Temporal Aggregates and Temporal Universal Quantification in Standard SQL

Esteban Zimányi
Dept. of Computer & Network Engineering
Université Libre de Bruxelles, CP 165/15
50 Av. F. Roosevelt, 1050 Bruxelles, Belgique
ezimanyi@ulb.ac.be

ABSTRACT

Although it has been acknowledged for many years that querying and updating time-varying information using standard (i.e., non-temporal) SQL is a challenging task, the proposed temporal extensions of SQL have not reach acceptance in the standardization committees. Therefore, nowadays database practitioners must still use standard SQL for manipulating time-varying information. This paper shows how to realize temporal aggregates and temporal universal quantifiers using standard SQL.

Keywords: Temporal databases, temporal aggregates, temporal universal quantifiers, SQL.

1. INTRODUCTION

Many applications need to keep the evolution of data that varies on time. For example, in order to conform with legal regulations and corporate policies, administrative databases must keep the evolution of many attributes of their employees, such as marital status, salary, affiliation, etc. Current Database Management Systems, and SQL in particular, provide little support for dealing with time-varying data: They only provide standard data types for encoding dates or timestamps. However, querying and updating time-varying data using standard SQL is a challenging task.

For this reason, in the last decades many research was devoted to the management of the temporal dimension in databases, resulting in the development of temporal databases. In particular, a temporal extension of SQL-92 called TSQL2 [7] was proposed to international standardization committees [9, 10] leading to a dedicated chapter of the SQL:1999 standard called "Part 7, SQL/Temporal". However, such an extension has not yet passed the standardization process [11, 2]. Another approach for coping with the temporal dimension in relational databases was proposed in [3].

The consequence of this state of affairs is that nowadays database practitioners are left with standard SQL for manipulating time-varying data. In [8] is presented how to manipulate temporal data with standard SQL, discussing in which situations and under which assumptions the corresponding SQL code can be applied. In particular, it is presented how to define temporal join, temporal projection, or temporal difference in SQL. However, at the best of our knowledge, there has not been studies showing how to deal with temporal aggregates as well as temporal universal quantification using standard SQL. This paper is devoted to this issue.

The paper is structured as follows. Section 2, inspired from [8], introduces temporal databases and shows how to realize temporal join and temporal projection in standard SQL. The section provides the necessary background for the rest of the paper. Sections 3 and 4, describing our contributions, are devoted to temporal aggregates and temporal universal quantification. Finally, in the conclusion we describe experimental results and comparison with related works.

2. TEMPORAL DATABASES

Employee

SSN	FirstName	LastName	BirthDate	Address

Affiliation

SSN	DNumber	FromDate	ToDate

Controls

PNumber	DNumber	FromDate	ToDate

Salary

SSN	Amount	FromDate	ToDate

WorksOn

SSN	PNumber	FromDate	ToDate

Figure 1: Example of a temporal database schema.

Figure 1 shows an example of a temporal database schema. Table `Employee` is non-temporal, while all other tables are temporal, and more precisely, they are valid-time tables: The columns `FromDate` and `ToDate` indicate when the information in the corresponding row is valid, e.g., the period of time during which an employee is affiliated to a department. As a data type for representing periods is not available in SQL, a period is encoded in two columns of type `Date`. The problem is that these peculiar columns (i.e., having such a particular semantics) are encoded in SQL in the same way as columns such as `BirthDate` which is also of type `Date` but does not have any specific semantics.

SSN	DNumber	FromDate	ToDate
123456789	D1	2002-01-01	2003-06-01
123456789	D1	2003-06-01	2003-08-01
123456789	D1	2003-08-01	3000-01-01
333444555	D2	2003-10-01	2004-01-01
333444555	D2	2004-01-01	3000-01-01

Figure 2: An example of temporal table.

Figure 2 shows an example of table `Affiliation`. A closed-open representation for periods is used, e.g., the validity of the first row is [2002-01-01, 2003-06-01). Also, a special date '3000-01-01' denotes currently-valid rows.

2.1 Temporal Join

The join operator is used to combine information scattered in different tables. If the tables to be combined are temporal, then a temporal join is needed. Expressing a temporal join in SQL requires four select statements and complex inequality predicates verifying that the validity periods of the rows to be combined intersect.

Recall from Figure 1 that tables Salary and Affiliation keep, respectively, the evolution of the salary and the evolution of the affiliation of employees. To determine the joint evolution of salary and affiliation a temporal join of these tables is needed. This can be done in SQL as follows.

```
SELECT S.SSN,Amount,DNumber,S.FromDate,S.ToDate
FROM Salary S, Affiliation A WHERE S.SSN=A.SSN
AND A.FromDate<S.FromDate AND S.ToDate<=A.ToDate
  UNION ALL
SELECT S.SSN,Amount,DNumber,S.FromDate,A.ToDate
FROM Salary S, Affiliation A
WHERE S.SSN=A.SSN AND S.FromDate>=A.FromDate
AND S.FromDate<A.ToDate AND A.ToDate<S.ToDate
  UNION ALL
SELECT S.SSN,Amount,DNumber,A.FromDate,S.ToDate
FROM Salary S, Affiliation A
WHERE S.SSN=A.SSN AND A.FromDate>=S.FromDate
AND A.FromDate<S.ToDate AND S.ToDate<A.ToDate
  UNION ALL
SELECT S.SSN,Amount,DNumber,A.FromDate,A.ToDate
FROM Salary S, Affiliation A WHERE S.SSN=A.SSN
AND A.FromDate>S.FromDate AND A.ToDate<S.ToDate
```

In the above query it is supposed that there are no duplicate rows in the tables: at each point in time an employee has one salary and one affiliation. The UNION ALL is used since the query does not generate duplicates and this is more efficient than using UNION.

Temporal join can be written in a single statement using either a CASE statement or using functions. Suppose that two functions minDate and maxDate are defined as follows.

```
CREATE FUNCTION minDate(one DATE,two DATE)
RETURNS DATE BEGIN
  RETURN CASE WHEN one<two THEN one ELSE two END
END
CREATE FUNCTION maxDate(one DATE,two DATE)
RETURNS DATE BEGIN
  RETURN CASE WHEN one>two THEN one ELSE two END
END
```

Thus, minDate and maxDate return, respectively, the minimum and the maximum of the two arguments. Using the above functions the temporal join can be defined as follows.

```
SELECT S.SSN,Amount,DNumber,
  maxDate(S.FromDate,A.FromDate) AS FromDate,
  minDate(S.ToDate,A.ToDate) AS ToDate
FROM Salary S, Affiliation A WHERE S.SSN=A.SSN
AND maxDate(S.FromDate,A.FromDate) <
    minDate(S.ToDate,A.ToDate)
```

The two functions are used in the SELECT clause for constructing the intersection of the corresponding validity periods. The condition in the WHERE clause ensures that the two validity periods overlap.

2.2 Temporal Projection

SSN	FromDate	ToDate
123456789	2002-01-01	2003-06-01
123456789	2003-06-01	2003-08-01
123456789	2003-08-01	3000-01-01
333444555	2003-10-01	2004-01-01
333444555	2004-01-01	3000-01-01

Figure 3: Projecting DNumber from Figure 2.

Given the table Affiliation of Figure 2, suppose that we want to obtain the periods of time in which an employee has worked in the company, independently of the department. Figure 3 shows the result of projecting out attribute DNumber from the table of Figure 2. As can be seen the resulting table is redundant. The first three rows are value equivalent (i.e., they equal on all their columns but FromDate and ToDate) and the validity periods of these rows meet. The situation is similar for the last two rows. Therefore, the result of the projection should be as given in Figure 4. This process of combining several value-equivalent rows into one provided that their validity periods overlap is called coalescing.

SSN	FromDate	ToDate
123456789	2002-01-01	3000-01-01
333444555	2003-10-01	3000-01-01

Figure 4: Coalescing the temporal table of Figure 3.

Coalescing is a complex and costly operation in SQL. It can be realized entirely in SQL as follows [1].

```
SELECT DISTINCT F.SSN,F.DNumber,F.FromDate,L.ToDate
FROM Affiliation F, Affiliation L
WHERE F.FromDate<L.ToDate
AND F.SSN=L.SSN AND F.DNumber=L.DNumber
AND NOT EXISTS ( SELECT * FROM Affiliation M
  WHERE M.SSN=F.SSN AND M.DNumber=F.DNumber
  AND F.FromDate<M.FromDate AND M.FromDate<=L.ToDate
  AND NOT EXISTS ( SELECT * FROM Affiliation M1
    WHERE M1.SSN=F.SSN AND M1.DNumber=F.DNumber
    AND M1.FromDate<M.FromDate
    AND M.FromDate<=M1.ToDate ) )
AND NOT EXISTS ( SELECT * FROM Affiliation M2
  WHERE M2.SSN=F.SSN AND M2.DNumber=F.DNumber
  AND ( (M2.FromDate<F.FromDate AND
        F.FromDate<=M2.ToDate)
     OR (M2.FromDate<=L.ToDate AND
        L.ToDate<M2.ToDate) ) )
```

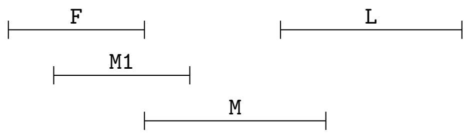


Figure 5: Coalescing value-equivalent rows.

Consider the diagram in Figure 5. Coalescing amounts to select the from and to dates of two value-equivalent rows F(irst) and L(ast) having no gap between these dates, i.e., for every value-equivalent row M whose validity period is between those of F and L there is another value-equivalent row

M1 whose validity period overlaps the beginning of M. The second NOT EXIST ensures that no other value-equivalent row M2 overlaps the period between the selected from and to dates and has an earlier from date or a later to date.

3. TEMPORAL AGGREGATION

SQL provides aggregation functions such as COUNT, MIN, MAX, and AVG. They are used for answering queries such as "List the maximum salary" or "Count the number of employees". If table Salary were non-temporal these queries can be written as follows.

```
SELECT MAX(Amount)         SELECT COUNT(*)
FROM Salary                FROM Salary
```

Another usual request is to combine rows according to a criterion specified in a GROUP BY clause previous to the application of the aggregation operator, as in the query "List the maximum salary by department". The non-temporal version of this query can be written in SQL as follows.

```
SELECT DNumber,MAX(Amount)
FROM Affiliation A, Salary S
WHERE A.SSN=S.SSN GROUP BY DNumber
```

The temporal version of the above queries require a three-step process: (i) identify the periods of time in which all values are constant, (ii) compute the aggregation over these periods, and finally (iii) coalesce the result.

For the first two queries, Figure 6 shows a diagram where table Salary has three employees E1, E2, and E3, as well as the temporal maximum and temporal count. Notice that the period in which no employee works in the company does not belong to the answer for the temporal maximum, but it appears for the temporal count with a value of 0.

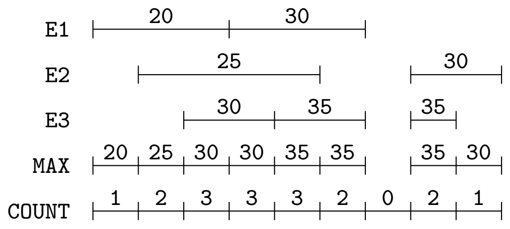


Figure 6: Evolution of the maximum salary and the number of employees.

The first step computes the periods on which the aggregation must be calculated, as follows.

```
CREATE VIEW SalChanges(Day) as
  SELECT DISTINCT FromDate FROM Salary
  UNION SELECT DISTINCT ToDate FROM Salary
CREATE VIEW SalPeriods(FromDate,ToDate) as
  SELECT P1.Day,P2.Day
  FROM SalChanges P1, SalChanges P2
  WHERE P1.Day<P2.Day AND NOT EXISTS (
    SELECT * FROM SalChanges P3
    WHERE P1.Day<P3.Day AND P3.Day<P2.Day )
```

View SalChanges gathers the days in which a salary change occurred, while view SalPeriods constructs the periods from such days.

The second step computes the aggregation on these periods. For the maximum salary this is done as shown below.

```
CREATE VIEW TempMax(MaxSalary,FromDate,ToDate) as
  SELECT MAX(Amount),P.FromDate,P.ToDate
  FROM Salary S, SalPeriods P
  WHERE S.FromDate<=P.FromDate
  AND P.ToDate<=S.ToDate
  GROUP BY P.FromDate, P.ToDate
```

Computing the number of employees is done as follows.

```
CREATE VIEW TempCount(NbEmp,FromDate,ToDate) as
  SELECT COUNT(*),P.FromDate,P.ToDate
  FROM Salary S, SalPeriods P
  WHERE S.FromDate<=P.FromDate
  AND P.ToDate<=S.ToDate
  GROUP BY P.FromDate, P.ToDate
UNION ALL
  SELECT 0,P.FromDate,P.ToDate FROM SalPeriods P
  WHERE NOT EXISTS ( SELECT * FROM Salary S
    WHERE S.FromDate<=P.FromDate
    AND P.ToDate<=S.ToDate )
```

Notice the second select that assigns a count value of 0 to those periods that do not appear in the first select.

Finally, in the third step it is necessary to coalesce the above views. This can be done as seen in Section 2.2.

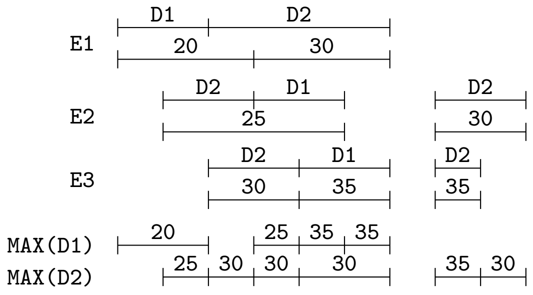


Figure 7: Maximum salary by department.

Now consider the second query asking the maximum salary by department. Figure 7 shows a diagram of possible values of tables Affiliation and Salary for employees E1, E2, and E3, and departments D1, and D2. If it is supposed that employees have salary only while they are affiliated to a department, the query can be written as follows.

In the first step it is necessary to compute by department the periods on which a maximum must be calculated.

```
CREATE VIEW Aff_Sal(DNumber,Amount,
    FromDate,ToDate) as
  SELECT DISTINCT A.DNumber,S.Amount,
    maxDate(S.FromDate,A.FromDate),
    minDate(S.ToDate,A.ToDate)
  FROM Affiliation A, Salary S WHERE A.SSN=S.SSN
  AND maxDate(S.FromDate,A.FromDate) <
      minDate(S.ToDate,A.ToDate)
CREATE VIEW SalChangesDep(DNumber,Day) as
  SELECT DISTINCT DNumber,FromDate FROM Aff_Sal
  UNION SELECT DISTINCT DNumber,ToDate FROM Aff_Sal
CREATE VIEW SalPeriodsDep(DNumber,FromDate,ToDate) as
  SELECT P1.DNumber,P1.Day,P2.Day
  FROM SalChangesDep P1, SalChangesDep P2
  WHERE P1.DNumber=P2.DNumber AND P1.Day<P2.Day
  AND NOT EXISTS ( SELECT * FROM SalChangesDep P3
    WHERE P1.DNumber=P3.DNumber AND P1.Day<P3.Day
    AND P3.Day<P2.Day )
```

View Aff_Sal realizes a temporal join of Affiliation and Salary. This yields the days in which a change of maximum salary of a department may occur. Next, view SalChanges collects such days by department, and finally view SalPeriods constructs the periods from these days.

The second step computes the maximum salary for the above periods.

```
CREATE VIEW TempMaxDep(DNumber,MaxSalary,
    FromDate,ToDate) as
  SELECT P.DNumber,MAX(Amount),P.FromDate,P.ToDate
  FROM Aff_Sal A, SalPeriodsDep P
  WHERE A.DNumber=P.DNumber
  AND A.FromDate<=P.FromDate AND P.ToDate<=A.ToDate
  GROUP BY P.DNumber, P.FromDate, P.ToDate
```

Finally, in the third step the above view is coalesced.

4. TEMPORAL UNIVERSAL QUANTIFIER

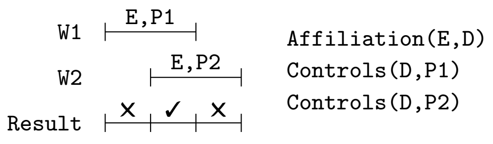
The universal quantifier is needed in many usual queries, such as "List the employees that work in **all** projects controlled by the department to which they are affiliated". As SQL does not provide the universal quantifier, the non-temporal version of the above query is written with two nested NOT EXISTS as follows.

```
SELECT SSN FROM Affiliation A WHERE NOT EXISTS (
  SELECT * FROM Controls C WHERE A.DNumber=C.DNumber
    AND NOT EXISTS ( SELECT * FROM WorksOn W
    WHERE C.PNumber=W.PNumber AND A.SSN=W.SSN ) )
```

Consider now the temporal version of the above query. As was the case for temporal aggregation, a three-step process is needed as follows: (i) identify the periods of time in which all values are constant, (ii) compute the universal quantification over these periods, and finally (iii) coalesce the result.

Four cases arise depending on whether the tables WorksOn, Affiliation, and Controls are temporal or not.

Case 1. Only WorksOn is temporal.



The above diagram shows possible values in the three tables and the result of the query. In the diagram W1 and W2 represent two rows of WorksOn relating employee E with projects P1 and P2. At the right of the diagram it is shown that employee E works in department D which controls both projects. Finally, Result shows the periods for which the universal quantification must be calculated, and whether the answer for the period is positive or negative.

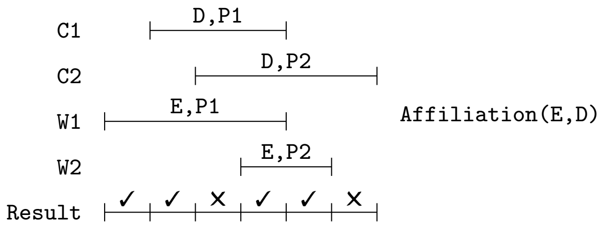
In this case the query can be written in two steps. The first step is shown next.

```
CREATE VIEW TempUnivC1(SSN,FromDate,ToDate) as
  SELECT DISTINCT W1.SSN,W1.FromDate,W2.ToDate
  FROM WorksOn W1, WorksOn W2, Affiliation A
  WHERE W1.SSN=W2.SSN AND W1.SSN=A.SSN
  AND W1.FromDate<W2.ToDate
  AND NOT EXISTS ( SELECT * FROM Controls C
    WHERE A.DNumber=C.DNumber
      AND NOT EXISTS ( SELECT * FROM WorksOn W
        WHERE C.PNumber=W.PNumber AND A.SSN=W.SSN
        AND W.FromDate<=W1.FromDate
        AND W2.ToDate<=W.ToDate ) )
```

The above view looks for two rows (possibly the same) in WorksOn from which the period in the result can be constructed, as well as a row in Affiliation determining the department to which that employee is affiliated. The two inner NOT EXISTS ensure that there is no project controlled by that department in which the employee does not work.

In the second step, the above view must be coalesced.

Case 2. Only Controls and WorksOn are temporal.



The above diagram shows possible values in the three tables and the result of the query. In the diagram C1 and C2 represent two rows of Controls relating department D with projects P1 and P2, while W1 and W2 represent two rows of WorksOn relating employee E with projects P1 and P2. At the right of the diagram it is shown that employee E is affiliated to department D. Finally, Result shows the periods for which the universal quantification must be calculated. Notice that employees may work in projects controlled by departments different to the one to which they are affiliated.

The first step constructs the periods on which the universal quantifier must be computed.

```
CREATE VIEW ProjChangesC2(SSN,Day) as
  SELECT DISTINCT SSN,FromDate
    FROM Affiliation A, Controls C
    WHERE A.DNumber=C.DNumber UNION
  SELECT DISTINCT SSN,ToDate
    FROM Affiliation A, Controls C
    WHERE A.DNumber=C.DNumber UNION
  SELECT DISTINCT SSN,FromDate FROM WorksOn UNION
  SELECT SSN,ToDate FROM WorksOn
CREATE VIEW ProjPeriodsC2(SSN,FromDate,ToDate) as
  SELECT P1.SSN,P1.Day,P2.Day
  FROM ProjChangesC2 P1, ProjChangesC2 P2
  WHERE P1.SSN=P2.SSN AND P1.Day<P2.Day
  AND NOT EXISTS ( SELECT * FROM ProjChangesC2 P3
    WHERE P1.SSN=P3.SSN AND P1.Day<P3.Day
    AND P3.Day<P2.Day )
```

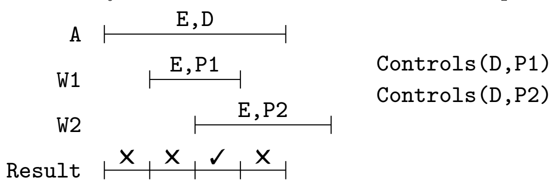
View ProjChangesC2 extracts for each employe the days in which his/her department starts or finishes to control a project, as well as days in which he/she starts or finishes to work in a project. View ProjPeriodsC2 constructs the periods from the above days.

The second step computes the universal quantifier on these periods.

```
CREATE VIEW TempUnivC2(SSN,FromDate,ToDate) as
  SELECT DISTINCT P.SSN,P.FromDate,P.ToDate
  FROM ProjPeriodsC2 P, Affiliation A
  WHERE P.SSN=A.SSN AND NOT EXISTS (
    SELECT * FROM Controls C WHERE A.DNumber=C.DNumber
    AND C.FromDate<=P.FromDate AND P.ToDate<=C.ToDate
    AND NOT EXISTS ( SELECT * FROM WorksOn W
      WHERE C.PNumber=W.PNumber AND P.SSN=W.SSN
      AND W.FromDate<=P.FromDate
      AND P.ToDate<=W.ToDate ) )
```

Finally, the third step is to coalesce the above view.

Case 3. Only `Affiliation` and `WorksOn` are temporal.



The above diagram shows possible values in the tables and the result of the query. In the diagram `W1` and `W2` represent two rows of `WorksOn` relating employee `E` with projects `P1` and `P2`, while `A` represents a row of `Affiliation` relating employee `E` with department `D`. The right of the diagram shows that department `D` controls both projects `P1` and `P2`. Finally, `Result` shows the periods for which the universal quantification must be calculated. As shown in the diagram, no hypothesis is made about the projects in which employees work, i.e., employees may work in projects controlled by departments different to the one to which they are affiliated.

For this query, the first step constructs the periods on which the universal quantifier must be computed.

```
CREATE VIEW Aff_WO(SSN,DNumber,FromDate,ToDate) as
  SELECT DISTINCT A.SSN,A.DNumber,
    maxDate(A.FromDate,W.FromDate),
    minDate(A.ToDate,W.ToDate)
  FROM Affiliation A, WorksOn W WHERE A.SSN=W.SSN
  AND maxDate(A.FromDate,W.FromDate) <
      minDate(A.ToDate,W.ToDate)
CREATE VIEW ProjChangesC3(SSN,DNumber,Day) as
  SELECT DISTINCT SSN,DNumber,FromDate
    FROM Aff_WO UNION
  SELECT DISTINCT SSN,DNumber,ToDate
    FROM Aff_WO UNION
  SELECT SSN,DNumber,FromDate
    FROM Affiliation UNION
  SELECT SSN,DNumber,ToDate FROM Affiliation
CREATE VIEW ProjPeriodsC3(SSN,DNumber,
    FromDate,ToDate) as
  SELECT P1.SSN,P1.DNumber,P1.Day,P2.Day
  FROM ProjChangesC3 P1, ProjChangesC3 P2
  WHERE P1.SSN=P2.SSN
  AND P1.DNumber=P2.DNumber AND P1.Day<P2.Day
  AND NOT EXISTS ( SELECT * FROM ProjChangesC3 P3
    WHERE P1.SSN=P3.SSN AND P1.DNumber=P3.DNumber
    AND P1.Day<P3.Day AND P3.Day<P2.Day )
```

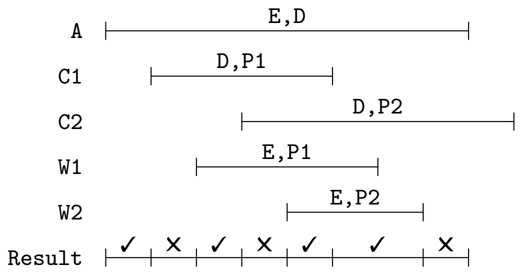
View `Aff_WO` realizes a temporal join of `Affiliation` and `WorksOn` (without the `PNumber` attribute). This collects the days when an employee starts or finishes to work in a project of his/her department. Then, view `ProjChangesC3` extracts those days from the previous view as well as the start and end day of affiliation of an employee to a department. The latter are needed to take into account the period between the beginning of an affiliation and the first time that the employee works in a project (as shown in the diagram), and the period between the last time that the employee works in a project and the end of the affiliation. Finally, view `ProjPeriodsC3` constructs the periods from the above days.

The second step computes the universal quantifier on these periods.

```
CREATE VIEW TempUnivC3(SSN,FromDate,ToDate) as
  SELECT DISTINCT P.SSN,P.FromDate,P.ToDate
  FROM ProjPeriodsC3 P WHERE NOT EXISTS (
    SELECT * FROM Controls C WHERE P.DNumber=C.DNumber
    AND NOT EXISTS ( SELECT * FROM WorksOn W
      WHERE C.PNumber=W.PNumber AND P.SSN=W.SSN
      AND W.FromDate<=P.FromDate
      AND P.ToDate<=W.ToDate ) )
```

Finally, the third step is to coalesce the above view.

Case 4. All tables are temporal.



The above diagram shows possible values in the three tables and the result of the query. In the diagram `W1` and `W2` represent two rows of `WorksOn` relating employee `E` with projects `P1` and `P2`, `C1` and `C2` represent two rows of `Controls` relating department `D` with the two projects, and `A` represents a row of `Affiliation` relating employee `E` with department `D`. Finally, `Result` shows the periods for which the universal quantification must be calculated. Notice that the end date of `W1` does not induce a period in the result since at that time project `P1` is not controlled by `E`'s department.

For this query the first step constructs the periods on which the universal quantifier must be computed as follows.

```
CREATE VIEW Aff_Cont(SSN,DNumber,PNumber,
    FromDate,ToDate) as
  SELECT DISTINCT A.SSN,A.DNumber,C.PNumber,
    maxDate(A.FromDate,C.FromDate),
    minDate(A.ToDate,C.ToDate)
  FROM Affiliation A, Controls C
  WHERE A.DNumber=C.DNumber
  AND maxDate(A.FromDate,C.FromDate) <
      minDate(A.ToDate,C.ToDate)
CREATE VIEW Aff_Cont_WO(SSN,DNumber,PNumber,
    FromDate,ToDate) as
  SELECT DISTINCT A.SSN,A.DNumber,W.PNumber,
    maxDate(A.FromDate,W.FromDate),
    minDate(A.ToDate,W.ToDate)
  FROM Aff_Cont A, WorksOn W
  WHERE A.PNumber=W.PNumber AND A.SSN=W.SSN
  AND maxDate(A.FromDate,W.FromDate) <
      minDate(A.ToDate,W.ToDate)
CREATE VIEW ProjChangesC4(SSN,DNumber,Day) as
  SELECT DISTINCT SSN,DNumber,FromDate
```

```
    FROM Aff_Cont UNION
  SELECT DISTINCT SSN,DNumber,ToDate
    FROM Aff_Cont UNION
  SELECT DISTINCT SSN,DNumber,FromDate
    FROM Aff_Cont_WO UNION
  SELECT DISTINCT SSN,DNumber,ToDate
    FROM Aff_Cont_WO UNION
  SELECT SSN,DNumber,FromDate
    FROM Affiliation UNION
  SELECT SSN,DNumber,ToDate FROM Affiliation
CREATE VIEW ProjPeriodsC4(SSN,DNumber,
    FromDate,ToDate) as
  SELECT P1.SSN,P1.DNumber,P1.Day,P2.Day
  FROM ProjChangesC4 P1, ProjChangesC4 P2
  WHERE P1.SSN=P2.SSN AND P1.DNumber=P2.DNumber
  AND P1.Day<P2.Day AND NOT EXISTS (
    SELECT * FROM ProjChangesC4 P3
    WHERE P1.SSN=P3.SSN AND P1.DNumber=P3.DNumber
    AND P1.Day<P3.Day AND P3.Day<P2.Day )
```

View `Aff_Cont` realizes a temporal join of tables `Affiliation` and `Controls`. This view is then used on view `Aff_Cont_WO` to realize a temporal join of the three tables `Affiliation`, `Controls`, and `WorksOn`. This computes the days in which an employee starts or finishes to work in a project of his/her department. Then, view `ProjChangesC4` extracts those days from the previous views as well as the start and end day of affiliation of an employee to a department. Finally, view `ProjPeriodsC4` constructs the periods from these days.

The second step computes the universal quantifier on these periods as follows.

```
CREATE VIEW TempUnivC4(SSN,FromDate,ToDate) as
  SELECT DISTINCT P.SSN,P.FromDate,P.ToDate
  FROM ProjPeriodsC4 P WHERE NOT EXISTS (
    SELECT * FROM Controls C WHERE P.DNumber=C.DNumber
    AND C.FromDate<=P.FromDate AND P.ToDate<=C.ToDate
    AND NOT EXISTS ( SELECT * FROM WorksOn W
      WHERE C.PNumber=W.PNumber AND P.SSN=W.SSN
      AND W.FromDate<=P.FromDate
      AND P.ToDate<=W.ToDate ) )
```

Finally, in the third step the above view is coalesced.

5. CONCLUSION

In this paper we adopted a practitioner's approach and showed how to compute temporal aggregation and temporal universal quantification in standard SQL. This provides a solution for users that need such time-varying facilities in their applications.

We randomly generated a set of coalesced tables having 100, 1K, 10K, 100K, and 1M lines using SQL Server 2000 on a Pentium 4 machine with 1G of RAM. Our experimental results showed that the complexity of the queries is high. Performance may be significantly improved by using procedural SQL (e.g., T-SQL for SQL Server) with cursors in some steps. In particular, the views `SalPeriods`, `SalPeriodsDep`, and the 3 views `ProjPeriodsC2-C4` are extremely inefficient while being conceptually very simple: they just construct time periods from a set of dates. The reason comes from the inner `NOT EXISTS` used for ensuring that `P2.Day` is the next date from `P1.Day`. Replacing these views by T-SQL procedures accessing the dates in ascending order using cursors decreased considerably the complexity. Similarly, for large tables it is more efficient to realize coalescing using cursors. instead of the declarative query given in Section 2.2. Pursuing these optimization efforts constitutes a direction for our future work.

Obviously, the best solution would be that the DBMS provide such time-varying facilities in a native way, since that would increase both database performance and application development productivity. Several solutions have been proposed for computing temporal aggregates such as aggregation trees [5], SB-trees [6] or balanced trees [12].

6. REFERENCES

[1] M. Böhlen, R. Snodgrass, and M. Soo. Coalescing in temporal databases. In *Proc. of the 22th Int. Conf. on Very Large Data Bases*, pages 180–191, 1996.

[2] H. Darwen. Valid time and transaction time proposals: Language design aspects. In Etzion et al. [4], pages 195–210.

[3] C. Date, H. Darwen, and N. Lorentzos. *Temporal Data and the Relational Model*. Morgan Kaufmann, 2002.

[4] O. Etzion, S. Jajodia, and S. Sripada, editors. *Temporal Databases: Research and Practice*. LNCS 1399. Springer-Verlag, 1998.

[5] N. Kline and R. Snodgrass. Computing temporal aggregates. In *Proc. of the 11th Int. Conf. on Data Engineering*, pages 222–231, 1995.

[6] B. Moon, I. Vega-López, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *IEEE Trans. on Knowledge and Data Engineering*, 15(3):744–759, 2003.

[7] R. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

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

[9] R. Snodgrass, M. Böhlen, C. Jensen, and N. Kline. Adding valid time to SQL/Temporal. ANSI X3H2-96-501r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2, 1996.

[10] R. Snodgrass, M. Böhlen, C. Jensen, and A. Steiner. Adding transaction time to SQL/Temporal: Temporal change proposal. ANSI X3H2-96-152r, ISO-ANSI SQL/ISO/IECJTC1/SC21/WG3 DBL MCI-143, 1996.

[11] R. Snodgrass, M. Böhlen, C. Jensen, and A. Steiner. Transitioning temporal support in TSQL2 to SQL3. In Etzion et al. [4], pages 150–194.

[12] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *Very Large Data Bases Journal*, 12(3):262–283, 2003.

Acknowledgements The author would like to thank Jef Wijsen and Alain Pirotte for their insightful comments that allowed to improve the quality of the paper.