REVIEW PAPER
International Journal of Recent Trends in Engineering, Vol 2, No. 4, November 2009

On Implementing Joins, Aggregates and Universal Quantifier in Temporal Databases using SQL Standards

Unnikrishnan K[1], Dr. K.V Pramod[2],
[1] Caledonian College of Engineering, CPO Seeb, Sultanate of Oman
Email: unni_krishnan_k@yahoo.com
[2] Dept. of Computer Applications, Cochin University of Science and Technology
Email: pramod_k_v@cusat.ac.in

***Abstract*— A feasible way of implementing a temporal database is by mapping temporal data model onto a conventional data model followed by a commercial database management system. Even though extensions were proposed to standard SQL for supporting temporal databases, such proposals have not yet come across standardization processes. This paper attempts to implement database operators such as aggregates and universal quantifier for temporal databases, implemented on top of relational database systems, using currently available SQL standards.**

***Index Terms*— Temporal Databases, SQL Standards, Aggregates, Universal Quantifier, TSQL**

I. INTRODUCTION

Temporal Database stores time-varying information. It supports different aspects of time. But user-defined time is not a part of the temporal database concept. Temporal database can represent the evolution of an entity in the database with respect to time. In order to achieve these features, different kinds of time-stamps will be added to data. One type of time-stamps is known as *transaction time* - the time that represents when the data is added or stored in the database. Another type of time-stamp is *valid time* - the time that represents when data is true or valid in the mini-world. When both valid-time and transaction-time time-stamps are used in a temporal database, then it is known as *bitemporal database*. There are different ways of representing the time-stamps – as time instants and time intervals. [1] - [4].

There are mainly two feasible ways of implementing temporal databases. One way is to develop a database system with temporal features from the scratch. But it will be a complex time consuming task. Another feasible way is to take a conventional commercial database management system and add temporal features to that [3] [5]. For example Oracle can be chosen as the conventional commercial database management system and the relational model features of Oracle are taken into consideration. The implementation of the temporal database can be done by mapping the temporal data model into the underlying relational model followed by Oracle. The implicit attributes, validity start time and validity end time, for data items can be represented as explicit attributes of the relation schema.

Here an example of temporal database schema for an employee database is given. This database is similar to the employee database followed in [2]. It consists of two relations Employee and Employee_Sal as given in Table I and Table II respectively. The non-temporal Employee relation maintains the details of individual employees. Employee_Sal is an example of a temporal relation with EMP_SAL as the time-varying attribute. In this relation, validity start time and validity end time for the temporal attribute EMP_SAL are represented by the explicit attributes START_DATE and END_DATE respectively. An example of Employee_Sal relation instance is given in Table III.

However manipulating temporal databases through currently available SQL standards is a complex task. Because of this rationale, a lot of research work was done on temporal extensions to SQL standards. TSQL2, extensions to the 1992 edition of the SQL standard were proposed [6], resulting in a chapter named Part 7 SQL/Temporal in the SQL: 1999 standard [7] [8]. But such extensions have not been accepted by the standardization processes [9] – [10].

In the current scenario the application developers take the help of currently available SQL standards for performing data manipulation of temporal databases. [11] gives a clear direction of using standard SQL for temporal database manipulation with suitable examples for temporal operations such as temporal joins, temporal coalescing etc. In this paper an attempt is made to define the methods that can be used for implementing aggregates and universal quantification in temporal databases. This study aims to provide a proper direction towards implementing these operators on a DBMS engine.

TABLE I. EMPLOYEE

EMP_ID	EMP_NAME	EMP_ADDRESS	DATE_OF_JOIN

TABLE II. EMPLOYEE_SAL

EMP_ID	EMP_SAL	START_DATE	END_DATE

TABLE III. EMPLOYEE_SAL RELATION INSTANCE

EMP_ID	EMP_SAL	START_DATE	END_DATE
101	10000	01/Jan/05	31/Dec/06
101	12000	01/Jan/07	31/Dec/07

II. IMPLEMENTING JOIN OPERATOR IN TEMPORAL DATABASES

Through join operation, tuples from two or more relations or views can be combined. Joins may be required by temporal database applications also. If through a Join operation two or more temporal relations are combined, it is a temporal join.

The employee database explained in introduction section has two relations Employee and Employee_Sal where Employee_Sal is a temporal relation. Assume there is one more relation Employee_Dept in this database with the schema as shown in Table IV. Employee_Dept is a temporal relation with time-varying attribute EMP_DEPT which describes the department in which an employee works during a particular time period. In order to find out the evolution of salary and department together, a join involving two relations Employee_Sal and Employee_Dept is required. This join can be implemented using standard SQL as follows:

```
SELECT A.EMP_ID, EMP_SAL, EMP_DEPT,
A.START_DATE, A.END_DATE
FROM EMPLOYEE_SAL A, EMPLOYEE_DEPT B
WHERE A.EMP_ID = B.EMP_ID
AND B.START_DATE < A.START_DATE AND A.END_DATE <=
B.END_DATE
UNION ALL
SELECT A.EMP_ID, EMP_SAL, EMP_DEPT, A.START_DATE,
B.END_DATE FROM EMPLOYEE_SAL A, EMPLOYEE_DEPT B
WHERE A.EMP_ID = B.EMP_ID
AND A.START_DATE >= B.START_DATE AND A.START_DATE <
B.END_DATE AND B.END_DATE < A.END_DATE
UNION ALL
SELECT A.EMP_ID, EMP_SAL, EMP_DEPT,
B.START_DATE, A.END_DATE
FROM EMPLOYEE_SAL A, EMPLOYEE_DEPT B
WHERE A.EMP_ID = B.EMP_ID
AND B.START_DATE >= A.START_DATE AND B.START_DATE <
A.END_DATE AND A.END_DATE < B.END_DATE
UNION ALL
SELECT A.EMP_ID, EMP_SAL, EMP_DEPT,
B.START_DATE, B.END_DATE
FROM EMPLOYEE_SAL A, EMPLOYEE_DEPT B
WHERE A.EMP_ID = B.EMP_ID
AND B.START_DATE > A.START_DATE AND B.END_DATE <
A.END_DATE
```

TABLE IV. EMPLOYEE_DEPT

EMP_ID	EMP_DEPT	START_DATE	END_DATE

It is assumed that there are no duplicate tuples the above two relations. The advantage of using UNION ALL clause than UNION here is that the query does not produce duplicates. In order to reduce the complexity of the above query, functions can be used. Using two functions the above query can be simplified as follows:

```
SELECT A.EMP_ID, EMP_SAL, EMP_DEPT
UPPER_DATE (A.START_DATE, B.START_DATE) AS START_DATE,
LOWER_DATE (A.END_DATE, B.END_DATE) AS END_DATE
FROM EMPLOYEE_SAL A, EMPLOYEE_DEPT B
WHERE A.EMP_ID = B.EMP_ID
AND HIGH (A.START_DATE, B.START_DATE) < LOW
(A.END_DATE, B.END_DATE)
```

The function HIGH can be defined as follows:

```
CREATE OR REPLACE UPPER_DATE (d1 DATE, d2 DATE)
RETURNS DATE
BEGIN
        RETURN CASE WHEN d1 > d2 THEN d1
                                ELSE d2 END
END
```

The function LOW can be defined as follows:

```
CREATE OR REPLACE LOWER_DATE (d1 DATE, d2 DATE)
RETURNS DATE
BEGIN
        RETURN CASE WHEN d1 < d2 THEN d1
                                ELSE d2 END
END
```

UPPER_DATE and LOWER_DATE functions in SELECT clause are supporting to make the intersection of corresponding validity time periods. Here the WHERE clause condition is used for testing the overlapping of validity time periods

III. IMPLEMENTING AGGREGATES IN TEMPORAL DATABASES

Aggregate functions in SQL return a single value based on values in an attribute of a relation. COUNT, AVG, MIN and MAX are the different aggregate functions defined in SQL. GROUP BY clause in SQL helps in combining the data across several records and then grouping the results by one or more attributes. Aggregate functions may be required by temporal database applications also. For example, retrieving the highest salary for the above employee database schema involves the use of aggregate function MAX. First step in retrieving the result for such a query is finding the periods during which aggregate should be computed. This can be done by defining a VIEW as follows: First using a view assemble the dates in which salary was modified and then using another view build the periods based on those dates. This is defined as follows:

```
CREATE VIEW MODIFYDAYS (MODIFY_DATE) AS
SELECT DISTINCT START_DATE FROM EMPLOYEE_SAL
UNION
SELECT DISTINCT END_DATE FROM EMPLOYEE_SAL

CREATE VIEW MODIFYPERIODS (START_DATE, END_DATE) AS
SELECT A.MODIFY_DATE, B.MODIFY_DATE
FROM MODIFYDAYS A, MODIFYDAYS B
WHERE A.MODIFY_DATE < B.MODIFY_DATE
AND NOT EXISTS (SELECT * FROM MODIFYDAYS C
WHERE A.MODIFY_DATE < C.MODIFY_DATE
AND C.MODIFY_DATE< B.MODIFY_DATE)
```

The view MODIFYDAYS finds the dates on which SALARY attribute was modified and the view MODIFYPERIODS assembles those dates into periods. Now the aggregates can be found based on these periods.

```
CREATE VIEW MAX_SAL (SAL, START_DATE, END_DATE) AS
SELECT MAX (EMP_SAL), B.START_DATE, B.END_DATE
FROM EMPLOYEE_SAL A, MODIFYPERIODS B
```

```
WHERE A.START_DATE <= B.START_DATE
AND B.END_DATE <= S.END_DATE
GROUP BY B.START_DATE, B.END_DATE
```

In this way, the maximum salary during valid periods can be retrieved. Now the possible implementation of an aggregate function with the group by clause for this temporal database application can be done. For example, retrieving the highest salaries for each department involves the use of aggregate function MAX along with the GROUP BY clause. This can be done by first making a join (which is temporal join as mentioned in section II) among the EMPLOYEE_SAL and EMPLOYEE_DEPT temporal relations. That is, define a view which will compute this join and retrieve the dates in which the maximum salary of a department is updated. Then bring together those dates department wise and build valid periods based on these dates. Then figure out the maximum salaries for these periods. This can be done as follows.

```
CREATE VIEW MODIFYDAYS_MAXSAL (EMP_DEPT, EMP_SAL,
START_DATE, END_DATE) AS
SELECT DISTINCT A.EMP_DEPT, B.EMP_SAL
UPPER_DATE (B.START_DATE, A.START_DATE),
LOWER_DATE (B.END_DATE, A.END_DATE
FROM EMPLOYEE_DEPT A, EMPLOYEE_SAL B
WHERE A.EMP_ID=B.EMP_ID
AND UPPER_DATE (B.FROM_DATE, A.FROM_DATE) <
LOWER_DATE (B.END_DATE, A.END_DATE)
CREATE VIEW MODIFYDAYS_MAXSAL_DEP (EMP_DEPT,
MODIFY_DATE) AS
SELECT DISTINCT EMP_DEPT, START_DATE FROM
MODIFYDAYS_MAXSAL
UNION
SELECT DISTINCT EMP_DEPT, END_DATE FROM
MODIFYDAYS_MAXSAL
CREATE VIEW MODIFYPERIODS_MAXSAL_DEP(EMP_DEPT,
START_DATE, END_DATE) AS SELECT A.EMP_DEPT,
A.MODIFY_DATE, B.MODIFY_DATE
FROM MODIFYDAYS_MAXSAL_DEP A,
MODIFYDAYS_MAXSAL_DEP B
WHERE A.EMP_DEPT = B.EMP_DEPT AND A.MODIFY_DATE <
B.MODIFY_DATE AND NOT EXISTS (SELECT * FROM
MODIFYDAYS_MAXSAL_DEP C
WHERE A.EMP_DEPT = C.EMP_DEPT AND A.MODIFY_DATE <
C.MODIFY_DATE AND C.MODIFY_DATE < B.MODIFY_DATE)
```

Now the maximum salary for these periods can be retrieved using another view as follows:

```
CREATE VIEW MAX_SAL_BY_DEPT (EMP_DEPT, MAX_SAL,
START_DATE, END_DATE)
AS
SELECT B.EMP_DEPT, MAX (EMP_SAL), B.START_DATE,
B.END_DATE
FROM MODIFYDAYS_MAXSAL A,
MODIFYPERIODS_MAXSAL_DEP B
WHERE A.EMP_DEPT=B.EMP_DEPT
AND A.START_DATE <= B.START_DATE AND B.END_DATE <=
A.END_DATE
GROUP BY B.EMP_DEPT, B.START_DATE, B.END_DATE
```

IV. IMPLEMENTING UNIERSAL QUANTIFIER IN TEMPORAL DATABASES

Universal quantification denotes that something is true for everything. This operator is required in many queries. Standard SQL does not directly provide any universal quantification operator and its implementation can be done with the help of NOT EXISTS clause.

Universal quantifier may be required by temporal database applications also. For example, assume that in the employee database, there are two relations Department_Proj and Employee_Proj, with the schema as shown in Table V and Table VI respectively.

Employee_Proj is a temporal relation with time-varying attribute EMP_PROJ describes the project in which an employee works during a particular time period. Application of universal quantifier is required for retrieving the employee who works on all projects monitored by their department. Department_Proj is another temporal relation describing the projects monitored by a department during a particular time period.

TABLE V. DEPARTMENT_PROJ

EMP_DEPT	EMP_PROJ	START_DATE	END_DATE

TABLE VI. EMPLOYEE_PROJ

EMP_ID	EMP_PROJ	START_DATE	END_DATE

In order to retrieve the set of employees working on all the projects monitored by their department, universal quantifier is required. In this scenario, both Department_Proj and Employee_Proj are temporal relations. The first task in evaluating this query is to build the validity periods over which the universal quantifier is to be applied. This can be achieved by a view as follows:

```
CREATE VIEW EMP_PROJ_MODIFY_DATE (EMP_ID,
FROM_TO_DATE) AS
SELECT DISTINCT EMP_ID, START_DATE
FROM EMPLOYEE_DEPT A, DEPARTMENT_PROJ B
WHERE A.EMP_DEPT = B.EMP_DEPT
UNION
SELECT DISTINCT EMP_ID, END_DATE
FROM EMPLOYEE_DEPT A, DEPARTMENT_PROJ B
WHERE A.EMP_DEPT = B.EMP_DEPT
UNION
SELECT DISTINCT EMP_ID, START_DATE
FROM EMPLOYEE_PROJ
UNION
SELECT DISTINCT EMP_ID, END_DATE
FROM EMPLOYEE_PROJ
```

This view takes out the dates in which the department in which he is working begins or ends controlling a project and also the dates in which the employee begins or ends in working in a project. Now periods have to be constructed based on these dates. This can be done with another view as follows:

```
CREATE VIEW MODIFY_PERIODS_PROJ_EMP (EMP_ID,
START_DATE, END_DATE)
AS
SELECT A.EMP_ID, A.FROM_TO_DATE, B.FROM_TO_DATE
FROM EMP_PROJ_MODIFY_DATE A, EMP_PROJ_MODIFY_DATE
B
WHERE A.EMP_ID = B.EMP_ID
AND A.FROM_TO_DATE < B.FROM_TO_DATE
```

*AND NOT EXISTS (SELECT * FROM PROJ_MODIFY_DATE C WHERE A.EMP_ID = C.EMP_ID AND A.FROM_TO_DATE <C.FROM_TO_DATE AND C.FROM_TO_DATE <B.FROM_TO_DATE)*

Now universal quantifier can be calculated over these time periods.

*CREATE VIEW UNIV_QUANTIFIER (EMP_ID, START_DATE, END_DATE) AS SELECT DISTINCT A.EMP_ID, A.START_DATE, A.END_DATE FROM MODIFY_PERIODS_PROJ_EMP A, EMPLOYEE_DEPT B WHERE A.EMP_ID = B.EMP_ID AND NOT EXISTS (SELECT * FROM DEPARTMENT_PROJ C WHERE B.EMP_DEPT = C.EMP_DEPT AND C.START_DATE <= A.START_DATE AND A.END_DATE <= C.END_DATE AND NOT EXISTS (SELECT * FROM EMPLOYEE_PROJ D WHERE C.EMP_PROJ = D.EMP_PROJ AND A.EMP_ID = D.EMP_ID AND D.START_DATE <= A.START_DATE AND A.END_DATE <= D.END_DATE))*

V. CONCLUSION

This paper attempts to realize aggregates and universal quantifier for temporal databases. Manipulating temporal databases with standard SQL is a complex time consuming task. Even though extensions were proposed to standard SQL for supporting temporal databases, such proposals have not yet come across standardization processes. In this scenario, application developers still make use of standard SQL for temporal database manipulations. Considering this well acknowledged fact, this paper attempts to implement some essential operators like aggregation and universal quantifier for temporal databases implemented on top of relational databases using currently available SQL standards.

For the experimentation, relations instances corresponding to the relations schemas as given in Table 1 through VI were simulated with different number of tuples 10K and 100K and the queries were executed on Oracle database server 10g on a system with AMD Athlon processor and IGB RAM and it was proved the complexity of the queries are high. Performance improvement can be done with the use of cursors. The views like MODIFYPERIODS_MAXSAL_DEP MODIFYPERIOD, are really adding to the complexity. But these views were conceptually really simple as they make periods out of dates and the major reason for this is the inner NOT EXISTS clause. If these views can be replaced by PL/SQL procedures which access these dates in ascending order generated by a cursor the complexity can be really reduced. Optimizing the implementation of these operators is a major future challenge.

REFERENCES

[1] C.D Dyreson, C.S Jensen. "*A Consensus Glossary of Temporal Database Concepts*." February, 1998 Vesrion. [Online] Available http://www.timeconsult.com/Publications/diss.pdf

[2] Navathe, Elmasri. "Fundamentals of Database Systems". Pearson Education Inv., Third Edition, 2000

[3] Nina Edelweiss, Patricia Hubler. Implementing a Temporal Database on top of a Conventional Database: Mapping of the Data Model and Data efinition Management. In *Proc. 15th razilian Symposium on Databases (SBBD)*, 2000.

[4] R.T Snodgrass, C.S Jensen. Temporal Data Management. *IEEE Trans. On Knowledge and Data Engineering*, 11(1), , Jan-Feb 1999.

[5] A.U Tansel. Temporal Relational Data Model. *IEEE Trans. on Knowledge and Data Engineering*, 9(3), May-June 1999.

[6] R.T Snodgrass The TSQL2 Temporal Query Language, Kluwer 1995, ISBN 0-7923-9614-6.

[7] R.T Snodgrass, M.Bohlen, C.Jensen and N. Kline Adding Valid Time to SQL/Temporal ANSI X3H2-96-501r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2, 1996.

[8] R T Snodgrass, M.Bohlen, C.Jensen and and A Steiner, Adding Transaction Time to SQL/Temporal ANSI X3H2-96-152r, ISO-ANSI SQL/ISO/IEC JTC1/SC21/WG3 DBL MCI-143, 1996.

[9] Hugh Darwen, " Valid Time and Transaction Time Proposals: Language Design Aspects", in *Temporal Databaseses Reseacrh and Practice* , vol 1399/1998, Springer Berlin/Heidelberg,1998, pp.195-210

[10] R.T Snodgrass, M.Bohlen, C.Jensen and A Steiner, "Transitioning Temporal Support in TSQL2 to SQL3", in *Temporal Databasese Reseacrh and Practice* , vol 1399/1998, Springer Berlin/Heidelberg,1998, pp.150-194

[11] R.T Snodgrass, "Developing Time-Oriented Database Applications in SQL", Mauragn Kaufman Publishers, 2000.