Relationship between time intervals in XML

COMP630H Fall 2003

Course Project Report

Tong Kwan-Ho

Law Ho-Man Janelle

{baek@ust.hk; janelle@ust.hk}

Content

Abstract

Point-based time data can be represented in database easily. However, it is more complex to represent time interval data in database. Time interval data have already been represented by object-relational DMBS, but not in semi-structured database like XML. In this project, we proposed a new data structure to represent time interval data in XML.

Introduction

Interval means a certain range. It is applicable in many areas, such as time, voltage, temperature, length, etc. Although the database technology have been developed for many years, it is surprising that no commonly used DBMS is supporting interval data, in the sense of storing interval data as an atomic value, and provides functions dealing with interval data.

Time interval data is even more complex than other interval data. Since time interval data may involve multi-dimensions, like if we want to store for the working hours, which is from Monday to Friday, 08:00a.m. to 5:00p.m. This kind of data does have numeric meaning, but it cannot be process in any database query so far we know.

XML (Extensible Markup Language) is the most popular semi-structured database technique nowadays. There are several advantages provided to store semi-structured data. So, by utilizing the advantages of XML, in this project, we are going to propose a time interval structured to store time interval data like we have discussed. Moreover, we are going to suggest some functions that will be commonly used by this structure.

Motivation

By using the current DBMS technology (or functions provided), to store time related problem, we have only 1 choice, which is to store the exact timing as an atomic value. Some DBMS provide us with DateTime structure, that is, to store an exact date time in one atomic value. For example, we can store 2003/12/25, 09:00am in one field. Inside the DBMS, this data is actually stored as a double number. The integer part of the double means the date, counting from 1900/1/1 (in MS Excel), and the decimal part declared the time within a single day.

This approach is easy to handle inside the DBMS, by transforming the datatime to a double value, it can easily determine which time data is before which time data, by comparing the numeric value of them. However, this approach cannot meet our requirements.

In many applications, we may have to store a certain period of time, like from 08:00 to 17:00. By using the current functions provided, we have to create two fields to store this data, let's say they are "fromtime" and "totime".

For example, Staff A works from 08:00 – 17:00. Staff B works from 14:00 – 22:00. How can we answer the query "what is the time range that both staff A and B are working?" The answer would be from 14:00 – 17:00 obviously. This query can be answered by human easily, but how the query language can find out the time overlapping between Staff A's working hours and Staff's B working hours.

Programmers always have to do it manually by writing program codes in order to answer the queries. Why don't we have some standard structure to store the time interval and provide programmer with some standard and commonly used functions dealing with them? The above situation can be even more complex if Staff A only works from Mon to Fri, while Staff B works from Mon to Sat.

Our goal of this project is to propose a time interval structure in XML to store and process these data. At the end, we can answer the queries like:

1. "StaffA.worktime overlap StaffB.worktime"
2. "Select * from Staff where worktime overlap "12:00 – 13:00"
3. "Will Staff A works on 29th Nov 2003 (an exact date)?"
4. "Will Staff B works on the last day of this Month?"
5. "When will be the next day that Staff A and Staff B work together?"

Moreover, the current comparison symbol (>, <, >=, <=, =, not =, and, or) are not enough to deal with time interval data. So we introduced some new predicates / functions over the proposed time structure.

IXSQL [2] has the concept of folding. Folding basically means expanding the time interval data into separated days. For example, if the time interval is “2003/12/25 – 2003/12/31”, after unfolding this data, we will have “2003/12/25, 2003/12/26, 2003/12/27, 2003/12/28, 2003/12/29, 2003/12/30, 2003/12/31”.

The problems of folding and unfolding are, first, the expanded data can be very large. Obviously, if we expand “2003/1/1 – 2013/12/31”, we will have 3,650+ records. The computational cost will be increased exponentially if we do joining over the expanded data.

Second, when we expand the data, are we going to expand it into days? Is that enough? Or should we expand it into hours, or minutes, or even seconds? We may not have idea in some cases. Therefore, the concept of folding still has some limitations.

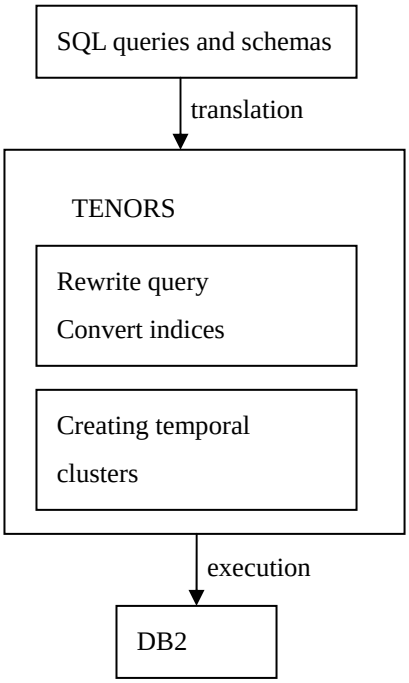
Related Work

In 1992 [1], the authors extended the relational data model to maintain the underlying view of temporal data as timepoints on a timeline, called extended SQL TIMELINESQL (TLSQL). TLSQL was based on the simple structural framework of SQL, with syntactic extensions to support operations on event and states in histories. They extended the standard relational data model in 3 ways to support TLSQL querying requirements. First, they incorporated timestamps into 2-dimensional relational table to store the temporal dimension of both instant- and interval-based data. Second, they created a set of operations on timepoints and intervals to manipulate timestamped data. Third, they modified the relational query SQL so that its underlying algebra supported the specified operations on timestamps in relational tables. 3 different types of medical data with time: instance-based, interval-based and time-invariant. Instance-based data was events (an event corresponds to a medical datum that occurs instantaneously). An event was represented by a pair of timepoints for lower and upper bounds of the closed interval of uncertainty (IOU). Interval-based data represented states and are bounded by start and stop events. The closed interval between the upper bound of the start event and the lower bound of the stop event was called the body of interval of certainty (IOC). Time-invariant data were stored as IOCs that existed for all possible timepoints. History was a relation to store instant-based, interval-based and time-invariant data type within a single type of relational table. Thus, timepoints, intervals and durations were selectable attributes and could be selected in an order FIRST, SECOND, THRID or LAST.

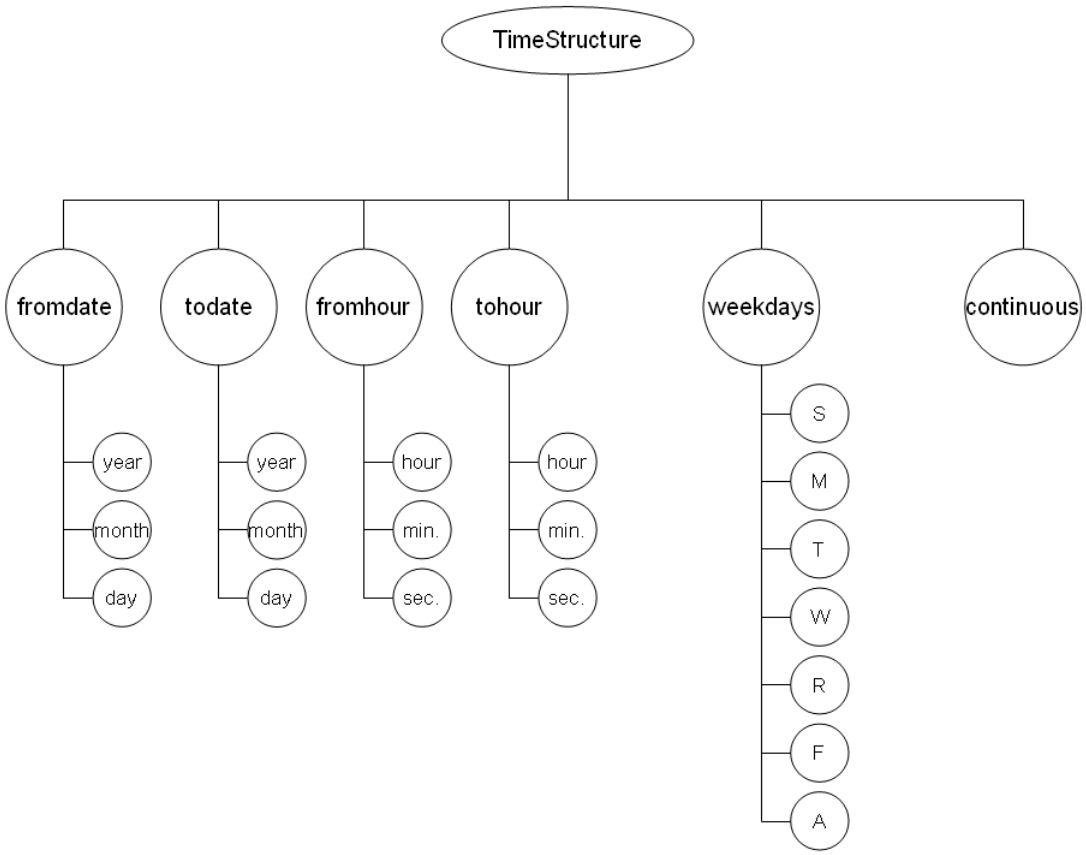
In 1997 [2], the authors proposed a management of n-dimensional data interval by extending the SQL2 to Interval Extension SQL (IXSQL). The new features of IXSQL over traditional SQL2 were direct support to a generic interval data type, incorporating new predicates, value functions, and set functions, extending the SQL manipulation operations and supporting the concept of the key of an interval relation (relations with more than one attribute of an interval type). It was possible to record two tuples whose respective values for two attributes overlap. Two intervals could be merged into one if their line segments touched or intersected. The relational algebra operations, union, difference, project and Cartesian product remained the same for interval relations. Selection had been extended and incorporated relational operators which could be applied to pairs of intervals. IXSQL aimed at handling any type of point or interval data.

In 2000 [3], the author proposed a new Relational Interval Tree (RI-tree) that was supported by any off-the-shelf relational DBMS and encapsulated by the object-relational data model. The RI-tree required $O(n/b)$ disk blocks of size b to store n intervals, $O(\log_b n)$ I/O operations for insertion or deletion, and $O(h*\log_b n + r/b)$ I/Os for an intersection query producing r results. The height h of the virtual backbone tree corresponded to the current expansion and granularity of the data space but didn't depend on n. Although the structure was space-oriented, the storage of intervals was object-driven and thus, no storage space was wasted for empty regions in the data space. It was just a dynamic expansion of the data space.

In 2003 [4], they used TENORS (Temporal Enhanced Object-Relational System) to support the SQL^T query language by using a point-based explicit temporal model. The point-based approach eliminated the need for coalescing after projection. Also, implicit temporal model could be used for overcoming the coalescing problems. TENORS used a Temporal Internal Model (TIM) that supported efficiently snapshot queries, intersection queries, and temporal join, and simplified the mapping to current O-R systems for both queries and indexing. The system architecture is shown as following:



Proposed time structure and descriptions



Here shows the time structure we proposed for representing the time interval. The meaning of each terms are shown as following:

fromdate (year, month, day) – the start date with year, month and date.
todate (year, month, day) – the stop date with year, month and date.
fromhour (hour, min, sec) – the start time with hour, minute and second.
todate (hour, min, sec) – the stop time with hour, minute and second.
weekdays (S,M,T,W,R,F,A) – Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday respectively.
continuous – a flag for stating the time continuity (to be discussed later).

The meaning of *Year*, *Month* and *Day* fields

The *year* field can show the exact year value like *2003/1/20*. It can also show an unknown year value by using wildcard *0* like *0/1/20* (it means January 20th of every year or unknown year with January 20th). By default, *year* is *0*.

The m*onth* field can show the exact month value like *2003/1/20*. It can also show an unknown month value by using wildcard *0* like 2003/0/20 (it means 20th of every month in 2003 or 20th of an unknown

month in 2003). By default, *month* is *0*.

The *day* field can show a normal date or some presentations for dates and weeks. By default, *day* is *0*. Here is the list of possible values in *day* field:

The value in *day* field	**Meaning**
0	Wildcard, can be all possibilities of the following values or unknown value.
1 – 31	A normal date from1st to 31st of a month
32-37	The week number (1-6) in a month, by subtracting 31. For example, 33 is the second week of a month because 33-31=2.
40-93	The week number (1-54) in a year, by subtracting 39. For example, 41 is the second week of a year because 41-39=2.
97	The 3rd last day of a month. For example, 2003/1/97 means January 29th 2003.
98	The 2nd last day of a month. For example, 2003/4/98 means April 29th 2003.
99	The last day of a month. For example, 2003/4/99 means April 30th 2003.

Note: No more than 6 weeks for a month because the worst case is like:

	Sunday	**Monday**	**Tuesday**	**Wednesday**	**Thursday**	**Friday**	**Saturday**
Week 1							1
Week 2	2	3	4	5	6	7	8
Week 3	9	10	11	12	13	14	15
Week 4	16	17	18	19	20	21	22
Week 5	23	24	25	26	27	28	29
Week 6	30	31					

Note: No more than 54 weeks for a year. Assume the year has 366 days, 366 / 7 = 52 weeks and 2 days. The extra 2 days may be on Saturday of the 1st week and the Sunday of the last week, like the date example of a month above. Thus, 54 weeks are the maximum for a year.

Data structure	**Unknown/default/wildcard**
fromdate (year, month, day)	fromdate(0, 0, 0)
todate(year, month, day)	todate(0, 0, 0)
fromhour(hour, min, sec)	fromhour(-1)
tohour(hour, min, sec)	tohour(-1)
weekdays(S,M,T,W,R,F,A)	weekdays(T,T,T,T,T,T,T)
continuous	F

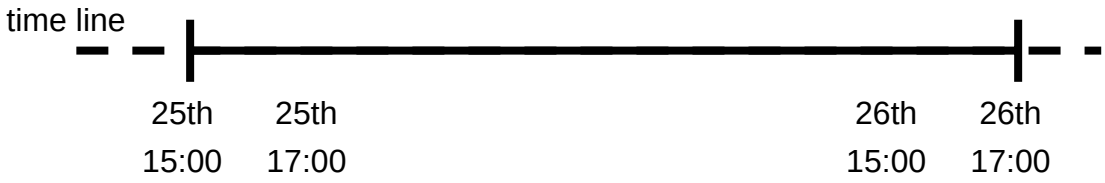
The meaning of *Continuous*

This Boolean field indicates whether the time interval is continuous or not.
For example we have the structure:

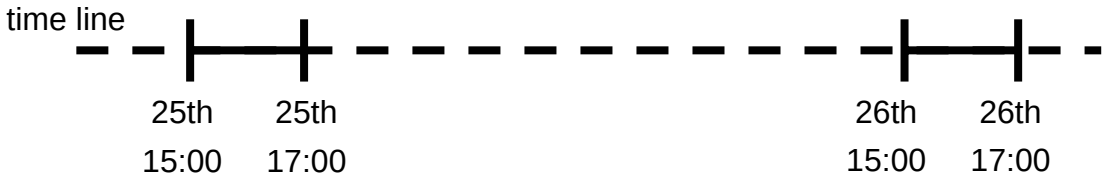
Fromdate = 2003/12/25
Todate = 2003/12/26
Fromtime = 15:00
Totime = 17:00

We will be confused that if the structure is telling every day between 25th December and 26th December, 15:00-17:00 (4 hours in total), or from 25th December 15:00 to 26th December, 17:00 (26 hours in total). So we have to use this extra field to indicate this.

If Continuous = TRUE, that means:



If Continuous = FALSE, that means:



Examples of functions and queries, with expected result

The following are some examples of how the data actually stored and their meanings.

Someone called you at…

2003 Oct 28, 16:30

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
2003	10	28	2003	10	28	16:30	16:30	T	T	T	T	T	T	T	F

John's birthday is on…

31st Nov

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
0	11	31	0	11	31	-1	-1	T	T	T	T	T	T	T	F

The privilege period is…

Every Tuesday

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
0	0	0	0	0	0	-1	-1	F	F	T	F	F	F	F	F

Mother's Day is on…

The 2nd Sunday of May, every year

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
0	5	33	0	5	33	-1	-1	T	F	F	F	F	F	F	F

Valentines' Day is on…

February 14th, every year

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
0	2	14	0	2	14	-1	-1	T	T	T	T	T	T	T	F

The salary day…

Last day of every month

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
0	0	99	0	0	99	-1	-1	T	T	T	T	T	T	T	F

The class will hold on…

Every Monday, 15:00 – 16:30

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
0	0	0	0	0	0	15:00	16:30	F	T	F	F	F	F	F	F

The COMP630H class is…

Every Tuesday, Thursday, 18:00- 19:20, from 1st Sep, 2003 – 15th Dec, 2003

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
2003	9	1	2003	12	15	18:00	19:20	F	F	T	F	T	F	F	F

First day of every month, but not Sun or Sat

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
0	0	1	0	0	1	-1	-1	F	T	T	T	T	T	F	F

Every day in February, 2003

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
2003	2	0	2003	2	0	-1	-1	T	T	T	T	T	T	T	F

Second Monday of every month

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
0	0	33	0	0	33	-1	-1	F	T	F	F	F	F	F	F

Second Monday of every month, 15:00 – 16:30

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
0	0	33	0	0	33	15:00	16:30	F	T	F	F	F	F	F	F

From May, 2001 to Sep 2003

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
2001	5	1	2003	9	99	-1	-1	T	T	T	T	T	T	T	F

From 15:00 25th Dec, 2003 to 17:00 26th Dec, 2003

From			To			From	To	Weekdays							Con
year	month	day	year	month	day	time	time	S	M	T	W	R	F	A	
2003	12	25	2003	12	26	15:00	17:00	T	T	T	T	T	T	T	T

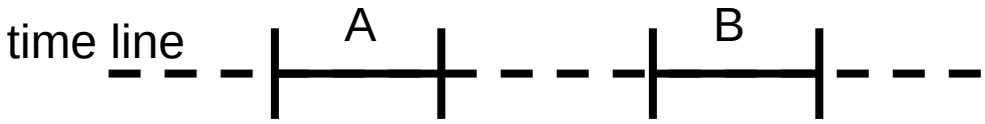
Functions to deal with the new data structure

After we have defined the time structure, we can define some functions, or predicates to process with this structure. To make it simple, we make use most of the predicates defined in IXSQL [2]. And now we select some of the most important predicates for discussion.

The following predicates take two time structures as parameter and return a Boolean value.

- Before / After

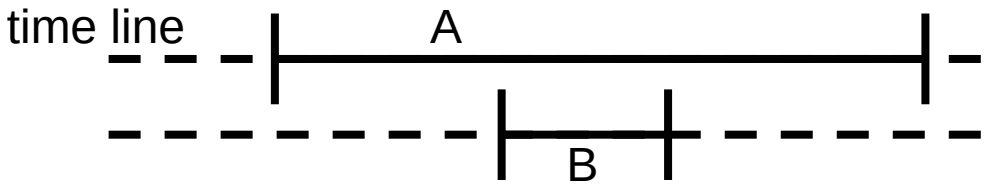
These are the most simplest and easy to understand functions. We say time A is before another time B iff the ending point of A is before the starting point of B. As shown below, time A is before time B. And time B is after time A.



Note that Before / After is not applicable to periodically repeated time, thus, we cannot say if time “Every Friday” is before “2003/12/25” or not.

- Covers / Covered

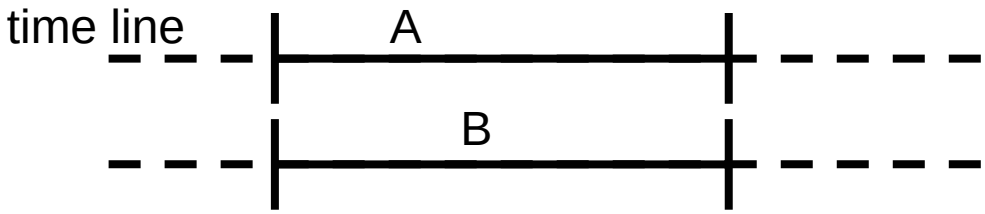
We say time A covers time B iff the starting point of A is before the starting point of B, and the ending point of A is after ending point of B. As shown below, time A covers time B, and time B is covered by time A.



Covers / Covered are applicable to periodically repeated time only when both times are periodically repeated. Such as “Every Friday” covers “Every Friday, 15:00 – 17:00”. The basic idea is, when every time point within a time structure do not excess another time structure, it is said to be covered.

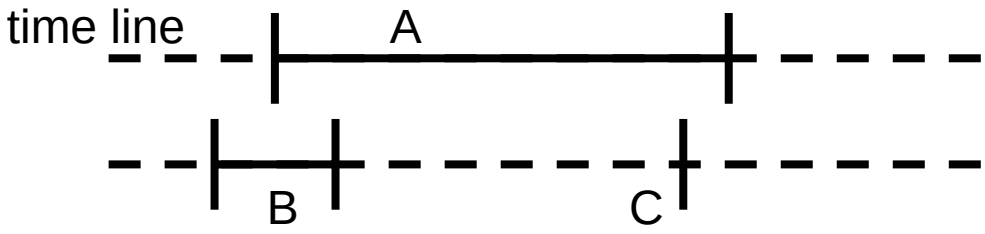
- = (equal)

If two time structure are meaning exactly the same period, it is said to be equal.



- CP (Common point)

CP is a predicate to check if there are any common points between two time structures. As shown below, A CP B returns true; A CP C returns true; B CP C returns false.



Except Boolean predicates, we proposed functions to deal with the time structure.

- Nearest Common Point

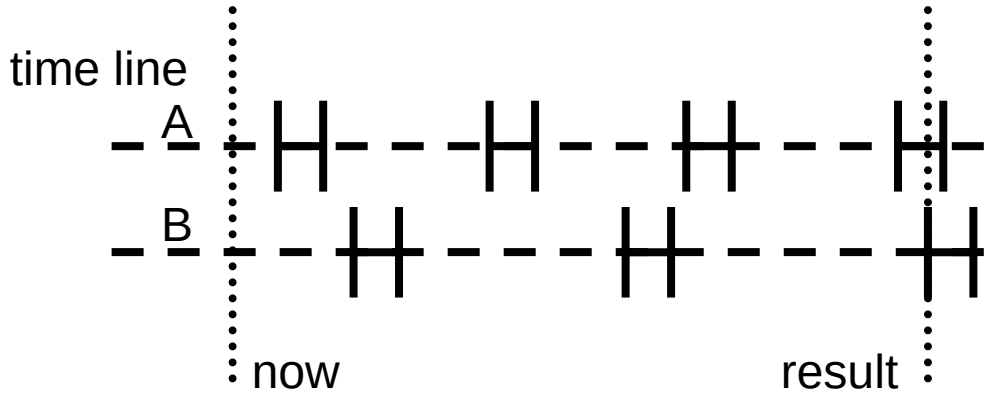
Prototype: *NearsetCP (TimeStructureA, TimeStructureB, TimeC):Time*

This function calculate the nearest common point of two time structure A and B, base on the exact time C. For example,

A = “Every Friday”

B = “1st of every month”

And we want to calculate the nearest day that can match both the time structures A and B from now.



- DaysCount

Prototype: *DaysCount(TimeStructureA):integer*

This function counts the number of days within a time structure. For example, the time is “from 2003/12/25 – 2003/12/31”, DaysCount should return 7.

Similar functions can be developed for HoursCount(), MinutesCount() or SecondaCount(), etc.

Memory Size

The memory size required for the proposed time structure is 17 bytes. The details are listed as follow:

Item		***Field type***	***Bits required***
From	Year	Integer	16
	Month	Integer	8
	Day	Integer	8
To	Year	Integer	16
	Month	Integer	8
	Day	Integer	8
From	Time	Double	32
To	Time	Double	32
Weekdays	Sun	Boolean	1
	Mon	Boolean	1
	Tue	Boolean	1
	Wed	Boolean	1
	Thu	Boolean	1
	Fri	Boolean	1
	Sat	Boolean	1
Continuous		Boolean	1
		Total	136 (17 bytes)

DTD and XML Coding

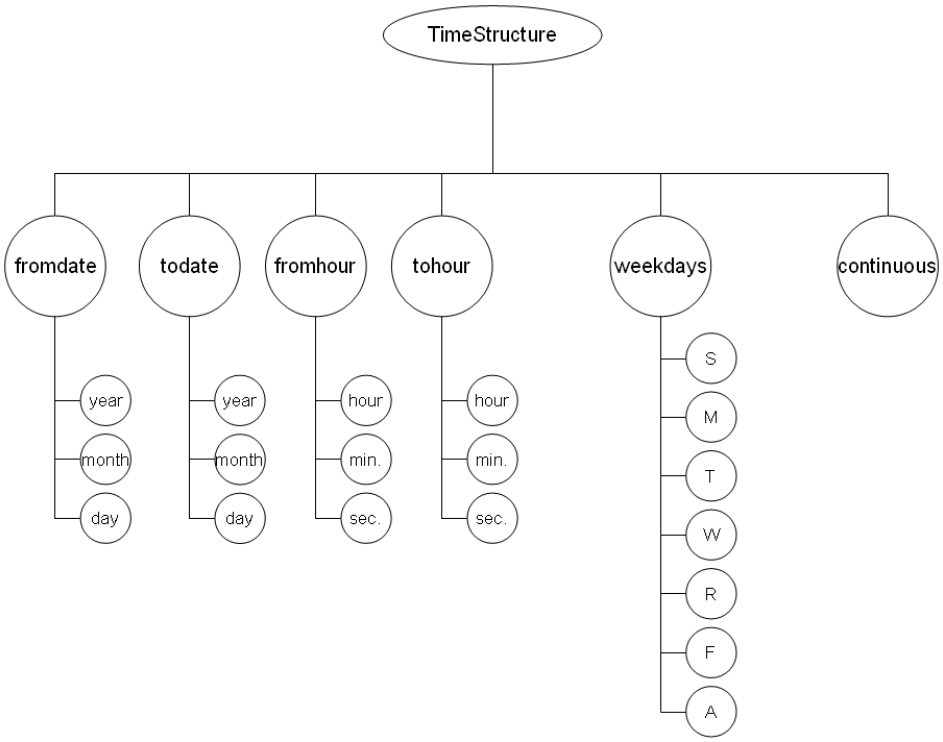


Figure 1: data structure for time interval

For example, in a company, the schema for the database and the DTD graph are shown in figure 2

Staff (ID, name, officehour)

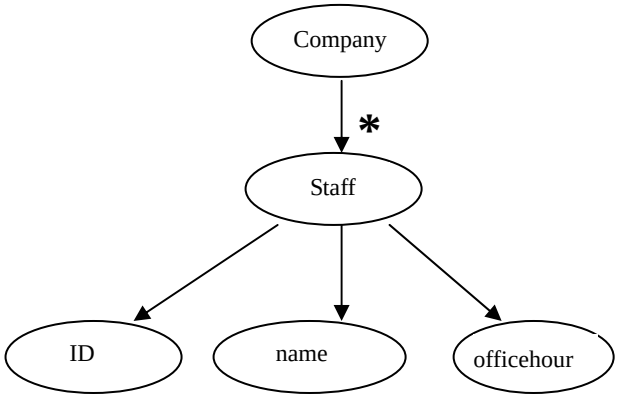


Figure 2: DTD graph

XML definition of the time structure:

```
<xsd:complexType name=" TimeStructure">
        <xsd:fromdate>
            <xsd:element name="year" type="xsd:integer"/>
            <xsd:element name="month" type="xsd:integer"/>
            <xsd:element name="day" type="xsd:integer"/>
        </xsd:fromdate>
        <xsd:todate>
            <xsd:element name="year" type="xsd:integer"/>
            <xsd:element name="month" type="xsd:integer"/>
            <xsd:element name="day" type="xsd:integer"/>
        </xsd:todate>
        <xsd:fromhour>
            <xsd:element name="hourminsec" type="xsd:double"/>
        </xsd:fromhour>
        <xsd:tohour>
            <xsd:element name="hourminsec" type="xsd:double"/>
        </xsd:tohour>
        <xsd:weekdays>
            <xsd:element name="S" type="xsd:boolean"/>
            <xsd:element name="M" type="xsd:boolean"/>
            <xsd:element name="T" type="xsd:boolean"/>
            <xsd:element name="W" type="xsd:boolean"/>
            <xsd:element name="R" type="xsd:boolean"/>
            <xsd:element name="F" type="xsd:boolean"/>
            <xsd:element name="A" type="xsd:boolean"/>
        </xsd:weekdays>
        <xsd:element name="continuous" type="xsd:boolean"/>
</xsd:complexType>
```

DTD schema:

```
<!DOCTYPE companyname [
    <!ELEMENT company ((Staff)*)>
    <!ELEMENT Staff(ID, name, officehour)>
    <!ELEMENT ID (#PCDATA)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT officehour (TimeStructure)>
]>
```

Usage sample in XML:

```
<xsd:element name = "company">
    <xsd:complexType>
        <xsd:Staff>
            <xsd:element name="ID" type="xsd:integer"/>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="officehour" type=" TimeStructure">
        </xsd:Staff>
    </xsd:complexType>
</xsd:element>
```

Limitations of the proposed structure

Although the time structure we proposed seems to be able to handle many kind of time interval problem, it still has some limitations in the moment.

First of all, this structure cannot be sorted, or, can only be sorted in a meaningless way. For some periodic value like “Every Sunday” and “Every Friday”, we cannot say which one is larger than which one. Similarly, we cannot decide if “Every Tuesday” is greater than “2003/12/25” or not. So it is meaningless to sort the data.

As the data is meaningless to sort, this time structure cannot act as a key. Apart from this reason, there is another reason that makes this time structure cannot act as a key. Since two time structure may intersect each other, but presented in two different ways, like “Every Wednesday” can intersect with “1st day of every month”. If we let the structure to be a key, every time when we add a record, we have to compare with all the existing records in the table in order to check if the key is unique. The computational cost to create a table will be O (n^2).

Conclusion

In this project, we concerned on Time Interval problem. It is easy to represent point-based time data in database. But it is complicated to deal with time interval, since time interval includes starting time, duration and stop time. Some scientists use object-relational DMBS to represent time interval data but none of them uses semi-structured database to store the time interval data. Therefore, we try to propose a time structure from object-relational database to XML. Our time structure can store point-based time, a period of time, or event with partial data and so on.

This project aims at presenting idea on designing a new type beside string, integer, double, boolean, etc., which is to deal with time intervals. To conclude, we do hope database field can come up with a standard to deal with this problem in future.

Reference

[1] Das, AK, Tu, SW, Purcell, GP, and Musen, MA. *An extended SQL for temporal data management in clinical decision-support systems.* Proceedings of the Sixteenth Annual Symposium on Computer Applications in Medical Care, 1992:128-32.

[2] N. A. Lorentzos and Y. G. Mitsopoulos. *SQL Extension for Interval Data.* IEEE Trans. Knowl. Data Eng. 9(3): 480-499(1997).

[3] Hans-Peter Kriegel, Marco Pötke, Thomas Seidl. *Managing Intervals Efficiently in Object-Relational Databases.* Proc. of the 26th Int'l Conference on Very Large Databases (VLDB). 2000.

[4] C. X. Chen, J. Kong and C. Zaniolo. *Design and Implementation of a Temporal Extension of SQL*. The19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India (ICDE 2003)

[5] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl *Interval Sequences: An Object-Relational Approach to Manage Spatial Data. SSTD'2000.*

[6] David Toman. *Point-Based Temporal Extension of Temporal SQL*. DOOD 1997: 103-121.

[7] Costas Vassilakis: *An Optimisation Scheme for Coalesce/Valid Time Selection Operator Sequences.* SIGMOD Record 29(1): 38-43(2000).