

INDIAN INSTITUTE OF MANAGEMENT CALCUTTA

WORKING PAPER SERIES

WPS No. 617/ January 2008

Studies in Temporal Databases

by

Anurag D and Anup Kumar Sen

IIM Calcutta, Diamond Harbour Road, Joka P.O., Kolkata 700104 India

Studies in Temporal Databases

Anurag D¹, Anup K Sen¹

¹Indian Institute of Management Calcutta, India
anurag@email.iimcal.ac.in, sen@iimcal.ac.in

Abstract. Among various models like TSQL2, XML and SQL:2003 for temporal databases, TSQL2 is perhaps the most comprehensive one and introduces new constructs for temporal query support. However, these constructs mask the user from the underlying conceptual model and essentially provides a “black box” flavour. Further, the temporal attributes are not directly accessible. In this paper, we propose a “chronon” based conceptual (relational) model, which works on a tuple timestamping paradigm. Here, the time attribute is treated in similar fashion as any other attribute and is available to the user for inclusion in SQL statements, making the model easy to understand and use. A corresponding efficient physical model based on the attribute versioning format akin to XML/SQL:2003 has also been proposed and the mapping between the proposed conceptual and physical models has been described. We evaluate the model by comparing with TSQL2 and SQL:2003, and show its equivalence to Allen’s algebra.

Keywords: Temporal Database, Chronon Model, TSQL2

1. Introduction

The need for a temporal database has been felt for over two decades now [3]. A *temporal database* attempts to capture time varying information in an efficient manner. Examples of temporal database applications include healthcare, where patient histories are to be maintained; scientific databases, where storage of timing sequences associated with experiments are critical; production environment where date and time of productions and use must be stored; multi-media databases, which need to synchronise the timing of the video and audio components in a multi-media object; and many more. However, till date there exists no universally accepted standard. Among various models proposed, the TSQL2 standard is perhaps the widely studied one. TSQL2 incorporates new keywords and aims to shield the user from the underlying complexity of the conceptual model. However, this hiding of the temporal component through the keywords costs the model flexibility and the naturalness of the relational model is lost. Thus the motivation for a new model is to develop a simplistic conceptual model for temporal databases and maintain the flavour and transparency of the relational model. The novel *chronon*¹ model extends the relational model through a time attribute. This attribute is allowed to be used in ways similar to

¹ A chronon is the smallest indivisible time unit [18].

any other attribute. Through examples we show the simplicity and flexibility achieved. Further, for the chronon model to be successfully implemented, we need an efficient physical model and this is borrowed from the concepts developed in SQL:2003 and XML. Thus the objectives of this paper can be summarized as follows:

- To propose a new chronon based conceptual (relational) model for temporal databases; the model uses tuple timestamping paradigm which enables the user to access the time attribute as any other attribute in SQL- only a single additional construct, MERGE BY, is required to solve the coalescing issues in temporal projection;
- To provide a corresponding physical model for efficient implementation of the conceptual model; the physical model is based on the attribute versioning scheme used in SQL:2003 and XML;
- To describe the mapping from the conceptual model to the physical model through appropriate examples of SQL Data definition and Data Manipulation constructs;
- To evaluate the model in comparison to TSQL2, SQL:2003 and XML, and discuss its equivalence to Allen's Algebra.

The paper is structured as follows. We start with the related work in Section 2, highlighting the problems and lack of temporal support in standard SQL. We briefly explain the salient features of the most significant models proposed for temporal databases viz. TSQL2, XML and SQL:2003. Section 3 develops the chronon based relational model and provides its evaluation. We also provide the equivalence to Allen's Algebra and a mapping from the conceptual to the physical model in this section. We talk about the scope for further work and conclude in Section 4.

2. Related Work, Concepts and Issues

Work on temporal semantics and databases have been on-going for over two decades now [3] and various models have been proposed to capture the essence of time varying information. It has been contended that most practical applications tend to have at least a few aspects which are temporal in nature. Pure non temporal databases exist only in textbooks! ([19], [14, pp.744]).

The conceptual modeling of temporal schema using ER diagramming received attention early and a complete review of the various constructs can be found in [9]. Many conceptual models have been proposed, however, no current standard exists [20]. The TSQL2 model, proposed by Richard Snodgrass, was adapted [18] and pushed to be included in the SQL3 standard but was cancelled in 2001 [15]. Subsequently, SQL:2003 has introduced features that provide indirect temporal support [6]. Substantial research has also looked at the viability of XML as a temporal database [4, 5, 7]. The commercial products based on temporal models proposed are very limited in number [20] with the most notable commercial support for temporal

queries coming from Oracle9i and Oracle10g. Here we review each of these significant areas of the temporal database research.

2.1 Tuple and Attribute Versioning

The most prominent way of capturing time varying information in a relational model is to add two columns denoting the start and end times. These time attributes can be of two types: valid time and transaction time. Valid time refers to the time when the fact was or will be true in the real world and transaction time refers to when the fact was recorded in the database. The semantics of these attributes have been studied in [3]. It has been shown that valid time and transaction time are orthogonal and through this, temporal specialization has been defined as the relationship between transaction time and valid time. Temporal databases, in general, have to support both retroactive and predictive applications. We define retroactive as the situation where the data has been valid before the actual storage and predictive as when we record, before the data becomes valid. In terms of the temporal semantics, retroactive has valid time less than the transaction time and predictive has valid time greater than transaction time.

In the traditional relational model, a fact is denoted by a tuple, however, in temporal models a fact can be denoted by a tuple or by an attribute. Its semantics are determined by the model [3]. The definition of a fact leads to two basic types of data storage: the tuple versioning and the attribute versioning. We may wish to define the entire tuple as a fact and thus obtain a tuple versioning model. We may, instead, decide to store the valid times of each attributes and this constitutes the attribute versioning model [1, pp. 608,609]. Adapting the representation from [1], the tuple versioning and attribute versioning models are depicted below. Note that the models depict only valid time. A similar semantic will be incorporated for transaction time. The keyword *now* can be thought of as *today's date*. It signifies that the particular tuple/attribute is current. The complete semantics of now can be found in [3].

Table 1. Tuple Versioning: table emp

| emp_ID | salary | dept | valid_start | valid_end |
|--------|--------|------|-------------|-----------|
| 01 | 10000 | 10 | 1-Jan-06 | 31-Dec-06 |
| 01 | 20000 | 10 | 1-Jan-07 | now |

Table 2. Attribute Versioning: table emp_attr

| emp_ID | salary | | dept | |
|--------|-------------|-----------|-------------|-----------|
| 01 | 10000 | | 10 | |
| | valid_start | valid_end | valid_start | valid_end |
| | 1-Jan-06 | 31-Dec-06 | 1-Jan-06 | now |
| | 20000 | | | |
| | valid_start | valid_end | | |
| | 1-Jan-07 | now | | |

In attempting to capture time varying data in a tuple versioned format, several issues arise and these have been briefly introduced below. The complete details can be obtained from [3].

2.2 Issues in Temporal Databases

Key Constraint: The primary key of Table 1 is a combination of (emp_ID,valid_start,valid_end). However, in a standard relational table, two tuples with the same emp_ID and valid_start, but with a different valid_end can be inserted. This conceptually means that the particular employee has different values for the same attributes simultaneously. Obviously this is an error and is the issue of *key constraint*. To avoid this error, the system needs to ensure that the new record of the same entity does not overlap in time.

Referential Constraint: Similarly, in trying to ensure referential integrity, the system needs to check that the referenced entity is a superset of the new tuple, in terms of time. Note that in the case of the key constraint, we need to ensure that the tuples *do not* overlap in time, while in the case of referential constraint, we *need to* ensure that the tuples overlap in time.

Temporal Projection: Notice the attribute, dept, of Table 1. A query asking to display the dept and the time associated, will come up with the two rows, when actually the result must be merged along time. This is the well know issue of *temporal projection*. See example below.

```
SELECT emp_id, dept,
       valid_start, valid_end
FROM emp
WHERE empid=01
```

Fig. 1(a). Query

| emp_ID | valid_start | valid_end |
|--------|-------------|-----------|
| 01 | 1-Jan-06 | 31-Dec-06 |
| 01 | 1-Jan-07 | now |

Fig. 1(b). Result of Query

Note that there is no new information presented in displaying the results in two rows. The result of the query should be coalesced along the time axis by having valid_start='1-Jan-06' and valid_end = 'now' in a single tuple.

Updates: Consider a query to update a record of Table 1. The operations involved would be to first close the record (update the valid_end) and create a new record. A single logical update is now an expanded set of instructions and is referred to as the issue of *updates* in temporal databases.

Joins: Queries written with the tuple based relational model have been found to be exceptionally difficult to write when *joins* between tables are involved. This problem is further aggravated in the attribute versioning scheme. (see section 3.2.)

The issues in trying to capture temporal information in a tuple versioned relational model are summarized below.

| Issue | Comment |
|-----------------------|---------------------------------|
| Key Constraint | Must not be Overlapping in time |
| Referential Integrity | Must be Overlapping time |
| Temporal Projection | Coalescing |
| Updates | Multiple operations |
| Joins | Query Complexity |

Fig. 2. Issues in capturing temporal data

2.3 TSQL2

TSQL2, a temporal extension to the SQL-92 standard, results from an initiative that aimed to consolidate more than 15 years of active research in temporal data models and query languages. A preliminary design was completed in 1994 and a revised and definite version was published in September 1994 [16]. The TSQL2 language supports both valid and transaction time queries, follows tuple versioning and supports coalesced relations [1]. The TSQL2 language was adapted, in 1995, to be incorporated as a standard into the SQL3 ANSI/ISO specification [18] however, the inclusion fell through.

TSQL2 indicates the valid time support through a new keyword “AS VALIDTIME PERIOD(DATE)”. The integrity constraints are enforced through two new keywords “VALIDTIME PRIMARY KEY” and “VALIDTIME REFERENCES”. TSQL2 looks to make query writing simple through the introduction of these new keywords. It thus shields the user from the underlying tuple versioned conceptual model. This, however, limits the flexibility in the queries that TSQL2 can handle and gives a black box flavour to the model. The complete specification of TSQL2 can be obtained from [16]. The TSQL2 language has been termed as being comprehensive and technically elegant [6]. However, due to disagreements within ISO as to where the temporal support in SQL should go, the inclusion was abandoned towards the end of 2001 [20].

2.4 Attribute versioning in XML

XML stands for eXtensible Markup Language and was developed by the World Wide Web Consortium (W3C) to deal with the shortcomings of HTML [4,26]. XML was made a W3C standard in 1998 [27]. The biggest difference between XML and HTML was the ability to create user-defined tags in XML. Although XML started as a way to mark-up language (i.e. denote which text is bold, which is a new paragraph etc) it was soon recognized as a way to define the structure of data and provide a mechanism for data exchange [4,23]. The structure of an XML document is defined in a DTD (Document Type Definition). Table 1 would look as shown below. Note that XML implements the attribute versioning scheme.

```
<emp>
  <emp_ID >01</emp_ID>
  <salary valid_start='1-Jan-06' valid_end='31-Dec-06'> 10000</salary>
  <salary valid_start='1-Jan-07' valid_end='now' > 20000</salary>
  <dept valid_start='1-Jan-06' valid_end='now' > 10</dept>
</emp>
```

Fig. 3. Temporal data in XML format

Query languages have been developed to extract subsets of an XML document. XQuery and XPath have been made standards by W3C [25, 24]. The ability in XML to create user-defined tags coupled with the query language has made it possible to store temporal information. The performance of XML against the relational model for use as a database has been made in [5]. Although the storage requirement in XML was found to be superior, due to the ASCII form of storage, the query language efficiency was found lacking. In XML, coalescing is not a problem due to the attribute versioning scheme. However, XML provides no additional features for temporal support. It thus suffers from the problems of *key constraint*, *referential integrity constraint*, *updates and joins*. Further, queries often become exceptionally difficult to write and tend to require embedded programming.

XML databases can be classified as follows [23]:

- Native databases : those which store the data in XML (ASCII) format
- XML-enabled: those which interface with the user through the XML model and Xquery but store the data physically in a different format (can be relational). The conversion between the physical model and XML is done by the DBMS
- XML-middleware: A stand alone application that converts data stored in relational model to XML and vice versa.

The performance of each of these approaches in comparison with the relational model is given in [7]. Although the storage requirement in XML was found to be superior, the query language efficiency of SQL was much better

2.5 Nested Relations in SQL:2003

The SQL:2003 standard was the fifth revision and introduced XML related features and nested constructs among others. The nested relations capability has been used to achieve a temporally grouped representation as discussed in [6]. Nested relations provide the ability to implement attribute versioning, a structure similar to the XML model. The Oracle database supported the nested structures even before the introduction of the SQL:2003 standard [22]. Through the use of these nested structures, SQL:2003 has only achieved the attribute versioning methodology. The queries by themselves have no support for temporal data and due to the attribute versioning scheme, suffer from the same problems as XML.

3. The Chronon based temporal model

Looking at the various models spoken in this paper, we readily identify that the models, at their core, are either of tuple versioning format or attribute versioning. TSQL2 follows the tuple versioning approach while SQL:2003 and XML follow attribute versioning. TSQL2 has single mindedly focused on simplifying the SQL statements by introducing new constructs. It has achieved this to a large extent, however, we content the problem lies not in just the lack of features in SQL, but in the conceptualization of the tuple versioning approach. A user writes SQL queries based on the conceptual model. For example, for a non temporal relational model, the conceptualization of a table of values is straight forward. However, the inclusion of valid and transaction time introduces complexities. TSQL2, by masking the user from the underlying time attributes, has moved the model away from the natural transparence of the relational model. This makes the model difficult to comprehend and has inhibited its flexibility. The temporal attributes, for example, valid times, are not directly available to the user. Further, TSQL2 has not provided an explicit physical model. Thus the motivation for a new model is twofold:

- Simpler conceptualization of the data model to ease the writing of SQL queries and maintain the transparence of the relational model
- Corresponding efficient physical model and the ease of mapping the physical to conceptual relational model.

The two aspects, namely, the *chronon based conceptual model* and the *physical model* are spoken about next. The user is concerned only with the conceptual model and writes queries based on this. The DBMS, however, stores data according to the physical model and works based on the mapping between the physical and the conceptual model

3.1 The proposed conceptual model

Inspired by the representation of time in [15], the conceptual model is made to comprise of a single time attribute for an entire relation. For example, a relation $R(A_1, A_2, A_3)$ is instead written as $R(A_1, A_2, A_3, T)$. Here T holds the time instant the relation was *saved*. Conceptually, a relation is saved every instant. Here to define instant, we borrow the definition of chronon from [18]. A *chronon is the smallest indivisible time unit*. A tuple of a relation would then look as follows:

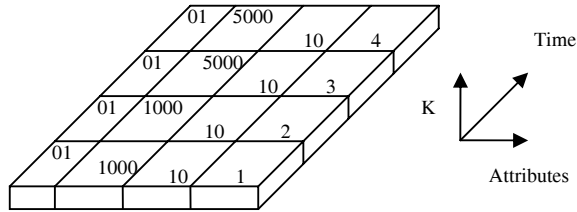


Fig. 4. The chronon based conceptual (relational) model

The figure shows the state of the tuple for every time instant. Here $T_2 - T_1 = 1$ chronon. We make the assumption that the key of the relation is time invariant. This assumption is not a serious concern since in most real life instances, the key is not changed and is unique. To include time-variant keys, we can introduce surrogates as discussed in [9]. We have also limited the time attribute to valid time. Transaction time can be modeled in a similar way. To understand the model, consider we have a relation, $emp(empID, salary, deptID, time)$. The first tuple was added on 1-Jan-07. Thus the table would look as follows:

Table 3. emp table on 1-Jan-07

| empID | salary | deptID | time |
|-------|--------|--------|----------|
| 01 | 1000 | 10 | 1-Jan-07 |

Table 4. emp table on 2-Jan-07

| empID | salary | deptID | time |
|-------|--------|--------|----------|
| 01 | 1000 | 10 | 1-Jan-07 |
| 01 | 1000 | 10 | 2-Jan-07 |

For simplicity, let's assume the granularity of time is a day, i.e. the user believes the attributes change at the end of a day. On 2-Jan-07, the table would look as in Table 4. There is no change in the attributes except time. The second tuple is telling us that the empID 01 continues to have a salary of 1000 and belongs to department 10 on 2-Jan-07. This second tuple is not inserted explicitly by the user, but is saved (conceptually) by the DBMS. This can be thought of as a virtual save. This automatic save will continue until the tuple is deleted. The user believes the daily save occurs. The physical model instead stores data in an efficient manner. Now, to appreciate the simplicity of the model, consider the different problems mentioned in section 2. To start with, the user creates a table by mentioning the time attribute explicitly, i.e. the following command is issued: `CREATE TABLE emp(empID, salary, deptID, time)`

Key Constraint: Notice that the user has mentioned time to be part of the key. The need for ensuring that the key is not overlapping in time is avoided. The user can conveniently state time as part of the key and not be aware of any new construct.

Referential Constraint: The user simply includes the time attributes in the syntax. This is the standard SQL construct without any need for enhancements: `FOREIGN KEY(deptID, time) REFERENCES mgr(deptID, time)`

Temporal Projection: Here, the need for a new construct is felt which should coalesce the values along time. For example, assuming today is 6-Jan-07, a `SELECT * FROM emp` would result in figure 4(a). Thus we propose a construct MERGE BY, with syntax as shown in Fig 4(b).

| empID | salary | deptID | time |
|-------|--------|--------|----------|
| 01 | 1000 | 10 | 1-Jan-07 |
| 01 | 1000 | 10 | 2-Jan-07 |
| 01 | 1000 | 10 | 3-Jan-07 |
| 01 | 2000 | 10 | 4-Jan-07 |
| 02 | 1000 | 20 | 4-Jan-07 |
| 01 | 2000 | 10 | 5-Jan-07 |
| 02 | 1000 | 20 | 5-Jan-07 |
| 01 | 2000 | 10 | 6-Jan-07 |
| 02 | 1000 | 10 | 6-Jan-07 |

Fig. 5(a). Query Output

```

SELECT attribute-and-function-list
FROM table-list
[ WHERE row-condition ]
[ GROUP BY grouping-attribute(s) ]
[ HAVING group-condition ]
[ MERGE BY time-column [WITH
time-condition] ]
[ ORDER BY attribute-list ]

```

Fig. 5(b). MERGE BY Syntax

The query of Fig.5(a), with MERGE BY, will result in the output as shown in Fig.5(b) after coalescing the tuples along time. Note that now represents today's date.

| empID | salary | deptID | time |
|-------|--------|--------|----------------------|
| 01 | 1000 | 10 | 1-Jan-07 to 3-Jan-07 |
| 01 | 2000 | 10 | 4-Jan-07 to now |
| 02 | 1000 | 20 | 4-Jan-07 to 5-Jan-07 |
| 02 | 1000 | 10 | 6-Jan-07 to now |

Fig. 6(a). Query

Fig. 6(b). Output of Query 5(a)

The MERGE BY clause will work only on the time attribute and can include a condition on this time attribute using a WITH clause. Its operation can be thought of as being similar to ORDER BY, where the rows of the result set are ordered based on values of other attributes and is then arranged in order of time. MERGE BY, then, for every contiguous time periods, reduces the number of rows to be displayed by showing time in the "from" and "to" format. We could think of replacing "from" with MIN(time) and "to" with MAX(time), however, this is possible only if MIN(time) and MAX(time) are contiguous. The WITH clause will work only on the time attribute of the result set passed by MERGE BY. This ensures the conditions test over contiguous time. The MERGE BY and ORDER BY operators can exist simultaneously with the ORDER BY working on the result set of MERGE BY. We have shown below different cases of usage with the GROUP BY, MERGE BY, ORDER BY and HAVING clauses to ensure there is no ambiguity when these clauses are used both independently and in conjunction with each other.

CASE 1: MERGE BY occurs independently

The columns to be displayed in the select clause are arranged and merged along time as seen in Fig. 5(b).

CASE 2: MERGE BY and WITH

The result of MERGE BY can be tested for conditions on time using the WITH clause as seen below:

```
SELECT empID, time FROM emp
MERGE BY time WITH
time >= '4-Jan-07'
```

Fig. 7(a). Query

| empID | time |
|-------|-----------------|
| 02 | 4-Jan-07 to now |

Fig. 7(b). Query Output

CASE 3: MERGE BY and GROUP BY

The GROUP BY clause with the standard definition can work on the time attribute in similar ways as any other attribute. The SELECT clause can include the aggregate operators linked with GROUP BY. The inclusion of MERGE BY will now effect in arranging the rows with the result of the aggregate operator included.

```
SELECT deptID, COUNT(*),
time FROM emp GROUP BY
deptID, time
MERGE BY time
```

Fig. 8(a). Query

| deptID | COUNT | time |
|--------|-------|----------------------|
| 10 | 1 | 1-Jan-07 to 5-Jan-07 |
| 20 | 1 | 4-Jan-07 to 5-Jan-07 |
| 10 | 2 | 6-Jan-07 to now |

Fig. 8(b). Query Output

CASE 4: GROUP BY and HAVING

The HAVING clause can include the columns specified in the GROUP BY clause as in the standard case. Note that this means, the inclusion of the time attribute is legal. See example below. Notice in particular, the difference of this output from the

```
SELECT empID, time FROM emp
GROUP BY (empID,time)
HAVING ( time >= '4-Jan-07' )
//notice we could have achieved the same
// using only a WHERE clause
```

Fig. 9(a). Query

| empID | time |
|-------|----------|
| 01 | 4-Jan-07 |
| 02 | 4-Jan-07 |
| 01 | 5-Jan-07 |
| 02 | 5-Jan-07 |
| 01 | 6-Jan-07 |
| 02 | 6-Jan-07 |

Fig. 9(b). Query Output

CASE 5: MERGE BY and ORDER BY

The ORDER BY clause works in the standard way and can include the time attribute. Notice that arranging according to time is equivalent to arranging along the “from” time.

```
SELECT empID, time FROM emp
MERGE BY time
ORDER BY time DESC
```

Fig. 10(a). Query

| empID | time |
|-------|-----------------|
| 02 | 4-Jan-07 to now |
| 01 | 1-Jan-07 to now |

Fig. 10(b). Query Output

To summarise, we have not altered the default action of any standard SQL statement. A single new clause MERGE BY is introduced which behaves in ways

similar to ORDER BY. It can have a condition to be tested on the time attribute through a WITH clause.

Updates: The user simply updates the tuple whose time value is in the range. For example, *UPDATE emp SET salary = 20000 WHERE time >= '1-Jan-07'*. The update is now a single logical statement. Multiple operations are avoided. The comparison with TSQL2 brings out the efficiency in the chronon based model.

Joins: The treatment of the time attribute is in the same manner as any other attribute and thus the joins follow the standard SQL syntax.

| NEED FOR A NEW CONSTRUCT | | |
|--------------------------|-------|---------------|
| PARAMETER | TSQL2 | CHRONON MODEL |
| Key | YES | NO |
| Foreign Key | YES | NO |
| Temporal Projection | YES | YES |
| Joins | YES | NO |
| Updates | YES | NO |

Fig. 11. Comparison of the new constructs needed in TSQL2 and the chronon model

3.2 Evaluation of the model

Comparison with TSQL2 and SQL:2003

We now run through the same examples provided by TSQL2 in [18] and bring out the conceptual simplicity of the chronon model through the comparison between TSQL2 and SQL:2003. The SQL:2003 statements have been tested on ORACLE 10g EXPRESS EDITION.

| | |
|--|---|
| <u>TABLES</u> | employee(eno,name,street,city) salary(eno,amount) |
| //Table creation in chronon model has been explained. SQL:2003 has nested tables of the same name as the attributes with time labeled valid_start and valid_end. | |
| <u>QUERY 1:</u> List those employees who have no salary and the associated time periods. | |
| <u>TSQL2</u> | VALIDTIME SELECT name FROM employee WHERE eno NOT IN (SELECT eno FROM salary) |
| <u>SQL:2003</u> | SELECT DISTINCT en.name FROM employee e, TABLE(name) en WHERE (e.eno IN (SELECT eno FROM salary, TABLE(amount)a1 WHERE a1.valid_end <> '31-dec-99' AND eno NOT IN (SELECT eno FROM salary, TABLE(amount) a2 WHERE a2.valid_start=a1.valid_end+1))) OR (e.eno NOT IN (SELECT eno FROM salary)) |

| | |
|-----------------------------|---|
| <u>CHRONON MODEL</u> | SELECT name,time FROM employee e WHERE eno NOT IN (SELECT eno FROM salary WHERE e.time=time) MERGE BY time |
| <u>NOTES</u> | <ul style="list-style-type: none"> • The join along time is explicit in the chronon model • In SQL:2003, we were unable to write a query which would obtain the time period of no-salary as well. |

| | |
|---|--|
| QUERY 2: List the number of employees with salary > 5000 in each city with the time periods. | |
| <u>TSQL2</u> | VALIDTIME SELECT city,count(*) FROM employee e, salary s WHERE e.eno=s.eno AND salary > 5000 GROUP BY city |
| <u>SQL:2003</u> | SELECT c.city, count(*) FROM employee e, TABLE (city) c, salary s, TABLE(amount) a WHERE e.eno=s.eno AND a.amount > 5000 AND a.valid_start <= c.valid_start AND a.valid_end > c.valid_end GROUP BY c.city; |
| <u>CHRONON MODEL</u> | SELECT city,count(*),time FROM employee e, salary s WHERE e.eno = s.eno AND e.time = s.time AND amount > 5000 GROUP BY (city,time) MERGE BY time |
| <u>NOTES</u> | <ul style="list-style-type: none"> • The MERGE BY in the chronon model includes the COUNT(*) with the city while displaying the time in the “from” and “to” format • In SQL:2003, although the answer would be correct, it misses out if finer granularity than that of the city time period exists in amount. |

| | |
|--|--|
| QUERY 3: Update salary of “Therese” to 6000 for year '06. | |
| <u>TSQL2</u> | VALIDTIME PERIOD '[1-Jan-06 - 31-Dec-06]' UPDATE salary SET amount=6000 WHERE eno IN (SELECT eno FROM salary WHERE name =‘Therese’) |
| <u>SQL:2003</u> | // we couldn't think of any other way than an embedded program. |
| <u>CHRONON MODEL</u> | UPDATE salary SET amount=6000 WHERE eno IN (SELECT eno FROM salary WHERE name=‘Therese’ AND time >= ‘1-Jan-06’ AND time <= ‘31-Dec-06’) AND time >= ‘1-Jan-06’ AND time <= ‘31-Dec-06’ |
| <u>NOTES</u> | <ul style="list-style-type: none"> • TSQL2 makes the assumption of consistent time across the two sub-queries. • The explicit mention of “time” in the chronon model makes the query much more readable and flexible |

| | |
|--|--|
| QUERY 4: Determine who was given salary raises. | |
| <u>TSQL2</u> | NONSEQUENCED VALIDTIME SELECT name FROM employee e, salary s1, salary s2 WHERE e.eno = s1.eno AND e.eno = s2.eno AND s1.amount < s2.amount AND VALIDTIME(s1) MEETS VALIDTIME(s2) |
| <u>SQL:2003</u> | SELECT eno FROM salary, TABLE(amount) a1 WHERE a1.amount > (SELECT a2.amount FROM salary, TABLE(amount) a2 WHERE a2.valid_end = a1.valid_start - 1 AND a1.valid_start > a2.valid_start) |

| | |
|----------------------|---|
| CHRONON MODEL | SELECT name FROM employee e WHERE eno IN (SELECT eno FROM salary s1 WHERE amount > (SELECT amount FROM salary s2 WHERE s1.eno=s2.eno AND s2.time = s1.time -1)) |
| NOTES | <ul style="list-style-type: none"> • The introduction of NONSEQUENCED in TSQL2 has made the query much more difficult to understand • Constructs like VALIDTIME(s1) MEETS VALIDTIME(s2) in TSQL2 are not needed in the chronon model. • SQL:2003 finds it relatively easier when the associated time periods are not to be displayed |

Fig. 12. Queries comparing TSQL2, SQL:2003 and chronon model

The chronon model although using only an additional construct, is much more expressive. The flexibility can be further gauged by the ability to count the number of days using COUNT(time), or to determine the first or the last day using MIN(time) or MAX(time) respectively.

Equivalence to Allen's Algebra

| Operator | Equivalent syntax |
|---|--|
| [valid_start,valid_end] INCLUDES [t1,t2] | MERGE BY time WITH (t1 >= time AND t2 <= time) |
| [valid_start,valid_end] INCLUDED_IN [t1,t2] | MERGE BY time WITH (t1 <= time AND t2 >= time) |
| [valid_start,valid_end] OVERLAPS [t1,t2] | MERGE BY time WITH (t1 <= time AND t2 >= time) |
| [valid_start,valid_end] BEFORE [t1,t2] | t1 >= time // MERGE BY not needed |
| [valid_start,valid_end] AFTER [t1,t2] | t2 <= time // MERGE BY not needed |
| [valid_start,valid_end] MEETS_BEFORE [t1,t2] | time=t1-1 // MERGE BY not needed |
| [valid_start,valid_end] MEETS_AFTER [t1,t2] | time=t2+1 // MERGE BY not needed |

Fig. 13. Allen's Algebra and its chronon model equivalent

Allen's algebra provides operators typically needed in temporal databases. We have provided, the chronon model equivalent for some of the operators [14].

3.3 Mapping of Chronon model to the Physical Model

The physical model exploits the attribute versioning scheme. This ensures optimal disk usage. The mapping is provided below.

CREATE TABLE: The standard constructs to be followed. A new parameter “chronon” introduced to indicate the valid time temporal support. Transaction time can be specified using the keyword “transact chronon”. Internally, attribute versioning is made.

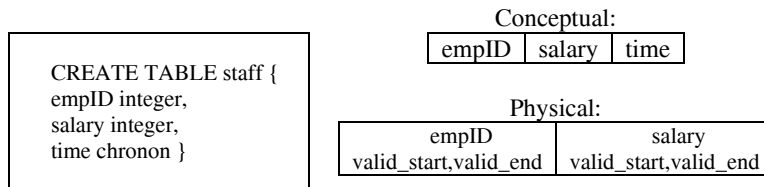


Fig. 14. Conceptual to Physical mapping for table creation

INSERT: We first analyse the different possibilities that an INSERT statement would have to handle and the plot is shown in Fig. 14. It shows the relationship between the value of time supplied by the user and today’s date, indicated by “now”. Case 1 indicates, the user is recording retroactive data (i.e. events that have occurred before today.) Case 2 indicates, the user is looking to record data that has occurred prior to today and is still current. This corresponds to a value of “now” for “valid_end” and is shown as an arrow in the plot. In the chronon model, the arrow can be thought to indicate the expectation of the user for the daily saves to occur from that point in time, onwards. Notice that case 2 is the most common INSERT scenario and this forms the default behaviour of the chronon model INSERT statement (see Fig. 15). Case 1 can be achieved by deleting the tuples not needed from case 2 (e.g. DELETE FROM staff WHERE time >5-Jan-07). Similarly, cases 3 and 5 can be realized.

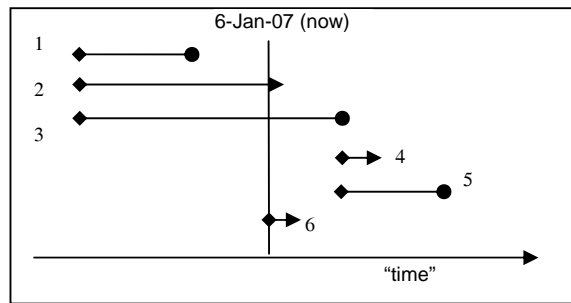


Fig. 15. The plot of the different cases of INSERT. The diamonds at the start of each timeline indicates “valid_start”. The dots/arrows indicate “valid_end”. An arrow indicates the value is “now”

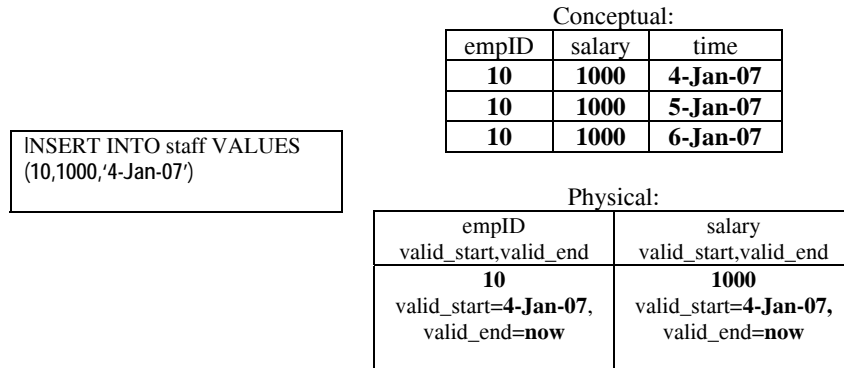


Fig. 16. The mapping of the conceptual to the physical model with today's date being 6-Jan-07

DELETE/UPDATE: Similar to the INSERT case, any of the six scenarios of Fig.14 is possible. Cases 1 and 2 occur when the need is to delete or update retroactive data. Cases 4, 5 and 6 occur when predictive data is to be modified. Such cases occur when data valid in the future have been inserted (INSERT scenarios 4, 5 and 6 of Fig. 14) and now there is a need to alter them. Case 3 occurs when the modification time is both retroactive and predictive. To delete a particular tuple or a set of tuples, the standard SQL syntax is used (e.g. DELETE FROM staff WHERE time > 5-Jan-07 AND time < 10-Jan-07). The UPDATE syntax is similar (UPDATE staff SET salary=2000 WHERE time=5-Jan-07). The physical model is updated with the appropriate change in the value of the attributes and “valid_start” and/or “valid_end”. Thus no extra constructs or added functionalities are needed. A DELETE/UPDATE can also potentially split a contiguous block into two. This transpires to two tuples in the physical model. Thus the logical DELETE/UPDATE in the chronon model remains a single operation while the multiple steps needed in the physical model are caught by the mapping.

4. Conclusion

In this paper we have developed the chronon based conceptual relational model for temporal databases and showed the simplicity and flexibility achieved using only a single additional construct, MERGE BY. We have evaluated the chronon model against the various pitfalls of using the standard relational model for temporal needs, compared with the TSQL2 model and provided the equivalence to Allen's Algebra. We have also proposed a physical model based on the current trends of attribute versioning and provided the mapping from the physical to the conceptual model.

There are a few issues to be addressed in the future. We need to analyse the complexities introduced due to the inclusion of transaction time support in the chronon model. We are also particularly interested in testing the model in a Spatio-Temporal setting where the need is to store not only *what* is important to an

application but also *when* and *where* [8]. The simplicity achieved in the temporal domain gives impetus to adapt the model in these new and upcoming areas.

References

- [1] A. Dekhtyar, "TSQL2 in a nutshell", www.cs.uky.edu/~dekhtyar/685/tsql.alex.ps, Sept 2000
- [2] C.S. Jensen et al, "A consensus test suite of temporal database queries", <ftp://ftp.cs.arizona.edu/tsql/doc/testSuite.ps>, 1993
- [3] C.S. Jensen and R.T Snodgrass, "Semantics of Time Varying Information", Information Systems, Vol 19, No. 4, 1994
- [4] D. Obasanjo, "An Exploration of XML in Database Management Systems", www.25hoursaday.com/StoringAndQueryingXML.html, 2001
- [5] F. Wang and C. Zaniolo, "XBiT: An XML-based Bitemporal Data Model", Proceedings of the 23rd International Conference on Conceptual Modelling(ER '04), Shanghai, Nov 2004
- [6] F. Wang, C. Zaniolo and X. Zhou, "Temporal XML? SQL Strikes Back!", 12th International Symposium on Temporal Representation and Reasoning (TIME), June 2005
- [7] F. Wang and C. Zaniolo, "An XML-Based Approach to Publishing and Querying the History of Databases", Internet and Web Information Systems, Vol 8, No 3, Sept 2005
- [8] G. Allen, A. Bajaj, V. Khatri, S. Ram and K. Siau, "Advances in Data Modeling Research", Communications of the Association of Information Systems, Vol 17, 2006
- [9] H. Gregerson and C.S. Jensen, "Temporal Entity-Relationship Models – A Survey", IEEE transactions on knowledge and Data Engineering, Vol 11, No 3, 1999
- [10] J. Clifford, C. Dyreson, T. Isakowitz, C.S. Jensen and R.T. Snodgrass, "On the Semantics of "now" in Databases", ACM transactions on database systems, Vol 22, No 2, June 1997
- [11] M.H. Bohlen, R.T. Snodgrass and M.T. Soo, "Coalescing in Temporal Databases", VLDB, 1996
- [12] Oracle9i Flashback Query, Oracle white paper, 2002
- [13] Oracle 10g Workspace manager overview, Oracle white paper, 2005
- [14] R. Elmasri and S.B. Navathe, Fundamentals of Database Systems, Third Edition, Pearson Education, 2003
- [15] R.T. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases", ACM SIGMOD International Conference on Management of Data, 1985
- [16] R.T. Snodgrass, The TSQL2 Temporal Query Language, Springer publication, 1995
- [17] R.T. Snodgrass, "Managing temporal data – a five part series", Database programming and design, TimeCenter technical report 1998
- [18] R. T. Snodgrass, M. H. Bohlen, C. S. Jensen, and A. Steiner, "Transitioning temporal support in TSQL2 to SQL3, Temporal Databases: Research and Practice, 1998; 150-194
- [19] R. T. Snodgrass, Developing Time Oriented Database Applications in SQL, Morgan Kaufmann Publishers, San Francisco, California, 2000
- [20] R.T. Snodgrass, Official website in University of Arizona, <http://www.cs.arizona.edu/~rts/sql3.html>
- [21] S. Abiteboul, S.R. Hull and V. Vianu, Foundations of Databases, Addison-Wesley Publishing Company, 1995
- [22] SQL:2003 standard support in Oracle Database 10g, Oracle white paper, 2003
- [23] W3C, <http://www.w3.org/XML/>
- [24] W3C, <http://www.w3.org/TR/xpath>
- [25] W3C, <http://www.w3.org/TR/xquery>
- [26] Wikipedia – HTML, <http://en.wikipedia.org/wiki/HTML>
- [27] Wikipedia – XML : <http://en.wikipedia.org/wiki/XML>
- [28] Wikipedia – XML Database: http://en.wikipedia.org/wiki/XML_database