Nemp
550000	AGRO-2000	3
	BIOTECH	1
	SURVEYOR	4

**Figure 10.20** - Aggregated data computed on a snapshot (on 2017-01-01)

This was the easy part. Easy, because we have got rid of the time dimension before aggregating the data. Now we will cope with this dimension by deriving *evolution data*, which will prove a bit more complex, notably because the concept of time-dependent aggregation is not as easy to define as it seems to be at first glance.

### 10.11.1 A first simple problem: counting

Let us start with something (seemingly) simple: showing the evolution, from the creation of a project to now, of the number of employees who worked on this project. Figure 10.21 shows all the states of H\_EMPLOYEE related to project AGRO-2000, sorted by increasing values of column start.

Table H_EMPLOYEE								
PROJECT [8 rows]								
	CODE	NAME	SALARY	CITY	PROJECT	start	end	
1	C45	Carlier	3400	Lille	AGRO-2000	2016-08-31	2016-12-11	
2	G96	Godin	3900	Genève	AGRO-2000	2016-08-31	2016-12-11	
3	A237	Antoine	4000	Grenoble	AGRO-2000	2016-09-16	2016-12-11	
4	G96	Godin	4100	Genève	AGRO-2000	2016-12-11	2017-05-19	
5	N240	Nguyen	3700	Genève	AGRO-2000	2016-12-29	2017-08-09	
6	A237	Antoine	3800	Grenoble	AGRO-2000	2016-12-29	9999-12-31	
7	D122	Declercq	4100	Toulouse	AGRO-2000	2017-05-19	2017-07-24	
8	G96	Godin	3900	Genève	AGRO-2000	2017-05-19	9999-12-31	

Figure 10.21 - Subset of the H\_EMPLOYEE table describing the history of the staff of project AGRO-2000

In Figure 10.22, we represent graphically the sequence of the states of the employees who worked (and sometimes still work) on this project.

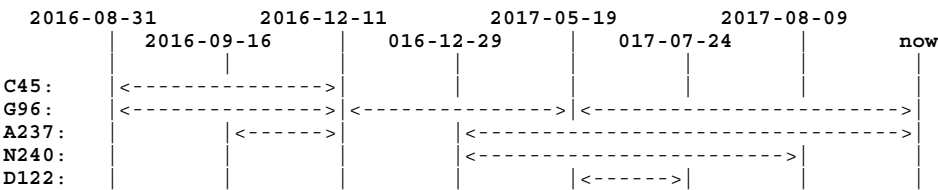


Figure 10.22 - Graphical representation of the life of the employees who worked on project AGRO-2000

To make intervals comparable, we split them into smaller intervals during which no modification occurred for any employee. We call them *stable intervals*. Now, each state of an employee can be defined as a sequence of one or more stable intervals. Since the set of stable intervals is common to all the employees of the project, counting the number of employees in each of these intervals is straightforward, as shown in figure 10.23.

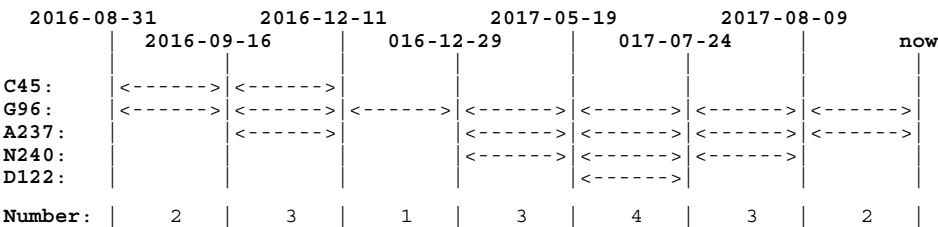


Figure 10.23 - Expressing employee states as a sequence of *stable intervals*

Hence the result we are looking for (Figure 10.24).

PROJECT	Number	Start	End
AGRO-2000	2	2016-08-31	2016-09-16
AGRO-2000	3	2016-09-16	2016-12-11
AGRO-2000	1	2016-12-11	2016-12-29
AGRO-2000	3	2016-12-29	2017-05-19
AGRO-2000	4	2017-05-19	2017-07-24
AGRO-2000	3	2017-07-24	2017-08-09
AGRO-2000	2	2017-08-09	9999-12-31

**Figure 10.24** - Evolution of the number of employees in project AGRO-2000

What this small example does not show is that two (or more) successive stable intervals may produce the same statistics. For instance, when, at the same date, an employee leaves a project while this project hires a new employee. In such a case we *reduce* this result by merging such value-equivalent states.

Now, we must implement these reasonings into SQL statements. Clearly, we must proceed in four steps:

- extracting all the events affecting the employees,
- creating the stable intervals,
- counting the number of employees in each stable interval
- and reducing value-equivalent stable interval.

#### a) Extracting all the events affecting the employees

Each event (i.e., data modification) generates a time point, which generally closes the current state and starts a new one, in which case this time point appears as both start and end values in two states. Except of course the first and last events of a series of successive states, which appear in a single state. To collect these time points, we just extract all the start and end values, without duplicates. Script 10.38 stores them in temporary table EVENT.

```
create temp table EVENT as
select PROJECT,Instant
from (select PROJECT,start as Instant from H_EMPLOYEE
union
select PROJECT,end as Instant from H_EMPLOYEE);
```

**Script 10.38** - Extracting the time points of table T\_EMPLOYEE

The contents of this table looks like:

PROJECT	Instant
AGRO-2000	2016-08-31
AGRO-2000	2016-09-16
AGRO-2000	2016-12-11
AGRO-2000	2016-12-29
AGRO-2000	2017-05-19
AGRO-2000	2017-07-24
AGRO-2000	2017-08-09
AGRO-2000	9999-12-31
BIOTECH	2014-11-23
...	...

### b) Creating the stable intervals

From table EVENT, we derive the *stable intervals*. Conceptually, the process is simple: with each event E, we associate the lowest Instant value among all the values greater than that of E. An SQL translation is proposed in Script 10.39, followed by the contents of table STABLE\_INTERVAL.

The way these intervals are built produces a complete normalized history, even though some intervals may have no value, for instance due to the fact that no project has any employee during them.

```
create temp table STABLE_INTERVAL as
select PROJECT,
       Instant as Start,
       (select min(Instant)
        from EVENT
        where PROJECT = E.PROJECT
         and Instant > E.Instant) as End
from   EVENT E
where  End is not null;
```

Script 10.39 - Building the stable intervals from table EVENT

PROJECT	Start	End
AGRO-2000	2016-08-31	2016-09-16
AGRO-2000	2016-09-16	2016-12-11
AGRO-2000	2016-12-11	2016-12-29
AGRO-2000	2016-12-29	2017-05-19
AGRO-2000	2017-05-19	2017-07-24
AGRO-2000	2017-07-24	2017-08-09
AGRO-2000	2017-08-09	9999-12-31
BIOTECH	2014-11-23	2014-12-04
...	...	...

Figure 10.25 - The stable intervals of project AGRO-2000

### c) Counting the number of states in each stable interval

Now, we must split the states of H\_EMPLOYEE according to the stable intervals computed in table STABLE\_INTERVAL to produce the eighteen interval instances of Figure 10.23. For this, we associate with each stable interval all the states of H\_EMPLOYEE with which they overlap. This typically is obtained by a temporal join (Script 10.40).

```
create temp view STABLE_EMPLOYEE as
select S.PROJECT, S.start, S.end
from   STABLE_INTERVAL S, H_EMPLOYEE E
where  E.PROJECT = S.PROJECT
and    E.start <= S.start and S.start < E.end;
```

**Script 10.40** - Computing the stable intervals of E\_EMPLOYEE

The contents of this view is shown below. Each row represents the contribution of one employee to a stable interval. This could be shown by adding E.CODE to the *select list* of the query.

PROJECT	Start	End
AGRO-2000	2016-08-31	2016-09-16
AGRO-2000	2016-08-31	2016-09-16
AGRO-2000	2016-09-16	2016-12-11
AGRO-2000	2016-09-16	2016-12-11
AGRO-2000	2016-09-16	2016-12-11
AGRO-2000	2016-12-11	2016-12-29
AGRO-2000	2016-12-29	2017-05-19
AGRO-2000	2016-12-29	2017-05-19
AGRO-2000	2016-12-29	2017-05-19
AGRO-2000	2017-05-19	2017-07-24
AGRO-2000	2017-05-19	2017-07-24
AGRO-2000	2017-05-19	2017-07-24
AGRO-2000	2017-05-19	2017-07-24
AGRO-2000	2017-07-24	2017-08-09
AGRO-2000	2017-07-24	2017-08-09
AGRO-2000	2017-07-24	2017-08-09
AGRO-2000	2017-08-09	9999-12-31
AGRO-2000	2017-08-09	9999-12-31
BIOTECH	2014-11-23	2014-12-04
...	...	...

Finally, applying the aggregate function is quite easy, since it is nothing more than a standard, non temporal, computation (Script 10.41).

```
select PROJECT, count(*) as Number, start, end
from   STABLE_EMPLOYEE
group by PROJECT, start, end;
```

**Script 10.41** - Finally, computing the evolution of the number of employees per project

The result is that of Figure 10.24. By developing the reference of the view in this query by its definition, we obtain the synthetic query of Script 10.42.

```
create temp table E_COUNT (PROJECT, Number, Start, End) as
select S.PROJECT, count(*), S.start, S.end
from   H_EMPLOYEE E, STABLE_INTERVAL S
where  E.PROJECT = S.PROJECT
and    E.start <= S.start and S.start < E.end
group by S.PROJECT, S.start, S.end;
```

**Script 10.42** - computing the evolution of the number of employees per project - Synthetic query

#### d) Reducing value-equivalent stable intervals

Consecutive value-equivalent states are merged through the reduce function (Script 10.43).

```
function LTemp:reduce {select PROJECT, Number from E_COUNT};
```

**Script 10.43** - Reducing value-equivalent states

### 10.11.2 Value-based statistics: avg, max and min

Counting states was an excellent exercise to help us solve more complex statistics. Let us start with avg, min and max aggregation functions.

We intend to compute the evolution of the average, minimum and maximum salary for each project. As we did for counting the employees, we start with table STABLE\_INTERVAL (Script 10.39 and Figure 10.25). We know that, by construction, the salary of employees never changed during any of those stable interval. So, we can add its unique salary to each interval to form view VALUED\_STABLE\_INTERVAL (Script 10.44).



```

create temp view VALUED_STABLE_INTERVAL as
select S.PROJECT, E.SALARY, S.start, S.end
from   STABLE_INTERVAL S, H_EMPLOYEE E
where  E.PROJECT = S.PROJECT
and    E.start <= S.start and S.start < E.end;

```

**Script 10.44** - Adding the salary value to each stable interval of E\_EMPLOYEE

The contents of this view is shown in Figure 10.26 and is graphically represented in Figure 10.27.

PROJECT	SALARY	Start	End
AGRO-2000	3400	2016-08-31	2016-09-16
AGRO-2000	3900	2016-08-31	2016-09-16
AGRO-2000	3400	2016-09-16	2016-12-11
AGRO-2000	3900	2016-09-16	2016-12-11
AGRO-2000	4000	2016-09-16	2016-12-11
AGRO-2000	4100	2016-12-11	2016-12-29
AGRO-2000	3700	2016-12-29	2017-05-19
AGRO-2000	3800	2016-12-29	2017-05-19
AGRO-2000	4100	2016-12-29	2017-05-19
AGRO-2000	3700	2017-05-19	2017-07-24
AGRO-2000	3800	2017-05-19	2017-07-24
AGRO-2000	3900	2017-05-19	2017-07-24
AGRO-2000	4100	2017-05-19	2017-07-24
AGRO-2000	3700	2017-07-24	2017-08-09
AGRO-2000	3800	2017-07-24	2017-08-09
AGRO-2000	3900	2017-07-24	2017-08-09
AGRO-2000	3800	2017-08-09	9999-12-31
AGRO-2000	3900	2017-08-09	9999-12-31
BIOTECH	...	...	...
...	...	...	...

**Figure 10.26** - The stable intervals of H\_EMPLOYEE showing their SALARY values (tabular view)

	2016-08-31	2016-12-11	2017-05-19	2017-08-09	now
	2016-09-16	016-12-29	017-07-24		
C45:	<-3400->	<-3400->			
G96:	<-3900->	<-3900->	<-4100->	<-3900->	<-3900->
A237:		<-4000->	<-3800->	<-3800->	<-3800->
N240:			<-3700->	<-3700->	<-3700->
D122:			<-4100->		
Average:	2650	3766.7	4100	3866.7	3875
Min:	3400	3400	4100	3700	3700
Max:	3900	4000	4100	4100	3900

**Figure 10.27** - The stable intervals of H\_EMPLOYEE showing their SALARY values (graphical view)

Computing the standard statistics (average, minimum, maximum) of SALARY in each stable state is just child's play (Script 10.45).

```
select  PROJECT,
        round(avg(SALARY),1) as Average,
        min(SALARY) as Min,
        max(SALARY) as Max,Start,End
from    VALUED_STABLE_INTERVAL
group by PROJECT,Start,End;
```

**Script 10.45** - Computing standard statistics of the stable interval of H\_EMPLOYEE

After reducing the value-equivalent intervals, we get this result:

PROJECT	Average	Min	Max	Start	End
AGRO-2000	3650.0	3400	3900	2016-08-31	2016-09-16
AGRO-2000	3766.7	3400	4000	2016-09-16	2016-12-11
AGRO-2000	4100.0	4100	4100	2016-12-11	2016-12-29
AGRO-2000	3866.7	3700	4100	2016-12-29	2017-05-19
AGRO-2000	3875.0	3700	4100	2017-05-19	2017-07-24
AGRO-2000	3800.0	3700	3900	2017-07-24	2017-08-09
AGRO-2000	3850.0	3800	3900	2017-08-09	9999-12-31
BIOTECH	...	...	...	...	...
...	...	...	...	...	...

### 10.11.3 Value-based statistics: sum

This last exercise is a bit more complex because it involves the *length of the intervals*. Still considering view VALUED\_STABLE\_INTERVAL (and Figure 10.27), we would like to know the *total salary cost* of each interval. Adding expression `sum(SALARY)` to query 10.45 is useless for two reasons: the cost we are looking for depends on the number of days in each interval and SALARY is a monthly salary.

- The number of days in interval  $[d1,d2)$  is given by function `durationDays(d1,d2)`. So, the length of an interval is obtained by expression `durationDays(start,end)`. However, this does not work for current states, for which `end = '9999-12-31'`. Replacing **end** with the current date (`$date$`) is also not satisfactory since its value changes from day to day, leading to unstable lengths. In such case, we suggest to replace **end** by the more realistic constant **maxValue**, computed as the highest **end** date of H\_EMPLOYEE + 100 days:

```
extract maxDate = select addToDate(max(end),100)
                    from    H_EMPLOYEE
                    where   end < '$Tfuture$';
```

The length of interval  $[start,end)$  will then be computed by expression:

```
durationDays(start,min(end,'$maxDate$'))
```

- Converting a monthly salary into a daily salary is a bit tricky. Indeed, months have different lengths but cost all the same salary, so that we are forced to work with some kind of average daily salary. On the average, in one year, 1 month comprises  $365/12 = 30.4167$  days. Let them be rounded to 30.42 days for this exercise.

Therefore, the salary cost of interval [start,end) is computed by expression:

`SALARY/30.42*durationDays(start,min(end,'$maxDate$'))`

The correct solution is shown in Script 10.46. Its result is shown in Figure 10.28, where TCost has been rounded to one decimal (`round()` function not shown).

```
select  PROJECT,
        sum(SALARY/30.42 *
            durationDays(Start,min(End,'$maxDate$'))) as TCost,
        Start,End
from    VALUED_STABLE_INTERVAL
group by PROJECT,Start,End;
```

**Script 10.46** - Computing the total salary cost of each stable interval

PROJECT	TCost	Start	End
AGRO-2000	3839.6	2016-08-31	2016-09-16
AGRO-2000	31946.1	2016-09-16	2016-12-11
AGRO-2000	2426.0	2016-12-11	2016-12-29
AGRO-2000	53767.3	2016-12-29	2017-05-19
AGRO-2000	33629.2	2017-05-19	2017-07-24
AGRO-2000	5996.1	2017-07-24	2017-08-09
AGRO-2000	25312.3	2017-08-09	9999-12-31
BIOTECH	...	...	...
...	...	...	...

**Figure 10.28** - Salary cost of stable intervals

The same query, slightly modified, will compute the total salary cost (so far) of each project (Script 10.47).

```
select  PROJECT,
        sum(SALARY/30.42 *
            durationDays(Start,min(End,'$maxDate$'))) as TCost
from    VALUED_STABLE_INTERVAL
group by PROJECT;
```

**Script 10.47** - Computing the total salary cost of each project

PROJECT	TCost
AGRO-2000	156916.5
BIOTECH	280328.7
SURVEYOR	243445.1

#### 10.11.4 Multidimensional statistics

The statistics discussed so far have considered one dimension of employee entities, namely PROJECT or CITY, that appeared in source table VALUED\_STABLE\_INTERVAL. The structure of this table can be formally defined by this pattern<sup>9</sup>:

PROJECT, interval  $\rightarrow \rightarrow$  SALARY

which tells that a *project* and an *interval* together determine a *set of salaries*, on which various aggregate functions can be applied. The presentation below of the data of Figure 10.26 makes this relationship explicit:

PROJECT	SALARY	Start	End
AGRO-2000	3400, 3900	2016-08-31	2016-09-16
AGRO-2000	3400, 3900, 4000	2016-09-16	2016-12-11
AGRO-2000	4100	2016-12-11	2016-12-29
AGRO-2000	3700, 3800, 4100	2016-12-29	2017-05-19
AGRO-2000	3700, 3800, 3900, 4100	2017-05-19	2017-07-24
AGRO-2000	3700, 3800, 3900	2017-07-24	2017-08-09
AGRO-2000	3800, 3900	2017-08-09	9999-12-31
...	...	...	...

This idea can be generalized to grouping criteria comprising more than one column. For example, let us examine this one:

PROJECT, CITY, interval  $\rightarrow \rightarrow$  SALARY

through which we associate a set of salaries with each couple (PROJECT, CITY), partitioned by stable intervals. Tables EVENT, STABLE\_INTERVAL and VALUED\_STABLE\_INTERVAL are computed accordingly (Script 10.48).

```
create table EVENT as
select PROJECT, CITY, Instant
from (select PROJECT, CITY, start as Instant from H_EMPLOYEE
union
select PROJECT, CITY, end as Instant from H_EMPLOYEE);
```

9. Yes, it's an *embedded multivalued dependency*, but don't tell anyone!

```

create table STABLE_INTERVAL as
select PROJECT,CITY,
       Instant as Start,
       (select min(Instant)
        from EVENT
        where (PROJECT,CITY) = (V.PROJECT,V.CITY)
        and Instant > V.Instant) as End
from EVENT V
where End is not null;

create table VALUED_STABLE_INTERVAL as
select S.PROJECT,S.CITY,E.CODE,E.SALARY,S.Start,S.End
from STABLE_INTERVAL S,H_EMPLOYEE E
where (S.PROJECT,S.CITY) = (E.PROJECT,E.CITY)
and S.Start >= E.start and S.End <= E.end;

```

**Script 10.48** - Computing interval tables for grouping criterion (PROJECT,CITY)

The contents of VALUED\_STABLE\_INTERVAL now looks like this (the codes of the employees have been added for clarity<sup>10</sup>):

PROJECT	CITY	CODE	SALARY	Start	End
AGRO-2000	Genève	G96	3900	2016-08-31	2016-12-11
AGRO-2000	Genève	G96	4100	2016-12-11	2016-12-29
AGRO-2000	Genève	G96	4100	2016-12-29	2017-05-19
AGRO-2000	Genève	N240	3700	2016-12-29	2017-05-19
AGRO-2000	Genève	G96	3900	2017-05-19	2017-08-09
AGRO-2000	Genève	N240	3700	2017-05-19	2017-08-09
AGRO-2000	Genève	G96	3900	2017-08-09	9999-12-31
AGRO-2000	Grenoble	A237	4000	2016-09-16	2016-12-11
AGRO-2000	Grenoble	A237	3800	2016-12-29	9999-12-31
AGRO-2000	Lille	C45	3400	2016-08-31	2016-12-11
AGRO-2000	Toulouse	D122	4100	2017-05-19	2017-07-24
BIOTECH	...	...	...	...	...

**Figure 10.29** - Contents of table VALUED\_STABLE\_INTERVAL following pattern PROJECT,CITY,interval →→ CODE,SALARY

We note that the sequence of stable intervals of cities within AGRO-2000 are independent. For instance, interval [2016-09-16,2016-12-11) of (AGRO-2000,Grenoble) is meaningless for (AGRO-2000,Genève). Each row describes the contribution of one employee to one stable interval of a city in a project. (AGRO-2000,Genève) comprises 5 stable intervals with two contributors, G96 and N240. G96 contributes to 5 intervals (5 rows) while N240 appears in 2 intervals (2 rows) for a total of 7 contributions (7 rows).

10. So that the pattern actually is PROJECT,CITY,interval →→ CODE,SALARY

The contents of this table, limited to project AGRO-2000, is represented graphically in Figure 10.30, in which city names have been abbreviated.

	2016-08-31	2016-12-11	2017-05-19	2017-08-09	now
	2016-09-16	016-12-29	017-07-24		
Gen (G96) :	<-3900--	--3900->	<-4100->	<-3900--	--3900->
Gen (C240) :			<-3700->	<-3700--	--3700->
Gre (A237) :		<-4000->	<-3800--	--3800--	--3800->
Lil (C45) :	<-3400--	--3400->			
Tou (D122) :			<-4100->		

**Figure 10.30** - Distribution among stable intervals of the H\_EMPLOYEE states of the employees who contributed to project AGRO-2000, grouped by the city of the contribution

Now, we are ready to query table VALUED\_STABLE\_INTERVAL. We will examine how to count, compute average and sums, as we did for unidimensional statistics.

### a) Counting employees

Script 10.49 computes the evolution of the number of employees on each project from each city. It is a natural extension of Script 10.41.

```
select PROJECT, CITY, count (*) as Number, Start, End
from   VALUED_STABLE_INTERVAL
group by PROJECT, CITY, Start, End
order by PROJECT, CITY, Start, end;
```

**Script 10.49** - computing the evolution of the number of employees per project and city (reduction not shown)

PROJECT	CITY	Number	Start	End
AGRO-2000	Genève	1	2016-08-31	2016-12-29
AGRO-2000	Genève	2	2016-12-29	2017-08-09
AGRO-2000	Genève	1	2017-08-09	2017-12-31
AGRO-2000	Grenoble	1	2016-09-16	2016-12-11
AGRO-2000	Grenoble	1	2016-12-29	2017-12-31
AGRO-2000	Lille	1	2016-08-31	2016-12-11
AGRO-2000	Toulouse	1	2017-05-19	2017-07-24
BIOTECH	...	...	...	...
...	...	...	...	...

### b) Computing average salaries

In Script 10.50, a natural extension of the query of Script 10.45, we compute the evolution of the average salary of the employees of each project from each city.

```

select  PROJECT,CITY,
        group_concat(distinct SALARY) as Salaries,
        round(avg(SALARY),1) as Average,Start,End
from    VALUED_STABLE_INTERVAL
group by PROJECT,CITY,Start,End
order by PROJECT,CITY,Start;

```

**Script 10.50** - Computing the average salary in each stable interval of 'AGRO-2000'

PROJECT	CITY	Salaries	Average	Start	End
AGRO-2000	Genève	3900	3900.0	2016-08-31	2016-12-11
AGRO-2000	Genève	4100	4100.0	2016-12-11	2016-12-29
AGRO-2000	Genève	4100,3700	3900.0	2016-12-29	2017-05-19
AGRO-2000	Genève	3900,3700	3800.0	2017-05-19	2017-08-09
AGRO-2000	Genève	3900	3900.0	2017-08-09	9999-12-31
AGRO-2000	Grenoble	4000	4000.0	2016-09-16	2016-12-11
AGRO-2000	Grenoble	3800	3800.0	2016-12-29	9999-12-31
AGRO-2000	Lille	3400	3400.0	2016-08-31	2016-12-11
AGRO-2000	Toulouse	4100	4100.0	2017-05-19	2017-07-24
BIOTECH	...	...	...	...	...
...	...	...	...	...	...

### c) Computing total salary costs

Same exercise for the sum statistics. Query 10.51 is an extension of Script 10.46.

```

select  PROJECT,CITY
        sum(SALARY/30.42 *
            durationDays(Start,min(End,'$maxDate$')))) as TCost,
        Start,End
from    VALUED_STABLE_INTERVAL
group by PROJECT,CITY,Start,End;

```

**Script 10.51** - Computing the evolution of the total salary cost of each stable interval

PROJECT	CITY	TCost	Start	End
AGRO-2000	Genève	13042.6	2016-08-31	2016-12-11
AGRO-2000	Genève	2419.7	2016-12-11	2016-12-29
AGRO-2000	Genève	36059.0	2016-12-29	2017-05-19
AGRO-2000	Genève	20432.7	2017-05-19	2017-08-09
AGRO-2000	Genève	74803.3	2017-08-09	9999-12-31
AGRO-2000	Grenoble	11278.7	2016-09-16	2016-12-11
AGRO-2000	Grenoble	100668.9	2016-12-29	9999-12-31
AGRO-2000	Lille	11370.5	2016-08-31	2016-12-11
AGRO-2000	Toulouse	8872.1	2017-05-19	2017-07-24
BIOTECH	...	...	...	...
...	...	...	...	...

#### d) Statistics on a subset of the aggregation criterion

Statistics on *non temporal data* can also be analyzed through the pattern we used to describe aggregation criteria and values to aggregate. We can, for instance, derive a synthetic table (let us call it PC\_S\_EMPLOYEE) based on this formula, in which the temporal dimension is discarded:

PROJECT,CITY  $\rightarrow\rightarrow$  SALARY

Such a table can be extracted through this query:

```
create table PC_S_EMPLOYEE as
select PROJECT,CITY,SALARY
from EMPLOYEE;
```

The nice feature of this table is that it provides the support of several aggregation criteria: (PROJECT,CITY) of course, but also its subsets (PROJECT) and (CITY). The following query is perfectly valid:

```
select PROJECT,
       count(*) as Number,
       avg(SALARY) as AvgSalary
from PC_S_EMPLOYEE
group by PROJECT;
```

#### Unfortunately, this does not work for temporal data!

Temporal table VALUED\_STABLE\_INTERVAL built by the queries of Script 10.48 **does not allow** data grouping by PROJECT or CITY alone. Indeed, as the life lines of Figure 10.30 shows it, the events of, say, (AGRO-2000, Genève) are independent of (and therefore different from) those of, say, (AGRO-2000, Grenoble). As a consequence, grouping data by PROJECT or CITY alone would be meaningless. The following query **will not** provide the evolution of the number of employees in each project:

```
select PROJECT,count(*) as Number,Start,End
from VALUED_STABLE_INTERVAL
group by PROJECT,Start,End;
```

We will address this restriction in the next section.

### 10.11.5 Temporal data with non-partitioned intervals

The history generated by the procedures developed so far are based on a partition of the source rows according to one or more columns. This partition is defined by expressions such as this one:

PROJECT,CITY,interval  $\rightarrow\rightarrow$  SALARY



Each subset of the partition is identified by a couple (p,c) such that H\_EMPLOYEE includes at least one row where PROJECT = p and CITY = c. This subset divides its time line into successive stable intervals. As we have observed, the divisions of two subsets do not match, so that they cannot be compared. This is why these stable intervals cannot be used to derive more coarse statistics such as those defined by PROJECT,interval  $\rightarrow\rightarrow$  SALARY or CITY,interval  $\rightarrow\rightarrow$  SALARY.

In this section, we will show how to compute stable intervals **common** to all the couples (p,c), that is, a division of the time line synchronous with that of all these couples. For this, we first create the list of all the events of H\_EMPLOYEE, independently of any partitioning criterion. We adapt the query of Script 10.38 by removing the reference to PROJECT (Script 10.52).

```
create temp table EVENT as
select Instant
from (select start as Instant from H_EMPLOYEE
union
select end as Instant from H_EMPLOYEE);
```

**Script 10.52** - Extracting the time points of table H\_EMPLOYEE

From this table, we build the stable intervals with the algorithm of Script 10.53 (derived from that of Script 10.39).

```
create temp table STABLE_INTERVAL as
select Instant as Start,
(select min(Instant)
from EVENT
where Instant > E.Instant) as End
from EVENT E
where End is not null;
```

**Script 10.53** - Building the stable intervals from table EVENT

Finally, we complete these intervals with all the values from H\_EMPLOYEE that will be used to compute the desired statistics (Script 10.54).

```
create temp table VALUED_STABLE_INTERVAL as
select E.CODE, E.SALARY, E.CITY, E.PROJECT, S.Start, S.End
from STABLE_INTERVAL S, H_EMPLOYEE E
where E.Start <= S.Start and S.Start < E.End;
```

**Script 10.54** - Completing stable intervals with additional data from H\_EMPLOYEE

The new state of table VALUED\_STABLE\_INTERVAL is shown in Figure 10.31 (sorted by PROJECT,CITY,Start). Each row describes the contribution of one employee to one interval. An employee may contribute to several intervals (poten-

tially with the same values of PROJECT, CITY and SALARY) and an interval may be contributed by several employees.

PROJECT	CITY	CODE	SALARY	Start	End
AGRO-2000	Genève	G96	3900	2016-08-31	2016-09-16
AGRO-2000	Genève	G96	3900	2016-09-16	2016-10-22
AGRO-2000	Genève	G96	3900	2016-10-22	2016-12-08
AGRO-2000	Genève	G96	3900	2016-12-08	2016-12-11
AGRO-2000	Genève	G96	4100	2016-12-11	2016-12-29
AGRO-2000	Genève	G96	4100	2016-12-29	2017-01-14
AGRO-2000	Genève	N240	3700	2016-12-29	2017-01-14
AGRO-2000	Genève	G96	4100	2017-01-14	2017-04-03
AGRO-2000	Genève	N240	3700	2017-01-14	2017-04-03
AGRO-2000	Genève	G96	4100	2017-04-03	2017-05-19
AGRO-2000	Genève	N240	3700	2017-04-03	2017-05-19
AGRO-2000	Genève	N240	3700	2017-05-19	2017-07-24
AGRO-2000	Genève	G96	3900	2017-05-19	2017-07-24
AGRO-2000	Genève	N240	3700	2017-07-24	2017-08-09
AGRO-2000	Genève	G96	3900	2017-07-24	2017-08-09
AGRO-2000	Genève	G96	3900	2017-08-09	9999-12-31
AGRO-2000	Grenoble	A237	4000	2016-09-16	2016-10-22
...	...	...	...	...	...
BIOTECH	Genève	G96	3400	2014-11-23	2014-12-04
BIOTECH	Genève	G96	3400	2014-12-04	2015-01-17
...	...	...	...	...	...
BIOTECH	Toulouse	N240	3100	2016-08-18	2016-08-31
BIOTECH	Toulouse	N240	3100	2016-08-31	2016-09-16
SURVEYOR	Genève	D107	4100	2016-03-22	2016-08-18
SURVEYOR	Genève	D107	4100	2016-08-18	2016-08-31
SURVEYOR	Genève	D107	4100	2016-08-31	2016-09-16
SURVEYOR	Genève	D107	4100	2016-09-16	2016-10-22
SURVEYOR	Genève	D107	4100	2016-10-22	2016-12-08
SURVEYOR	Grenoble	D107	3800	2016-02-16	2016-03-21
SURVEYOR	Grenoble	D107	3800	2016-03-21	2016-03-22
SURVEYOR	Grenoble	N240	3700	2016-09-16	2016-10-22
...	...	...	...	...	...
SURVEYOR	Toulouse	A68	3700	2016-09-16	2016-10-22

Figure 10.31 - Non-partitioned version of VALUED\_STABLE\_INTERVAL

This single version of VALUED\_STABLE\_INTERVAL will allow us to compute all the statistics we have described in the preceding sections.

### a) Counting employees

To compute the evolution of the number of employees from each city working on each project, Script 10.55 first counts them<sup>11</sup> in each stable interval, then reduces

11. Since the *select list* of the query includes column *End*, the grouping criterion should be written "group by PROJECT, CITY, Start, End" according to the SQL standard. We have omitted the *End* component, which is (illegally) allowed by SQLite (and by MySQL, among others) for performance reason (the execution is 10 to 15% faster).

value-equivalent intervals. Note that expression `count(distinct CODE)` can be more simply written `count(*)`.

```
create temp table H_COUNT as
  select PROJECT,CITY,count(distinct CODE) as Number,Start,End
  from   VALUED_STABLE_INTERVAL
  group by PROJECT,CITY,Start;
function LTemp:reduce {select PROJECT,CITY,Number from H_COUNT};
```

**Script 10.55** - Computing the evolution of the number of employees in *each project* and *each city*

```
create temp table H_COUNT as
  select PROJECT,count(distinct CODE) as Number,Start,End
  from   VALUED_STABLE_INTERVAL
  group by PROJECT,Start;
function LTemp:reduce {select PROJECT,Number from H_COUNT};
```

**Script 10.56** - Computing the evolution of the number of employees in *each project*

```
create temp table H_COUNT as
  select CITY,count(distinct CODE) as Number,Start,End
  from   VALUED_STABLE_INTERVAL
  group by CITY,Start;
function LTemp:reduce {select CITY,Number from H_COUNT};
```

**Script 10.57** - Computing the evolution of the number of employees in *each city*

CITY	Number	Start	End
Genève	1	2014-11-23	2016-03-22
Genève	2	2016-03-22	2016-12-08
Genève	1	2016-12-08	2016-12-29
Genève	2	2016-12-29	2017-08-09
Genève	1	2017-08-09	9999-12-31
Grenoble	1	2016-02-16	2016-03-22
Grenoble	2	2016-09-16	2016-12-08
Grenoble	3	2016-12-08	2016-12-29
Grenoble	2	2016-12-29	9999-12-31
Lille	1	2015-08-22	9999-12-31
Paris	1	2014-12-04	2015-06-15
Paris	2	2015-06-15	2016-10-22
Paris	3	2016-10-22	2017-05-19
Paris	1	2017-05-19	9999-12-31
Toulouse	1	2015-06-09	2016-08-18

Printed 29/8/19

Toulouse	2	2016-08-18	2016-09-16
Toulouse	1	2016-09-16	2016-10-22
Toulouse	1	2017-05-19	2017-07-24

**Figure 10.32** - Result of Script 10.57: evolution of the number of employees in *each city*

### b) Computing average, min and max salaries

The evolution of *average*, *min* and *max* statistics of SALARY values is also derived from the non-partitioned version of VALUED\_STABLE\_INTERVAL (Scripts 10.58).

```
create temp table H_AVG_SALARY as
select PROJECT, CITY,
       round(avg(SALARY), 1) as Average,
       min(SALARY) as Min,
       max(SALARY) as Max, Start, End
from   VALUED_STABLE_INTERVAL
group by PROJECT, CITY, Start;
function LTemp:reduce {select PROJECT, CITY, Average, Min, Max
                        from   H_AVG_SALARY};
```

**Script 10.58** - Computing the evolution of the main statistics of SALARY

Producing these statistics per project or per city is quite similar.

### c) Computing salary costs

Table H\_COST sums and records, for each stable interval, the cost of all the employees who were active in this interval. It is created by the query of Script 10.59.

```
create temp table H_COST as
select PROJECT, CITY,
       sum(SALARY)/30.42 *
       durationDays(Start, min(End, '$maxDate$')) as TCost,
       Start, End
from   VALUED_STABLE_INTERVAL
group by PROJECT, CITY, Start;
```

**Script 10.59** - Computing the salary cost of each stable interval

The evolution of salary cost just requires an appropriate presentation of H\_COST (Script 10.60). In Script 10.61, the values of TCost are summed for each interval of each project.

```
select PROJECT,CITY,round(TCost,1) as "TCost",Start,End
from   H_COST
order by PROJECT,CITY,Start;
```

**Script 10.60** - Evolution of the salary cost of project and city

```
select PROJECT,round(sum(TCost),1) as "TCost",
       min(Start) as "Start",max(End) as "End"
from   H_COST
group by PROJECT,Start;
```

**Script 10.61** - Evolution of the salary cost of each project

By summing the values of TCost for each project and/or each city, we get their total salary cost (Scripts 10.62 and 10.63). The table of the total cost of each project is shown in Figure 10.33.

```
select PROJECT,CITY,int(sum(TCost)) as TotalCost
from   H_COST
group by PROJECT,CITY;
```

**Script 10.62** - Computing the total salary cost of each project in each city

```
select PROJECT,int(sum(TCost)) as TotalCost
from   H_COST
group by PROJECT;
```

**Script 10.63** - Computing the total salary cost of each project

PROJECT	TotalCost
AGRO-2000	156,917
BIOTECH	280,329
SURVEYOR	243,445

**Figure 10.33** - Total cost of each project (Script 10.63)

### 10.11.6 Efficient computing of VALUED\_STABLE\_INTERVAL

The three operations that produce the basic VALUED\_STABLE\_INTERVAL table (through Script 10.48) from which we have computed our statistics can prove too

slow for large temporal table. They can be replaced by function **valuedInterval** of the **LTemp** library, which is both simpler and much faster. The sequence of Script 10.48 can be replaced by Script 10.64, which produces the data of Figure 10.29.

```
function LTemp:valuedInterval
{select PROJECT,CITY,CODE,SALARY from H_EMPLOYEE
group by CITY,PROJECT
into VALUED_STABLE_INTERVAL};
```

**Script 10.64** - Computing VALUED\_STABLE\_INTERVAL through function LTemp:valuedInterval

The gain in execution time results mainly from the way the stable intervals are computed from the EVENT table, through a single scan of this table instead of executing a self-join particularly expensive.

To produce non-partitionned intervals (Scripts 10.52, 10.53, 10.54), we simply omit the group by clause (Script 10.65).

```
function LTemp:valuedInterval
{select PROJECT,CITY,CODE,SALARY from H_EMPLOYEE
into VALUED_STABLE_INTERVAL};
```

**Script 10.65** - Computing VALUED\_STABLE\_INTERVAL for common intervals through function LTemp:valuedInterval

### 10.11.7 Temporal data with fixed and unit intervals

In the techniques discussed so far, the stable intervals are naturally derived from the data themselves. Though the concept of stable interval is intuitive, it does not scale easily (Section 10.11.4, d) and sometimes leads to non trivial queries. In addition, it provides synthetic data that are difficult to plot on a timeline due to the uneven distribution of the time points. Hence the idea to extract statistics based on *fixed-length intervals*: seconds, days, months, years, etc.

The algorithm of Script 10.66 generates a table in which each row is assigned to a day within interval [minDate,maxDate). The day is represented by unit interval [Start,End), the length of which is the granularity of the time measure (Figure 10.34).

To generate other fixed-length interval, we replace expression `nextDate(D)` by `addToDate(D,n)`, where `n` is the period. So, `addToDate(D,7)` will provide weekly intervals.

```

create temp table CALENDAR(Start date,End date);
with recursive Dates(Start,End)
as (select '$minDate$',nextDate('$minDate$')
    union all
    select D.End, nextDate(D.End) as Last
    from   Dates D
    where  Last <= '$maxDate$'
)
insert into CALENDAR select * from Dates;

```

**Script 10.66** - Building a daily calendar

This calendar is the ultimate version of STABLE\_INTERVAL. Joined with table H\_EMPLOYEE (Script 10.67), it provides a version of VALUED\_STABLE\_INTERVAL in which each row describes one day in the life of one employee (Figure 10.35).<sup>12</sup>

Start	End
2014-11-23	2014-11-24
2014-11-24	2014-11-25
2014-11-25	2014-11-26
2014-11-26	2014-11-27
2014-11-27	2014-11-28
2014-11-28	2014-11-29
2014-11-29	2014-11-30
2014-11-30	2014-12-01
2014-12-01	2014-12-02
...	...

**Figure 10.34** - A unit (daily) calendar

From now on, we can ignore temporal concerns in aggregation functions and apply them as we usually do in non-temporal processing. Moreover, the queries will be more regular. The price to pay is that this table may be quite large. In our case study, it comprises 6,110 rows instead of 28 in H\_EMPLOYEE and 130 in non-partitioned VALUED\_STABLE\_INTERVAL.

```

create temp table VALUED_STABLE_INTERVAL as
select E.CODE,E.NAME,E.PROJECT,E.CITY,E.SALARY,C.Start,C.End
from   CALENDAR C,H_EMPLOYEE E
where  C.Start >= E.start
and    C.End    <= E.end;

```

**Script 10.67** - Daily distribution of table H\_EMPLOYEE

12. Converting a temporal table according to unit intervals is often called **unfolding**.

PROJECT	CODE	NAME	CITY	SALARY	Start	End
AGRO-2000	G96	Godin	Genève	3900	2016-08-31	2016-09-01
AGRO-2000	G96	Godin	Genève	3900	2016-09-01	2016-09-02
AGRO-2000	G96	Godin	Genève	3900	2016-09-02	2016-09-03
AGRO-2000	G96	Godin	Genève	3900	2016-09-03	2016-09-04
AGRO-2000	G96	Godin	Genève	3900	2016-09-04	2016-09-05
AGRO-2000	G96	Godin	Genève	3900	2016-09-05	2016-09-06
...	...	...	.....	...	...	...
AGRO-2000	G96	Godin	Genève	4100	2017-03-15	2017-03-16
AGRO-2000	G96	Godin	Genève	4100	2017-03-16	2017-03-17
AGRO-2000	G96	Godin	Genève	4100	2017-03-17	2017-03-18
AGRO-2000	G96	Godin	Genève	4100	2017-03-18	2017-03-19
BIOTECH	...	...	.....	...	...	...
...	...	...	.....	...	...	...

Figure 10.35 - Daily history of employees (table VALUED\_STABLE\_INTERVAL)

Script 10.68 shows that source table H\_EMPLOYEE can be rebuilt from VALUED\_STABLE\_INTERVAL. So both tables contain the same information.

```
function LTemp:project
{select CODE,NAME,SALARY,CITY,PROJECT
 from   VALUED_STABLE_INTERVAL into H_EMPLOYEE};
```

Script 10.68 - Reconstructing H\_EMPLOYEE from VALUED\_STABLE\_INTERVAL

This version of VALUED\_STABLE\_INTERVAL can now be used to compute all the statistics of the preceding section.

Computing statistics against larger fixed-length intervals, such as weeks or months may be more complex because these intervals generally do not match those of source data, and therefore are not stable. Indeed, several stable intervals may contribute, often partially, to one fixed-length interval. Starting from *unit valued stable intervals* makes it easier to aggregate data on larger fixed-length intervals. Implementing this idea is left as an (easy) exercise to the reader.

A last remark: aggregation based on unit intervals has been called in the literature *Instantaneous temporal aggregation*.

## 10.12 Temporal library LTemp

We have shown the management of all the temporal data models studied in the first part of this case study and the exploitation of temporal data can be performed through pure SQL queries. However, this may raise two potential problems:



- Some of these SQL queries are fairly complex, as testified by the generalized projection operator (see Script 10.22). In addition, some operations require a sequence of several queries, which does not contribute to the readability of temporal data processing scripts.
- Due to the declarative nature of SQL, the execution of some queries may prove rather slow, particularly for large data sets, depending on the optimization strategies applied by the DBMS.<sup>13</sup> In some cases, the result of these queries can be obtained faster by a simple scan of the temporal table.

The SQLfast library **LTemp** provides a collection of operators the goal of which is to simplify and optimize the processing of temporal data. This library is a Python module that works on the currently opened database. Each function requires a single argument which is a query expressed in a pseudo-SQL language. It returns a numeric code indicating how the execution was carried out.

The table and column names that appear in the argument of the functions must obey the following syntactic rules: strict SQL naming conventions, no SQL reserved words<sup>14</sup>, no quoted or bracketed names and no name beginning with an underscore character.

In all the temporal tables referenced in the function argument, the interval columns (*start*, followed by *end*, whatever their real name) must be the last ones of their schema.

At the present time, this library offers five operators, but more may be developed in the future.

### a) Temporal projection

This operator executes a generalized projection of a source table on selected columns and stores the result in a target table.

#### Syntax

```
LTemp:project {<project-query>;}
```

#### Argument

```
<project-query> ::= select <columns>
                    from   <source>
                    [where  <condition>]
                    into   <target>

<source>      ::= table name
<columns>    ::= column list
<condition>  ::= SQL condition
```

13. For instance, the restricted version of the projection of Script 10.20 (with one level of subquery) is particularly fast, while the generalized version of Script 10.22 (with two levels of subquery) is much slower. This observation has been done for SQLite v3.28. The conclusion can be different for other versions or other DBMS.

14. Except for *start* and *end* for interval columns, courtesy of the benevolence of SQLite!

`<target> ::= table name`

### Semantics

Computes the generalized projection of table (or view) `<source>` on its columns `<columns>` and stores the result in table `<target>`. If `<condition>` is specified, only the rows that satisfy this condition are processed. If table `<target>` does not exist it is created.

### Constraint

- The database containing table `<source>` must be opened.
- The columns of `<columns>` may not specify column aliases.
- The from clause may not specify a table alias.

### Return code

- 0: successful execution
- 1: no database opened
- 2: syntax error in `<project-query>`
- 3: error in source data extraction
- 4: error in generating the result data

### Example

```
function status = LTemp:project {select CITY,PROJECT
                                from   H_EMPLOYEE
                                where  Salary > 6000
                                into  H_CITY_PRO};
```

## b) Temporal reduction

This operator implements a simplified version of the projection. It reduces the consecutive value-equivalent states in the source table itself.

### Syntax

`LTemp:reduce {<reduce-query>;}`

### Argument

```
<reduce-query> ::= select <columns>
                  from   <source>
<source>      ::= table name
<columns>    ::= column list
```

### Semantics

Reduces the consecutive value-equivalent states in the source table itself. Same effect as `LTemp:project` where the target table is the source table.

**Constraint**

- The database containing table <source> must be opened.
- <source> must be a base table, not a view.
- The columns of <columns> may not specify column aliases.
- The from clause may not specify a table alias.

**Return code**

- 0: successful execution
- 1: no database opened
- 2: syntax error in <project-query>
- 3: error in source data extraction
- 4: error in generating the result data

**Example**

```
function status = LTemp:reduce {select CITY,PROJECT from H_CP};
```

**c) Temporal intervals (basic)**

This generates the stable intervals of the source table, augmented by additional column values and stores the result in a target table. This operator prepares data for computing aggregates on the additional columns.

**Syntax**

```
LTemp:valuedInterval {<interval-query>;
```

**Argument**

```
<interval-query> ::= select <columns>
                    from   <source>
                    [where <condition>]
                    [group by <groupBy>]
                    into   <target>

<source>      ::= table name
<columns>    ::= column list
<condition>  ::= SQL condition
<groupBy>    ::= column list
<target>     ::= table name
```

**Example**

```
function status = LTemp:valuedInterval
{select PROJECT,CITY,SALARY
 from   H_EMPLOYEE
 where  SALARY > 6000
 group by PROJECT,CITY
```

```

        into VALUED_STABLE_INTERVAL};

function status = LTemp:valuedInterval
{select PROJECT,CITY,SALARY
 from   H_EMPLOYEE
 into   COMMON_VALUED};

```

### Semantics

This operator is used as the first step of temporal aggregate functions. It computes the largest intervals in source table *<source>* (among the rows that satisfy *<condition>*) in which the values of *<groupBy>* are constant. Then, it associates with each of these intervals the values of *<columns>* that hold in the source table in this interval. From this result, that has been stored in table *<target>*, various temporal aggregate functions can be applied on *<columns>* for each *<groupBy>* group. Table *<target>* is created if it does not exist.

When *<groupBy>* is absent, the intervals are computed differently. An interval is considered stable throughout table *<source>* if the values of *<columns>* (not only of *<groupBy>* as in the former formula) are constant in this interval.

### Constraint

- The database containing table *<source>* must be opened.
- The columns of *<columns>* may not specify column aliases.
- The columns of *<groupBy>* form a subset of those of *<columns>*.
- The from clause may not specify a table alias.

### Return code

- 0: successful execution
- 1: no database opened
- 2: syntax error in *<project-query>*
- 3: error in source data extraction
- 4: error in generating the result data

### Example

```

function status = LTemp:valuedInterval
{select PROJECT,CITY,SALARY
 from   H_EMPLOYEE
 where  SALARY > 6000
 group by PROJECT,CITY
 into   VALUED_STABLE_INTERVAL};

function status = LTemp:valuedInterval
{select PROJECT,CITY,SALARY
 from   H_EMPLOYEE
 into   COMMON_VALUED};

```

### d) Temporal intervals (extended)

Same function as `valuedInterval` with more flexible *select list*. An element of the *select list* can be a column name but also a constant or a scalar SQL expression.

#### Syntax

```
LTemp:valuedIntervalE {<interval-query>;}
```

#### Argument

```
<interval-query> ::= select <columns>
                    from   <source> <alias>
                    [where <condition>]
                    [group by <groupBy>]
                    into   <target>

<source>          ::= table name
<exp-list>        ::= list of SQL scalar expressions
                    where column names are prefixed
                    with <alias> for <source> columns
                    and 'S' for Start and End
                    from "_STABLE_INTERVAL"

<condition>       ::= SQL condition where column names are prefixed
                    with <alias>

<groupBy>         ::= column list where column names are prefixed
                    with <alias>

<target>          ::= table name
```

#### Semantics

This operator is used as the first step of temporal aggregate functions. It computes the largest intervals in source table `<source>` (among the rows that satisfy `<condition>`) in which the values of `<groupBy>` are constant. Then, it associates with each of these interval the values of `<exp-list>` that hold in the source table in this interval. Elements of the select list can be a column name, a constant or any expression that evaluates into a single value.

From this result, that has been stored in `<target>`, various temporal aggregate functions can be applied on `<exp-list>` for each `<groupBy>` group.

When `<groupBy>` is absent, the intervals are computed differently. An interval is considered stable throughout table `<source>` if the values of `<exp-list>` are constant in this interval.

#### Constraint

- The database containing table `<source>` must be opened.
- The column names that appear in `<exp-list>` must be prefixed with `<alias>` except columns `Start` and `End`, defining the interval, that must be prefixed by alias `'S'`.
- The columns of `<groupBy>` form a subset of those of `<columns>`.

**Return code**

- 0: successful execution
- 1: no database opened
- 2: syntax error in <project-query>
- 3: error in source data extraction
- 4: error in generating the result data

**Example**

```
function status = LTemp:valuedIntervalsE
{select E.PROJECT,E.CITY,E.SALARY*12/365,
      durationDays(S.Start,S.End) as Dur
 from   H_EMPLOYEE E
 where  E.SALARY > 6000
 group by E.PROJECT, E.CITY
 into VALUED_STABLE_INTERVAL};
```

**e) Temporal normalization**

This operator completes a source table by inserting fictitious states made up of *null* values for each missing interval. It is particularly useful to prepare tables for outer joins.

**Syntax**

```
LTemp:normalize {<normalize-query>;
```

**Argument**

```
<normalize-query> ::= update <source>
                    [wrt <compare>]
                    on  <Scolumns> [= <Ccolumns>]
                    [into <target>]

<source> ::= table name
<compare> ::= table name
<Scolumns> ::= column list
<Ccolumns> ::= column list
<target> ::= table name
```

**Semantics**

If, for each value of <Scolumns>, table <source> is incomplete and includes gaps, aka *missing intervals*, this operator replaces them by fictitious states comprising the value of <Scolumns> + *null* values. So, the history of each value of <Scolumns> is now complete. If <target> is specified, the completed data are stored in table <target>. Otherwise, the new states are inserted in table <source> itself. If table <target> does not exist, it is created.

If table <compare> is specified, <Ccolumns> must also be specified. When comparing the history of a value of <Scolumns> in table <source> with the history of the same value of <Ccolumns> in <compare>, if the latter history

either starts earlier or ends later, then one or two fictitious states are added to the history of `<source>` in order to adjust both histories. If this way, the history of `<source>` does not start after that of `<compare>` and does not end before that of `<compare>`, and this, for each value of `<Scolumns>`. This function is mainly used in the preparation of the arguments of outer joins.

If table `<source>` is empty, a *null* state is generated for each value of `<Ccolumns>`. Its interval is the life span of this value in `<compare>`.

#### Constraints

- The database containing table `<source>` must be opened.
- If `<target>` is specified, `<source>` can be a view.
- The columns of `<columns>` may not specify column aliases.
- The from clause may not specify a table alias.

#### Return code

- 0: successful execution
- 1: no database opened
- 2: syntax error in `<project-query>`
- 3: error in source data extraction
- 4: error in generating the result data

#### Examples

```
function status = LTemp:normalize {update H_EMP on Proj;

function status = LTemp:normalize {update H_EMP on Proj
                                   into NewEMP;

function status = LTemp:normalize {update H_EMP wrt H_PRO
                                   on Proj = ProjID
                                   into NewEMP;

function status = LTemp:normalize {update H_T2 wrt H_T1
                                   on A1,A2 = B1,B2
                                   into NewEMP;
```

## 10.13 Temporal data format conversion

The first part of this case study describes some of the most popular models of temporal data. The first one, *entity-based*, or more generally *tuple-based*, have been our main support to develop the algorithms of this second part. In this model, all the attributes of an entity are collected to form a single row (or tuple), with which we associate a time interval that specifies the period during which these values were valid, either in the real world (valid time) or in the database (transaction time).

The other formats are briefly described and illustrated in this first part, but we have not developed algorithms to convert temporal data from one model into another one. The reasons are twofold: some conversion rules are obvious (*horizontal splitting* for example) while others require operators that were not yet introduced at that stage.

In this section, we examine the conversion rules of entity-based temporal data to attribute-based, event-based and document-oriented models.

### 10.13.1 Entity-based to attribute-based history conversion

Each attribute table is obtained through a temporal projection. Script 10.69 generates attribute tables H\_SAL, H\_CIT and H\_PRO from columns SALARY, CITY and PROJECT of table H\_EMPLOYEE. *null* states can be deleted if needed.

The source table can be recovered through a simple *temporal join* of the attribute tables if *null* states are preserved. Otherwise, H\_EMPLOYEE must be rebuilt through an *outer join*.

```
function LTemp:project
  {select CODE, SALARY from H_EMPLOYEE into H_SAL};
function LTemp:project
  {select CODE, CITY from H_EMPLOYEE into H_CIT};
function LTemp:project
  {select CODE, PROJECT from H_EMPLOYEE into H_PRO};
```

Script 10.69 - Projecting of H\_EMPLOYEE on each non-PK column

### 10.13.2 Entity-based to event-based history conversion

While entity(or tuple)-based history represents the successive *states* of each entity, *event-based history* records the *state transitions* induced by data modification operation. Each row comprises a value for each entity attribute plus the nature of the event and the instant this event occurred. Figure 10.36 reminds us what the transaction time history of projects looks like.

TITLE	THEME	BUDGET	Time	Event
AGRO-2000	Crop improvement	65000	21	create
AGRO-2000	Crop improvement	75000	36	update
AGRO-2000	Crop improvement	82000	41	update
BIOTECH	Biotechnology	180000	2	create
BIOTECH	Genetic engineering	160000	7	update
BIOTECH	Genetic engineering	120000	9	update
BIOTECH	Genetic engineering	140000	20	update
BIOTECH	Biotechnology	140000	44	update
SURVEYOR	Satellite monitoring	310000	12	create



	SURVEYOR		Satellite monitoring		375000		18		update	
	SURVEYOR		Satellite monitoring		345000		31		update	
	SURVEYOR		Satellite monitoring		345000		40		delete	
+-----+-----+-----+-----+-----+-----+										

**Figure 10.36** - Event-based history of projects

The conversion of entity-based to event-based history relies on the following rules, stated for a definite project:

- *create*: this event is identified by the *first state* of the project. Its **Time** value is that of **start** of the row of this state.
- *delete*: this event corresponds to the *last state* of the project, provided the value of **end** of this state is not *infinite future*. Its **Time** value is that of **end** of the row of this state.
- *update*: this event is defined by a state of the project that is not the first one. Its **Time** value is that of **start** of the row of this state.

We observe that the last state of a deleted entity generates two events: an *update* or *create* event and a *delete* event.

Script 10.70 translates these rules to create event-based table EH\_PROJECT. The creation of table EH\_EMPLOYEE is quite similar.

```
create table EH_PROJECT as
  select TITLE,THEME,BUDGET,start as Time,'create' as Event
  from   H_PROJECT P
  where not exists(select *
                  from   H_PROJECT
                  where TITLE = P.TITLE and end = P.start)

  union

  select TITLE,THEME,BUDGET,end as Time,'delete' as Event
  from   H_PROJECT P
  where not exists(select *
                  from   H_PROJECT
                  where TITLE = P.TITLE and start = P.end)
  and    end <> $future$

  union

  select TITLE,THEME,BUDGET,start as Time,'update' as Event
  from   H_PROJECT P
  where exists( select *
                from   H_PROJECT
                where TITLE = P.TITLE and end = P.start);
```

**Script 10.70** - PROJECT: from entity-based to event-based

Let us now examine how each event of EH\_PROJECT can be translated into entity-based state.

- *create*: we create a state of the project (the first one) with the values of TITLE, THEME and BUDGET of the event row. Its **start** value is that of **Time** of the row of this event. If this event is followed by a next event on this project, then the **Time** value of the latter is assigned to the **end** value of the state in construction. Otherwise, if it is the only event of the project, then the state is current and its **end** value is set to infinite future.
- *update*: we create a state of the project. The values of columns TITLE, THEME, BUDGET, **start** and **end** are assigned with the same rules as those of the *create* event.
- *delete*: this event was just an additional by-product of the analysis of entity states. Its information has already been taken into account when examining the other two event rows. We can ignore it.

The algorithm of Script 10.71 generates entity-based view VEH\_PROJECT from table EH\_PROJECT. It proceeds in two steps: generation of past states and generation of current states.

- A *past state* is generated by two successive events of the project, the first one being a *create* or *update* event. Two events are successive if there is no other event between them.
- A *current state* is generated by a create or update event row that is not followed by another event of the same project.

We observe that *delete* events rows have not been used in the conversion process.

```
create view VEH_PROJECT (TITLE,THEME,BUDGET,start,end) as
-- Generate past states --
select P1.TITLE,P1.THEME,P1.BUDGET,P1.Time,P2.Time
from   EH_PROJECT P1,EH_PROJECT P2
where  P1.TITLE = P2.TITLE and P1.Time < P2.Time
and    P1.oper in ('create','update')
and    not exists(select * from EH_PROJECT
                  where  TITLE = P1.TITLE
                  and    Time > P1.Time and Time < P2.Time)

union

-- Generate current states --
select P.TITLE,P.THEME,P.BUDGET,P.Time,$future$
from   EH_PROJECT P
where  P.oper in ('create','update')
and    not exists(select * from EH_PROJECT
                  where  TITLE = P.TITLE
                  and    Time > P.Time);
```

**Script 10.71** - PROJECT: from event-based to entity-based

### 10.13.3 Entity-based to document-oriented history conversion

We will describe a method that produces data according to a light format using SQLfast multilist data structures. This format is close to that described in Section 9.9.5 and illustrated by Figure 9.18 but easier to generate and to decode. Deriving from it the algorithm that generates pure JSON variant is immediate.

The Projects value of employee A237 is as follows (we work on the transaction time version for simplicity):

AGRO-2000:24-30,33-999999;BIOTECH:30-33

It is structured as a 4-level hierarchy:

**level 1:** the value of Projects is a list of *value-intervals* components, (here, AGRO-2000:24-30,33-999999 and BIOTECH:30-33), separated by symbols ' ; ';

**level 2:** a *value-interval* component is a list of two elements, separated by symbol ' : ' ; the first element is the *value* of a project title (e.g., AGRO-2000) and the second one provides the *intervals* associated with this value;

**level 3:** the *intervals* are structured as a list of intervals, separated by symbols ' , ' ;

**level 4:** an *interval* is a list of two time points, separated by symbol ' - ' .

We will derive this structure from the attribute-based model generated by projecting H\_EMPLOYEE on each column. Through Script 10.69, we get tables H\_SAL, H\_CIT and H\_PRO. For simplicity, we ignore constant column NAME.

Then, we compute, for each distinct value of each column for each employee, the list of its intervals (Script 10.72 and Figure 10.37 for column PROJECT).

```
create table HH_PRO as
select CODE,PROJECT,group_concat(start||'-'||end,',') as Projects
from H_PRO group by CODE,PROJECT;
```

**Script 10.72** - Grouping intervals for each PROJECT value of each employee entity

CODE	PROJECT	Projects
A237	AGRO-2000	24-30,33-999999
A237	BIOTECH	30-33
A68	BIOTECH	8-13
A68	SURVEYOR	13-39
C45	AGRO-2000	22-28
C45	BIOTECH	11-22,28-999999
D107	BIOTECH	35-999999
D107	SURVEYOR	15-35
D122	AGRO-2000	37-42
D122	BIOTECH	10-16
D122	SURVEYOR	16-37

Printed 29/8/19

G96	AGRO-2000	23-999999
G96	BIOTECH	3-23
M158	BIOTECH	4-14, 34-999999
M158	SURVEYOR	14-34
N240	AGRO-2000	32-43
N240	BIOTECH	19-25
N240	SURVEYOR	25-32

Figure 10.37 - Project history of each employee entity

Finally, we collect these values together with their associated intervals for each CODE value (Script 10.73 and Figure 10.38).

```
select CODE,group_concat(PROJECT||':'||Projects,';') as Projects
from   HH_PRO group by CODE;
```

Script 10.73 - Associating their value-intervals pairs with each CODE value

CODE	Projects
A237	AGRO-2000:24-30, 33-999999;BIOTECH:30-33
A68	BIOTECH:8-13;SURVEYOR:13-39]
C45	AGRO-2000:22-28;BIOTECH:11-22,28-999999
D107	BIOTECH:35-999999;SURVEYOR:15-35]
D122	AGRO-2000:37-42;BIOTECH:10-16;SURVEYOR:16-37
G96	AGRO-2000:23-999999;BIOTECH:3-23
M158	BIOTECH:4-14, 34-999999;SURVEYOR:14-34
N240	AGRO-2000:32-43;BIOTECH:19-25;SURVEYOR:25-32

Figure 10.38 - Document-oriented presentation of the projects of employee entities

The final document-based table is obtained by joining this table with those built in a similar way on SALARY and CITY. By integrating these small scripts into a single SQL query, we get the final algorithm of Script 10.74.

Rebuilding the source data from this presentation is of course quite possible. However, doing this with SQL queries only would be fairly complex. Considering the 4-level structure of the data, the most obvious approach would be a procedure comprising embedded loops, each decoding a level of the data structure. This is what Script 10.75 implements for column PROJECT.

- The main loop reads each row of DOC\_EMPLOYEE, extracting the values of CODE (in variable cod) and Projects (in variable pro) of one employee.
- The next inner loop extracts successive *value-intervals* component of Projects (in variable project).
- The most inner loop extract from this component each individual interval (in variable interval).
- This interval is finally split into start and end values.

```

create table DOC_EMPLOYEE(CODE,Salaries,Cities,Projects);
with
  HH_SAL(CODE,SALARY,Salaries)
as (select CODE,SALARY,group_concat(start||'-'||end,',')
    from H_SAL group by CODE,SALARY),
  HH_CIT(CODE,CITY,Cities)
as (select CODE,CITY,group_concat(start||'-'||end,',')
    from H_CIT group by CODE,CITY),
  HH_PRO(CODE,PROJECT,Projects)
as (select CODE,PROJECT,group_concat(start||'-'||end,',')
    from H_PRO group by CODE,PROJECT)
insert into DOC_EMPLOYEE
select sal.CODE,sal.Salaries,cit.Cities,pro.Projects
from (select CODE,group_concat(SALARY || ':'||Salaries,';')
      as Salaries
    from HH_SAL group by CODE) sal,
     (select CODE,group_concat(CITY || ':'||Cities,';')
      as Cities
    from HH_CIT group by CODE) cit,
     (select CODE,group_concat(PROJECT || ':'||Projects,';')
      as Projects
    from HH_PRO group by CODE) pro
where sal.CODE = cit.CODE and sal.CODE = pro.CODE;

```

**Script 10.74** - Converting the contents of H\_EMPLOYEE into the document-oriented model

A simple join (or outer join, depending on whether source columns were nullable or not) will rebuild the original H\_EMPLOYEE table.

```

create table H_PRO(CODE char(5),PROJECT char(20),
                  start integer,end integer);
for cod,pro = [select CODE,Projects from DOC_EMPLOYEE];
  for project = [item(;) $pro$];
    compute value = item('$project$',1,':');
    compute intervals = item('$project$',2,':');
    for interval = [item(,) $intervals$];
      compute start = item('$interval$',1,'-');
      compute end = item('$interval$',2,'-');
      insert into H_PRO(CODE,PROJECT,start,end)
        values ('$cod$','$value$','$start$','$end$');
    endfor;
  endfor;
endfor;

```

**Script 10.75** - Rebuilding table H\_PRO from document-oriented data

## 10.14 Performance analysis and optimization

TDB.db, the database we have used so far, was fine to elaborate temporal data models and to develop querying and transforming algorithms but it is not at all suitable for measuring the performances of these algorithms. To this aim, we will use a series of experimental temporal databases with increasing size. They all include a unique table, H\_EMPLOYEE, with the same schema as that of TDB.db. They are named TDB-xxxx.db, where xxxx, ranging from 100 to 200000, is the number of employees the states of which they contain. They include about 10 states per employee (as a reminder, database TDB.db describes the 28 states of 8 employees). These eleven valid time databases can be downloaded from the SQLfast website (in Section *Technical complements*).

These databases are derived from the employees.db database available at [https://github.com/siara-cc/employee\\_db](https://github.com/siara-cc/employee_db). The latter was itself indirectly translated from an XML dataset developed by Fusheng Wang and Carlo Zaniolo in 2003. This dataset is available at <http://timecenter.cs.aau.dk/software.htm>.

To make the source database comply with our TDB.db database, we have applied the following transformations:

- The data are integrated into a single table H\_EMPLOYEE
- Rows with invalid intervals are deleted
- Employee rows where properties (title, salary, etc.) are missing are deleted
- A one-letter prefix is added to emp\_id to produce column CODE
- Columns first\_name and last\_name columns are merged to form column NAME
- Column title is converted into column PROJECT
- Column department is converted into column CITY
- Yearly salary is converted into monthly SALARY
- SALARY values are rounded to a multiple of 5
- This rounding produces consecutive value-equivalent states; they are reduced
- Symbolic future '9999-01-01' is changed to '9999-12-31'

In the following sections, we measure the execution time of the algorithms that implement the main temporal operators for each of the experimental databases. We also show that some of these algorithms can be dramatically improved by carefully selected indexes while for others, seemingly promising indexes just slow them down.

It should not be forgotten that the performance figures depend on the optimization strategies of the DBMS. Different conclusions could be drawn with other DBMS or with variants of the SQL queries developed in this case study.

## Scalability of the algorithms

A major concern, when we develop an algorithm, is the way it behaves when it processes larger data sets. This issue is commonly described by the *time complexity* of the algorithm, that is (said very simply), by the shape of the function that relates the running time against the size of the problem, generally noted  $N$ , here the number of employees.<sup>15</sup> If applying an operator on 10,000 employees takes 10 seconds, how much time will the algorithm require to process 20,000 employees? The best we can expect (in a reasonable world) is that this time will double to about 20 seconds. If this proves exact, then our algorithm will be said to have a *linear complexity*, noted  $O(N)$ . If we are unlucky, its complexity could be  $O(N \cdot \log N)$ , or even worse  $O(N^2)$ . We will pay attention to this aspect in the analysis of the algorithms of the temporal operators

### 10.14.1 Temporal projection

We have identified two patterns of SQL temporal projection operators (called predicative techniques), namely *entity-based projection* (Script 10.20) and *generalized projection* (10.22). The first one is devoted to the frequent, but limited case in which the projection columns include the components of the entity primary key, such as CODE for H\_EMPLOYEE and TITLE for H\_PROJECT. The second one does not impose any constraint on the composition of the projection columns and can be used to solve any pattern of projection.

Besides the SQL implementation of the projection operator, a *procedural algorithm* is available through function LTemp:project (Script 10.24).

Of the three algorithms, the *generalized predicative technique*, despite its elegance, is quickly disqualified from the other two algorithms: the projection on CITY,PROJECT of H\_EMPLOYEE with 1,000 employees cost 9 seconds, compared to 16 milliseconds for the procedural technique (more than 560 times slower). This technique could be used, if desired, for datasets not larger than 200 employees, if a cost of 2.23 seconds is considered acceptable. So, we discard this technique to concentrate on entity-based predicative and procedural techniques. The execution times of these techniques, for experimental databases from 1,000 to 50,000 employees (about 10,000 to 500,000 states) are shown in the table of Figure 10.39 and in the graph of Figure 10.40.

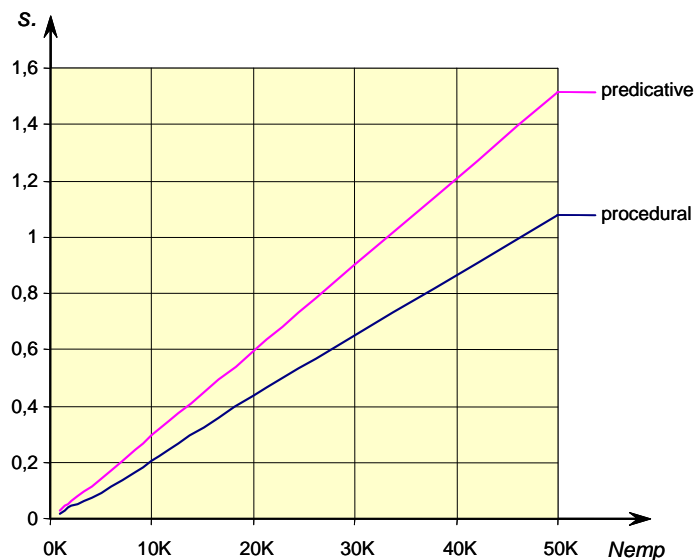
These figures show that,

- both techniques are reasonable fast,
- the procedural technique is 40% faster than the predicative technique,
- the evolution of the execution time against the database size is linear: if we double the size, we just double the time. This is natural for the procedural algorithm, which merely scans the data, but it means that the SQL optimizer was able to generate a similar query plan despite the complexity of the SQL query.

15. See [https://en.wikipedia.org/wiki/Time\\_complexity](https://en.wikipedia.org/wiki/Time_complexity)

	predicative	procedural
1K	0.031	0.015
2K	0.062	0.047
5K	0.141	0.093
10K	0.297	0.203
20K	0.594	0.437
50K	1.515	1.078

**Figure 10.39** - Computing entity-based projection H\_EMPLOYEE[CODE,CITY]



**Figure 10.40** - Graphical representation of the execution time of the projection H\_EMPLOYEE[CODE,CITY]

### 10.14.2 Temporal inner join

Joining tables requires at least two tables (unless one accepts to perform self joins only!) such as H\_EMPLOYEE and H\_PROJECT of database TDB.db. To get realistic performance figures we will prepare two joinable tables, H\_EMP1 and H\_EMP2, in databases TDB-100 to TDB-50000. The contents of H\_EMP1 is generated by the projection of H\_EMPLOYEE on columns CODE, NAME, CITY while H\_EMP2 results from the projection on CODE, SALARY, PROJECT:

```
function LTemp:project {select CODE,NAME,CITY
                        from H_EMPLOYEE into H_EMP1};
```



```
function LTemp:project {select CODE,SALARY,PROJECT
                        from H_EMPLOYEE into H_EMP2};
```

CODE is the primary key (and therefore a unique index) in both tables. On each experimental database, table H\_EMP1 contains about 1.1 state per employee (properties *name* and *city* are fairly stable) and H\_EMP2 contains about 10 states per employee. Joining these tables on entity primary key CODE provides the time scores of Figure 10.41.

	H_TEMP1 * H_EMP2
1K	0.094
2K	0.109
5K	0.172
10K	0.266
20K	0.485
50K	0.969

**Figure 10.41** - Joining tables H\_EMP1 and EMP2

These results show that the execution times follow a linear rule fairly similar to that of procedural projection (Figure 10.40).

### 10.14.3 Temporal outer join

For this experiment, we will use tables H\_EMP1 and H\_EMP2 built to study inner join. In the largest, H\_EMP2, we delete about 10% of the states, randomly selected, with the following query:

```
set step = 19;
with OID(oid) as
  (select random_i(0,$step$)
   union all
   select oid + random_i(1,$step$)
   from OID where oid < $max$ )
delete from H_EMP2 where rowid in (select oid from OID)
```

The left outer join of H\_EMP1 with H\_EMP2 proceeds in two steps: replacing the missing states in H\_EMP2 with null states (the *normalization* step) then applying an inner join between H\_EMP1 and H\_EMP2 so completed. We have proposed two normalization techniques, namely predicative (Script 10.31) and procedural (Script 10.35).

The table of Figure 10.42 and the graph of Figure 10.43 show the execution times of the *normalization* step of H\_EMP2 wrt H\_EMP1 for databases TDB-1000 to TDB-50000.

	Predicative	Procedural
1K	0.219	0.047
2K	0.297	0.078
5K	0.657	0.141
10K	1.125	0.266
20K	2.141	0.547
50K	5.047	1.375

Figure 10.42 - Outer join preparation: normalizing incomplete table H\_TEMP2

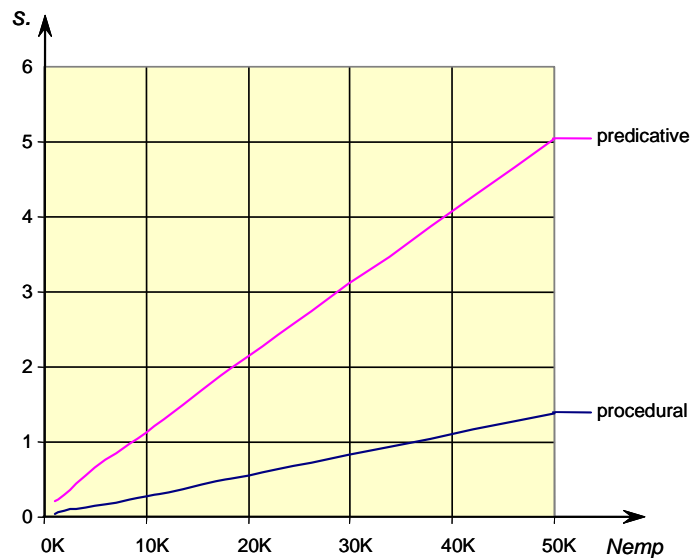


Figure 10.43 - Graphical representation of the execution time of the normalization of H\_EMP2

Once again, the normalization time follows a linear rule for both techniques. The execution time of the whole outer join process is therefore also linear. Both techniques provide reasonable times for small to medium dataset sizes. For large datasets, the procedural technique clearly is recommended.

#### 10.14.4 Aggregation

Temporal aggregation consists in applying standard **group by** queries on a special table, VALUED\_STABLE\_INTERVAL (STABLE\_EMPLOYEE is just a variant of it), derived from the source table. Due to their different nature, we consider these two steps separately.

## Step 1: computing VALUED\_STABLE\_INTERVAL

This operator comprises three main operations. We will examine each of them to analyze its behavior and to check if, and how, they can be improved.

### Computing the events of each project (table EVENT)

The algorithm we propose computes the events as the union of the sets of start and end values of H\_EMPLOYEE:

```
select PROJECT,start as Instant from H_EMPLOYEE
union
select PROJECT,end as Instant from H_EMPLOYEE;
```

Its cost accounts for **15%** of the lowest total computing time. We observe that this algorithms (probably) will scan table H\_EMPLOYEE twice and that duplicate elimination requires a preliminary sorting of the dates. To avoid these operations, we could propose the following version:

```
select PROJECT,start as Instant from H_EMPLOYEE
union all
select PROJECT,max(End) as Instant from H_EMPLOYEE;
```

It is based on the fact that the sets of values of start and end are the same, except for the lowest within the set of start values and the highest within the set of end values. The gain is quite interesting: **0.7 s.** against **3.2 s.** for the initial version, these values being observed on the largest database (Nemp = 200,000). Unfortunately, these algorithms are not equivalent. The former is valid even if the history includes gaps, while the latter requires that the history be complete.

Therefore, we keep the initial version of the algorithm, which is more general.

### Computing the stable intervals of each project (table STABLE\_INTERVAL)

The stable intervals are computed through a self-join of table EVENT, which associate with each Instant value the next value in EVENT, except for the last one:

```
select PROJECT,
Instant as Start,
(select min(Instant) from EVENT
where PROJECT= E.PROJECT
and Instant > E.Instant) as End
from EVENT E where End is not null;
```

So far, we have defined no index on EVENT. Let us create one on (PROJECT, Instant), that should speed up the join operation:

```
create index XEVENT on EVENT(PROJECT,Instant);
```

The effect of this index is particularly dramatic: while table `STABLE_INTERVAL` is created in **98 s.** without index, the latter performs this creation in no more than **0.078 s.** (index creation included), that is **0.08%** of the initial cost (observed for `Nemp = 10,000`)!

We naturally add this index.

### Computing the valued stable intervals (table `VALUED_STABLE_INTERVAL`)

This operation consists in adding to each stable interval a set of values extracted from `H_EMPLOYEE`. This is done through a simple inner join:

```
select S.PROJECT, E.SALARY, S.start, S.end
from   STABLE_INTERVAL S, H_EMPLOYEE E
where  E.PROJECT = S.PROJECT
and    E.start <= S.start and S.start < E.end;
```

`H_EMPLOYEE` has a primary key made up of columns `CODE` and `start`, which means that a unique index has been created on these columns. As to `STABLE_INTERVAL` it has so far been given no index. Let create this one, which, hopefully, may support the join operator:

```
create index XSI on STABLE_INTERVAL(PROJECT, Start);
```

Once again this index has a decisive effect: table `VALUED_STABLE_INTERVAL` is created in **93.7 s.** without index and in **12.7 s.** with this index (index creation included), that is **13.6%** of the initial cost (also observed for `Nemp = 10,000`).

We suggest to keep this index.

### The global process

Now, let us consider the cost of the whole operation. The table of Figure 10.44 shows the contribution of each subprocess for different configurations of the indexes. Label **pred (a+b)** indicates that index `EVENT(PROJECT, Instant)` is created ( $a=1$ ) or not ( $a=0$ ) and that index `STABLE_INTERVAL(PROJECT, Start)` is created ( $b=1$ ) or not. The last row gives the execution time of the procedural algorithm as provided by function `LTemp.valuedInterval`. These times have been collected for the `TDB-10000` database (10,000 employees).

Two important observations:

1. The gain provided by the indexes is very important, the execution time falling from 191.9 s. to 13 s., that is, **6.8%** of the initial time, or, more impressive, an improvement of **93.2%**.
2. The optimized predicative and procedural algorithms have the same cost, with a slight advantage for the predicative algorithm. This shows that native SQL queries may prove as efficient (and often more efficient) than procedural techniques.<sup>16</sup>

	EVENT	STABLE_INTERVAL	VALUED_ STABLE_INTERVAL	Total
<b>size</b>	28,017	28,011	30,677,511	
<b>pred (0+0)</b>	0.19	98.00	93.70	191.9
<b>pred (0+1)</b>	0.19	98.00	12.70	110.9
<b>pred (1+0)</b>	0.19	0.078	93.70	93.9
<b>pred (1+1)</b>	0.19	0.078	12.70	13.0
<b>procedural</b>				13.4

**Figure 10.44** - Detailed cost for Nemp = 10,000 employees (100,755 states)

3. The optimized predicative algorithm generates the valued stable intervals at an impressive rate of 2.36 million states per second.

The five predicative and procedural algorithmic variants described above provide the results shown in the table of Figure 10.45.

	predicative				procedural
	0+0	0+1	1+0	1+1	
<b>1K</b>	12.3	9.9	3.0	0.6	0.6
<b>2K</b>	35.9	27.5	9.5	1.6	1.7
<b>5K</b>	102.8	71.1	35.4	5.9	6.1
<b>10K</b>	191.9	110.9	93.9	13.0	13.4
<b>20K</b>	350.9	160.3	218.7	28.4	28.6
<b>50K</b>	765.9	242.1	601.7	69.9	72.8

**Figure 10.45** - Execution time (in seconds) of the main variants of algorithms for increasing values of Nemp

The figures are reported in the graph of Figure 10.46, which shows the trend of the running time of the algorithms against the increasing values of Ntemp. This trend suggests that the fastest implementations of the operator run in quasi-linear time, which is very good news.<sup>17</sup>

The choice between (optimized) predicative and procedural techniques is a matter of taste since their performances are fairly equivalent.

16. Such an observation has also been made for case study *Conway's Game of Life*, where careful SQL query tuning allowed the computing time to drop from 838 s. to just 0.6 s.

17. For TDB-200000: 300 s. for 656 million valued intervals (*predicative 1+1* technique).

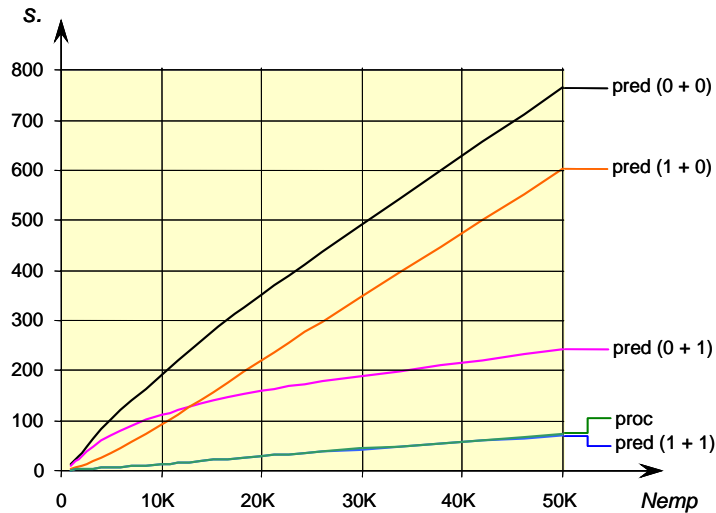


Figure 10.46 - Computing the valued stable intervals: evolution of the running time

## Step 2: data aggregation

Now, let us examine the last part of temporal aggregate queries. Once the valued stable states have been extracted, computing the statistics is expressed through simple and straightforward **group-by** SQL queries that seem to leave little room for improvement.

A popular thumb rule states that an index on the **group-by** criterion of frequent aggregate queries is likely to speed up their execution. So, let us create an index on `STABLE_EMPLOYEE(PROJECT,Start)` to count employees per project or `VALUED_STABLE_INTERVAL(PROJECT,Start)` for the other statistics. It should make the computing of statistics faster since it is built on the aggregation criterion.

Actually, it does not, as Figure 10.47, that reports on the times of the **count** function, makes it clear. Column **count** gives the execution times when no index has been created on `STABLE_EMPLOYEE`. Column **(index)+count** gives the times for the same function when an index has been created (index creation time **not included**). Column **index** provides the index creation times. Even if we ignore the latter time, computing the **count** statistics in a table devoid from index is far faster than computing in an indexed table.

The trend is similar for the other statistics. So, we abandon the idea.

We might be surprised by the length of these processes. We must consider that they are carried out on very large tables. Table `STABLE_EMPLOYEE` of `TDB-10000.db` database comprises about 31 million rows while `TDB-50000.db` generates 163 million of them.

	count	(index)+count	index
1K	1.09	1.41	1.27
2K	3.74	4.67	4.24
5K	15.10	17.63	16.14
10K	34.60	40.61	37.32
20K	75.65	89.93	83.07
50K	204.44	305.80	288.25

**Figure 10.47** - Can an index on STABLE\_EMPLOYEE help when counting employees?

## 10.15 Temporal data management in current DBMS

Introducing temporal features in SQL standard took quite a long time, nearly three decades. Since the first scientific publications in the eighties (Allen's relations were published in 1983), hundreds of proposals have been published. They were devoted to the semantics of time (what does such terms as interval, time point, now, past and future mean; temporal calculus and algebra), to its historical, geographic and cultural aspects, to temporal languages for time-dependent data and to efficient implementation of temporal operators in DBMS. SQL:2011 was the first standard to introduce SQL extensions to represent and manipulate temporal data. Though these extensions are incomplete, they make modeling and manipulation of temporal data easier and more reliable. Perhaps as important, they also pave the way to a better understanding of the temporal data paradigm by practitioners. From these proposals, most major DBMS have introduced all or a part of the temporal concepts of SQL:2011. In the rest of this section, we describe these concepts and compare them to the material presented of this case study.

SQL:2011 introduce the core concept of *period*. *Period* is not a data type but a new schema object defined in the table declaration. It is built from two time values (dates or timestamps) and is given a user name. In a *valid time table*, these time values are controlled by the applications and form an *application-time period*.

In a *transaction time table* (called *system-versioned table*) the period is a *system-time period* with standard name SYSTEM\_TIME. Its components are time-valued columns that are managed by the system.

An SQL:2011 table can be bitemporal, that is, with both valid time and transaction time dimensions.

### 10.15.1 Application-time temporal table

In this case study, both valid time and transaction time data describe the evolution of a collection of entities and of their associations. Evolution events always apply on their current states. There is no future state and past states cannot be updated.

The valid time data model of SQL:2011 is more general in that the values of both *start* and *end* temporal columns are under the responsibility of the application. This means, in particular, that

- the application is free to set both start and end times of a state,
- future events can be coped with, therefore generating future states,
- missing intervals (gaps) are allowed
- value-equivalent intervals may overlap,
- in some cases, even non value-equivalent intervals may overlap, thus specifying that an entity is in more than one state; however, the standard seems to consider this pattern as not recommended and provides for a way to reject it,
- (whole or part of) states can be modified and deleted.

The temporal dimension of a valid time table is specified by an application-controlled period, here below named *Vperiod*:

```
create table H_EMPLOYEE (
  CODE      char(5)  not null,
  NAME      char(10) not null,
  SALARY    integer not null,
  CITY      char(10) not null,
  PROJECT   char(20) not null,
  Vstart    date     not null,
  Vend      date     not null,
  period for Vperiod (Vstart,Vend) ;
```

If a primary key is to be specified, it comprises some non temporal columns (here *CODE*) + the valid time period:

```
primary key (CODE,Vperiod)
```

To avoid problematic data patterns, we add a constraint enforcing the uniqueness of rows on *CODE* values *at every time point*, that is, in every snapshot of the table

```
primary key (CODE,Vperiod without overlaps)
```

Referential integrity is expressed through a foreign key comprising some non temporal columns (here *PROJECT*) + the valid time period.

```
foreign key (PROJECT,period Vperiod)
references H_PROJECT (TITLE,period Vperiod)
```



Any state of temporal data can be modified, be it current, past or future. We consider this state of project BIOTECH:

TITLE	THEME	BUDGET	start	end
...	...	...	...	...
BIOTECH	Biotechnology	180000	2014-11-18	2015-02-27
...	...	...	...	...

It tells that, from 2014-11-18 to 2015-02-27, the budget of this project was 180,000. Now, we are told that during January of 2015, this budget actually was 150,000. So, we translate this information as follows:

```
update H_PROJECT
for portion of Vperiod from '2015-01-01' to '2015-01-31'
set BUDGET = 150000
where TITLE = 'BIOTECH';
```

The former state is replaced by these new three states:

TITLE	THEME	BUDGET	start	end
...	...	...	...	...
BIOTECH	Biotechnology	180000	2014-11-18	2015-01-01
BIOTECH	Biotechnology	150000	2015-01-01	2015-01-31
BIOTECH	Biotechnology	180000	2015-01-31	2015-02-27
...	...	...	...	...

Similarly, we can delete a state or a portion of a state:

```
delete H_PROJECT
for portion of Vperiod from '2015-01-01' to '2015-01-31'
where TITLE = 'BIOTECH';
```

A valid time table can be queried as a standard table, with conditions on any columns, including the temporal columns. Extracting the current state of all the employees is straightforward:

```
select CODE, NAME, Vstart
from H_EMPLOYEE
where Vend = '9999-12-31';
```

It can also be queried through temporal predicates based on the period of the table (they are derived from, but not identical to, Allen's relations). This query extract a snapshot of all the employees at a definite date:

```
select CODE, NAME, Vstart
from   H_EMPLOYEE
where  Vperiod contains '2016-12-31'
```

This one extract the states falling in a time slice (year 2016):

```
select CODE, NAME, Vstart
from   H_EMPLOYEE
where  Vperiod overlaps period ('2016-01-01', '2016-12-31');
```

### 10.15.2 System-versioned temporal table

The transaction time data model of SQL:2011 is very close to that developed in this case study<sup>18</sup>. A system-versioned table (*transaction time table*) comprises two timestamp columns that are set and modified by the DBMS alone. They are used to define system period `SYSTEM_TIME`. Primary and foreign keys are defined on the current state without reference to the temporal components of the table:

```
create table H_EMPLOYEE(
    CODE    char(5)    not null,
    ...,
    PROJECT char(20) not null,
    Sstart timestamp not null generated always at row start,
    Send    timestamp not null generated always at row end,
    period for SYSTEM_TIME (Sstart, Send),
    primary key (CODE),
    foreign key (PROJECT) references H_PROJECT(TITLE))
with system versioning;
```

Update and delete queries are those of non temporal tables:

```
update H_EMPLOYEE
set PROJECT = 'BIOTECH'
where CODE = 'A237';

delete H_EMPLOYEE
where CODE = 'A237';
```

Querying a system-versioned table follows a specific syntax that is different from that of application-time tables. The current states is extracted as follows:

```
select CODE, NAME, Sstart, Send
from   H_EMPLOYEE
where  ...
```

18. The main difference is that the system clock must provide sufficient precision to prevent two independent transactions to get the same timestamp. A precision of a millisecond was not considered sufficient to enforce this property, hence the need of the `CLOCK` table.

And this is how we extract a snapshot:

```
select CODE,NAME
from H_EMPLOYEE
for SYSTEM_TIME as of '2016-12-31'
where PROJECT = 'BIOTECH';
```

A time slice can be specified by a *closed-open* interval:

```
select CODE,NAME,Sstart,Send
from H_EMPLOYEE
for SYSTEM_TIME from '2016-01-01' to '2017-01-01'
where ...
```

... or by a *closed-closed* interval:

```
select CODE,NAME,Sstart
from H_EMPLOYEE
for SYSTEM_TIME between '2016-01-01' and '2016-12-31'
where ...
```

### 10.15.3 Future directions<sup>19</sup>

The SQL:2011 standard specifies the temporal data model and language extensions to formulate elementary queries. However it does not address more complex queries such as inner and outer joins, projection (and its coalescing derivative), and aggregate functions. Their expression is the responsibility of application programmers. While the translation of an inner join is quite straightforward, thanks to function overlaps, the other complex queries require much effort that prevents programmers from implementing them, as shown in this case study.

## 10.16 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory SQLfast/Scripts/Case-Studies/Case\_Temporal\_DB. Actually, they can be run from main script **TDB-MAIN.sql**, that displays the selection box of Figure 10.48.

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to let them develop their own applications.

---

19. This section relies on the eponym section of reference [Kulkarni, 2012].

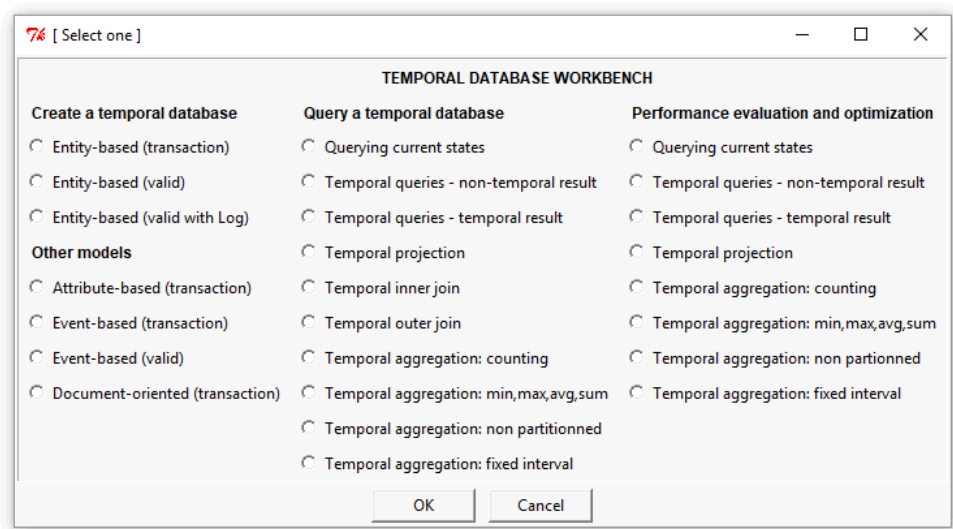


Figure 10.48 - The scripts of Part 1 and Part 2 can be run from this main panel

10.17 References

Among the many hundreds of references on temporal data management, we have chosen three of the most practical ones. They all are available on the web as free pdf documents.

Snodgrass, R., T., *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann, 2000.

This large book (528 pages) explores in much detail the concept of time and its multiple aspects. It describes various ways to model temporal data and develops SQL queries and triggers implementing temporal data management primitives (applied to base tables, not to views). Then, it analyzes and translates in SQL:1999 the main queries as well as temporal projection and joins. Finally, it describes an incremental method to build temporal databases. This text may be difficult to read and to apply practically since all the developments are based on an unnormalized temporal model, where an entity history may include gaps, overlapping states and successive value-equivalent states. This model is a generalization of the entity-based model described in this study but it makes the definition of such basic concepts as primary and foreign keys, as well as the fundamental algorithms fairly complex. This book does not address aggregation functions nor performance issues. A good reading to consolidate and extend the knowledge acquired in this case study. Also shows what lies under the hood of modern SQL:2011 DBMS.

Jensen, C., J., and Snodgrass, R., T. (Ed.), *Temporal Database Entries for the Springer Encyclopedia of Database Systems*, Springer, 2008.

A collection of 81 entries describing most of the standard and advanced concepts in temporal data management published in Springer Encyclopedia of Database Systems. Most of these entries have been retained in the 2018 edition of this encyclopedia.

<http://www.cs.arizona.edu/people/rts/sql3.html>

Kulkarni, K., Michels, J., E., *Temporal features in SQL:2011*, SIGMOD Record, September 2012 (Vol. 41, No. 3)

This article describes some of the concepts of temporal databases that have been introduced in standard SQL:2011.

Finally, the reader can find in these slideshows (November 2015) an alternative view of temporal databases:

<https://www.slideshare.net/torp42/temporal-databases-data-models>

<https://www.slideshare.net/torp42/temporal-databases-queries>

