# Spatial Data – the Final Frontier

(To boldly go into 3, 4 or more dimensions).

Feb 2009

## *This month's topic: Time – The 4th Dimension*

All databases have some time component, whether it is in the metadata, telling how timely the data is, or as part of the detail of an operation (e.g. when a title was transferred). Where the database in question is spatial in nature, (whether 2D or 3D) the time "dimension" must be carefully handled.

### True 4D

Thanks to Albert Einstein, the concept of time as a fourth dimension is well known (but not in the form he used it).

Thinking of time as a dimension, an event can be represented as a "point in time" with four coordinate values (*x*,*y*,*z*,*t*). Objects that last for a certain duration and do not move become "cylinders", with their boundaries defined by 3D objects, and their time "walls" being vertical. Moving objects become "tilted cylinders".

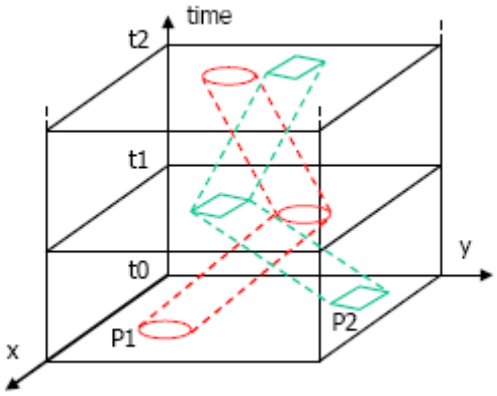


**Figure 1 Moving 2 dimensional objects as tilted cylinders**

It is hard enough to draw 3D diagrams, without attempting 4D, so this figure shows 2D moving objects in 2+1D.

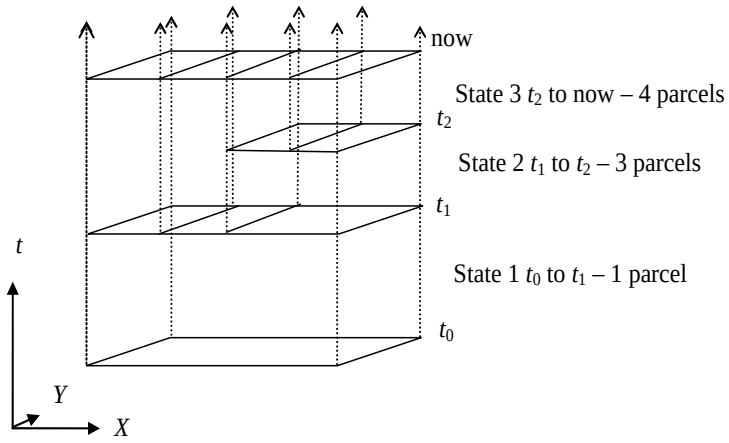


**Figure 2 Subdivisions of a rectangular parcel in time**

Subdivisions of parcels can likewise be stored as 4D objects. In this case, 2D land parcels being subdivided take the form of solid blocks.

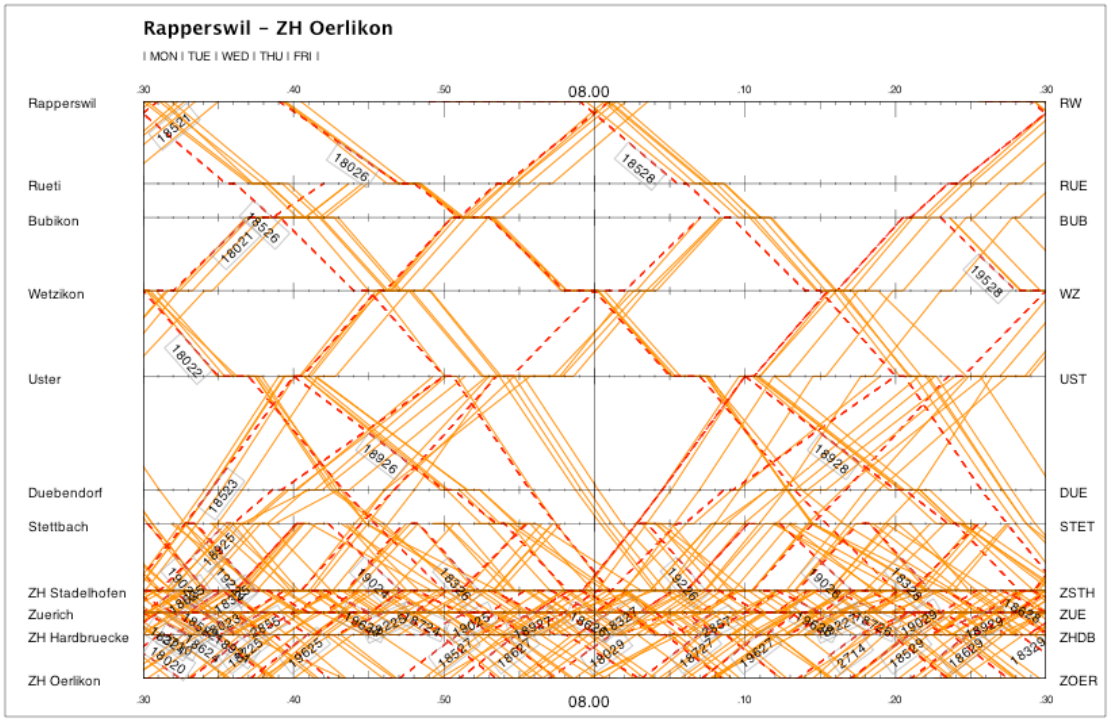


**Figure 3 A planning timetable for railway operations (Note – in this case, space is up the page, time is across)**

Note, that this kind of thinking is used in railway timetable planning [1], where the movement of a train becomes a diagonal line. Spatially, it is not quite 2D. The tracks are effectively 1D objects, but there are multiple tracks – otherwise the lines could not cross.

Apart from this use, the representation of time in spatial databases does not often use the "one more dimension" approach. This is because the extra complexity required to add a dimension to the problem is high. The following techniques make use of the special nature of the time dimension to simplify the problem.

# Time Stamping

Many databases and interchange specifications include fields such as "creating_time_stamp" and "destroying_time_stamp" (or "retiring_time_stamp") as attributes of one or more of the object classes or tables. In other models, such as the Land Administration Domain Model (LADM) many of the classes are subclasses of "VersionedObject" (or other similarly named class), which carries the time stamps.

This is now probably the most widely accepted method of including history in a spatial database, but this was not always the case. When we introduced the technique to the DCDB, it was not well known, but it did work, and proved its efficiency.

In 1997, Peter van Oosterom (of the Dutch Cadastre) carried the idea further, writing a paper [2] that showed that a full, consistent topology could be maintained with a complete history of the changes. Moreover, the topological connectivity was maintained at all times. At the time, topology was considered to be more than complex enough, without having to worry about history, so this was quite an eye-opener.

The beauty of the technique is that the true complexity of the system is not significantly increased by this method of including history. It is highly recommended for use within the relational database model.

## What You Get

It is possible, in effect; to view the database as it was at any point in time (between when the database was created and now). The result of enquiring on the database "as at" a particular date/time is exactly what would have been reported on that day. Thus if a recommendation had been made based on information obtained from the database, that information can be retrieved exactly.

It is also possible to retrieve a full history of the changes that were made within a range of dates; and for any specific update, it is possible to determine what database records were changed, and the exact change made to each.

## How to Do It

Assuming we have a well-designed data model for a system, but wish to include history in that database; all that is necessary is to add a "create" and a "destroy" time stamp to each object class in the system. Instead of changing any object in the database, that object is retired using the "destroy" time stamp, and a new version is created with a "create" time stamp having the same value.

New objects are created with a null "destroy" time stamp, and instead of deleting an object, it is given a non-null "destroy" time stamp.

In the relational model, to query the database "as at" time *t*, the following code is added to each select statement:

```
WHERE tab.creating_time_stamp <= t
AND tab.destroying_time_stamp > t
```

There must be a pair of "AND" statements for each table "tab" in the select statement.

That is all there is to it – all enquiry programs will retrieve the information as it was at time *t*. The enquiry on the current database just becomes a special case – in which *t* is

now. Note that the null value used in the destroy time stamp must test as greater than any time *t*. (i.e. *t* < null for all *t*).

Address Table

| number | street | suburb | postcode | creating time stamp | destroying time stamp |
|---|---|---|---|---|---|
| 39 | Heath St | Tingalpa | 4169 | small null | 25-Aug-2008 09:45 |
| 41 | Heath St | Tingalpa | 4169 | 25-Aug-2008 09:45 | 12-Nov-2008 13:20 |
| 41 | Heathrow St | Tingalpa | 4169 | 12-Nov-2008 13:20 | large null |
| | | | | | |
| | | | | | |

**Figure 4 Example - an address table with timestamping**

For example, in Figure 4, the clauses

```
WHERE tab.creating_time_stamp <= “1-Nov-2008 15:00”
AND tab.destroying_time_stamp > “1-Nov-2008 15:00”
```

will retrieve the address as “41 Heath St Tingalpa 4169”, while 1-Dec-2008 will return “41 Heathrow St Tingalpa 4169”, provided that the “large null” in the destroying lock number tests as greater than any valid date.

## Advantages

It does not increase the complexity of the programming – apart from just adding the “AND” statements.

It only mimimally increases the access times. (Because the database now grows with the history records, access times do become slightly slower, but the usual indexing prevents that from being too extreme).

It is possible to create a view of the database for users who do not wish to consider the temporal aspect. For example,

```
CREATE VIEW current_address AS SELECT number, street,
suburb, postcode FROM address where destroy_time_stamp
IS NULL;
```

This view behaves as though it is a table containing only the current addresses.

It is easy to archive database objects with a destroy stamp earlier than a particular date, moving them to off-line storage, but this is seldom done in practice, since it is convenient and relatively cheap to keep them on line.

## Limitations

Typically, this approach records a history of the database – not a history of the real world. Thus, if an error in the database is corrected, it remains in history. For example, if a street name was misspelled in the database, and corrected last week, if the database is queried as at two weeks ago, the name is still wrong. It is important to be aware of this fact.

Note that the relational tables in this case cannot have a simple “primary key”. It is usual to include the destroy time stamp as the least significant of what becomes a unique key to each row, but this is not a primary key.

For example in the DCDB, the unique key to the parcel table is "segment number", "parcel number" and "destroying_lock_nr". A constraint ensures that there are no two rows in the table with this same key, and therefore that there are no two current rows with the same segment/parcel numbers. This is not a primary key, since access to the database is not by exact search for a particular segment/parcel/destroying_lock_nr combination. The nearest to a primary key access to this table would be:

```
WHERE segment_nr = … AND parcel_nr = …
AND creating_lock_nr <= …
AND destroying_lock_nr > …
```

This is significant when considering Object/Relational mapping approaches.

### *SQL3 and Oracle 10g*

The support for this type of timestamping is included in the SQL3 standard, and in fact is extended to allow for real world history as well as database history (using the concepts of "valid time' and "transaction time" respectively) [3] (see section on 2D time below). Some of these concepts have been included in Oracle 10g [4], albeit in a form with different syntax.

### *Timestamping at the Attribute Level*

It is also possible to apply the timestamping approach at the attribute level. That is to say, rather than creating a new version of the whole table row for each update, each individual attribute is associated with a creating and destroying time stamp. While this might save some storage, it greatly complicates the model in a relational database, and is perhaps better suited to an object-oriented model.

## Reverse Delta

Another major approach to allowing databases to record a history is the "reverse delta" (the forward delta being far less common).

The term delta is taken from mathematics, where the Greek letter delta $\delta$ is often used to represent a difference. For example, SourceSafe uses this form.

In this approach, each time a database object is modified, its current record is changed, but an additional record is created with enough information to allow the previous object to be re-created. This may be implemented by simply creating a "before image" of the database object, or by recording which attribute(s) have been modified.

For example, an address record could contain: "39 Heath St, Tingalpa", and be corrected at time 25-Aug-2008:09:45 to "41 Heath St, Tingalpa". The database could now contain two records – a current record with the new address, "last updated" at 25-Aug-2008:09:45, and a reverse delta record saying "house number changed from 39".

The example of Figure 4 would be encoded in reverse delta form as:

Address Collection

| number | street | suburb | postcode | last update date | last update pointer |
|---|---|---|---|---|---|
| 41 | Heathrow St | Tingalpa | 4169 | 12-Nov-2008 13:20 | 3004 |
| | | | | | |
| | | | | | |

Address Delta

| id | field | prior value | update date | prior update pointer |
|---|---|---|---|---|
| 3001 | number | 39 | small null | null |
| 3004 | street | Heath St | 25-Aug-2008 09:45 | 3001 |
| | | | | |
| | | | | |

**Figure 5 Reverse delta encoding of address changes**

This method is unsuitable for use in relational databases since the complexity of the SQL increases explosively with each temporal table added.

## What You Get

As with the timestamping approach, it is possible to view the database as it was at any point in time (between when the database was created and now). The result of enquiring on the database "as at" a particular date/time is exactly what would have been reported on that day.

It is also possible to retrieve a full history of the changes that were made within a range of dates, and for a specific update action, it is possible to determine which database records were changed, and the exact change made to each.

## How to Do It

Assuming we have a well-designed data model for a system, but wish to include history in that database, we need to add a delta class for each object class in the database. We also need to add a timestamp for the date/time of update to each object class, and a pointer to the changes. For example, in Figure 5, the original table has had "last update date" and "last update ptr" added to it. The "delta" records are stored in another collection, and are chained in reverse chronological order.

In order to view the database as at a given date/time *t*, it is necessary to seek the current objects that represent the data to be retrieved, and to seek back along the list until the last_update_date < *t*.

## Advantages

The seeking of current data is hardly affected at all by this approach.

It may require less storage than the timestamping approach – if only the changed attributes are recorded.

It is easy to archive delta objects with a last update date earlier than a particular date, but this is seldom done, since it is convenient and relatively cheap to keep them on line.

### Limitations

As with the timestamping approach, it usually records a history of the database – not a history of the real world.

The time taken to scan the chain of updates increases linearly the further back time *t* is. Further the access to an object type requires potentially two collections to be queried.

This approach is inefficient in a relational database, and prevents any normal SQL access to the historic data.

## Other Temporal Issues

There are some important considerations in including the time dimension in a database.

### Database Time

It has been mentioned above that the timestamping and reverse delta approach typically record a history of the database rather than the history of the world. In many cases, this is exactly what is required, but not always.

It must be remembered that it is database history, so errors in history will always remain. This also includes database constraint violations. For example, if the database validation is tightened at time $t_1$, introducing a constraint, then an extract of data "as at" before $t_1$ may return data that violates this constraint. If programs have subsequently been written that rely on that constraint being enforced, they may fail.

In effect, this means that programs written to use the database must cater for the weakest form of validation that was in force through the life of the database.

### Real World Time

It is possible to vary these approaches to consider real world time, but it is not easy. For example, it would be possible to insist that the time stamp values used were the time the real world object changed e.g. the road was widened at time $t_w$. This means that, ignoring the possibility of errors in the database, what is retrieved from the database is a record of what was true in the real world as at the time *t*.

**Several problems arise:**

Errors do exist, and must be corrected. Not only the current record needs to be changed, but history needs to be rewritten. For example, if a road name was misspelt, then that name need to be corrected on the current road record, and also on the road record in history before the widening event.

Road Table

| road_id | name | width | creating time stamp | destroying time stamp |
|---|---|---|---|---|
| 1005 | Death St | 20 | small null | 25-Aug-2008 09:45 |
| 1005 | Death St | 25 | 25-Aug-2008 09:45 | large null |
| | | | | |

The record of the widening event

Road Table

| road_id | name | width | creating time stamp | destroying time stamp |
|---|---|---|---|---|
| 1005 | Heath St | 20 | small null | 25-Aug-2008 09:45 |
| 1005 | Heath St | 25 | 25-Aug-2008 09:45 | large null |
| | | | | |

After the error has been corrected

**Figure 6 Correcting an error in a real-world history**

It is not always easy to distinguish between database and real-world events – e.g. in the above case the street may have been named "Death St", and the residents objected and had it changed.

## Two Dimensional Time

It may be necessary to maintain database time as well as real-world time – leading to the possibility of two dimensions of time, with a history of the history being maintained. For example, a real-world history is maintained, but with a database audit trail logging changes to the database.

Road Table

| road_id | name | width | real create ts | real destroy ts | db create ts | db destroy ts |
|---|---|---|---|---|---|---|
| 1005 | Death St | 20 | small null | 25-Aug-2008 09:45 | small null | 2-Feb-2009 12:23 |
| 1005 | Heath St | 20 | small null | 25-Aug-2008 09:45 | 2-Feb-2009 12:23 | large null |
| 1005 | Death St | 25 | 25-Aug-2008 09:45 | large null | small null | 2-Feb-2009 12:23 |
| 1005 | Heath St | 25 | 25-Aug-2008 09:45 | large null | 2-Feb-2009 12:23 | large null |
| | | | | | | |

**Figure 7 The example of Figure 6, with audit trail of changes**

This probably requires a time lord to comprehend the issues (where is Doctor Who when you need him?) – e.g. SQL3 allows selections to be coded to solve these examples by Snodgrass *et al* [3]:

Which Tuscon employees are currently paid highly?

Who did we think are the highly paid Tuscon employees?

When was it recorded that a Tuscon employee is currently highly paid?

When did we think that a Tucson employee was paid highly, at the time they resided in Tucson?

Who was recorded, perhaps erroneously, to have resided in Tucson at some time and was paid highly, perhaps at some other time?

And so on …

### Future Time

These techniques can all be adapted to allow future events to be recorded. That is to say, an event that must occur at some time in the future can be accommodated in the database, and when the world catches up, the event becomes visible. For example, if it is known that a place will be officially renamed at future time *t'*, the existing name record could be retired at *t'*, and the new name record created with time stamp *t'*. For time < *t'*, all enquiries to the database (at time "now") will show the old name. Following *t'*, and with no change to the database, enquiries will now show the new name.

This is not always what is wanted, and cannot be applied to database time. In some systems such as DCDB and RIME, there is kind of "virtual future" time (way off in the future) that changed records are placed in until they are signed off. Then they return to our time scale, but only by program intervention.

### Other Events

This is a form of real world time that is simpler than the two dimensions of time mentioned above. There are often events in the history of the objects being represented that are needed to be recorded. For example, the date of publication of a map, These can be happily recorded in the database like any other attribute, and there is no extra complexity. Indeed, if the publication date is updated, there will be a history record created showing the old date prior to update, and the new date post-update.

This is not a problem, but can lead to some confusion where the date has some connection with the database. For example, where a data item has "date of capture" as an attribute, confusion can arise. The user may be able to update the "date of capture" attribute, while the creation time stamp cannot be updated. Thus there may be conflicting information in the database. (This may be correct, for example, because the date of capture could be recording the date the data was loaded into some prior computer system, while the timestamp records entry to this system).

## Event Table

In describing the timestamping and reverse delta approaches, it has been assumed that there is a timestamp data type that records the date and time of the operation, and can be stored in the object.

There is an alternate approach, where a table is maintained, with a sequentially allocated identifier being allocated to each event that affects the database in any way. For example, in the DCDB, there is a history_header table, where each update to the database is logged. Because the updates are done in sequential order, as records are added to this table, the next sequential number can be allocated and used as a unique identifier.

| ID | TimeStamp | UserId | Project | Update deatils |
|---|---|---|---|---|
| 2024 | 2008/12/1:09:00:00.0000 | Thompson_R | upgrade | from field data Nov 2008 |
| 2025 | 2008/12/1:09:00:23.0000 | Jones_K | update | Correction of errors |
| 2026 | etc | | | |

**Figure 8 History header table**

The ID of the event table can then be used as a timestamp for the creation and retiring events to the database objects, and will be applied to all the table rows that are affected by the change. This also resolves the unlikely case of two updates that occur at exactly the same time.

Special events of "zero" (less than any ID in the event table) and "infinity" are defined and used for the null values for the creating and destroying time stamps respectively.

The advantage of this approach is that the event table can record metadata about the update – who made it, why, from what terminal, etc as well as the date and time. It also prevents clashes that could come about if updates were allowed to take place at exactly the same data and time.

(In the RIME database, there is an extra level of apparent complexity caused by the event table being a subset of the metadata table. Metadata items of type "update" are considered to be events, and the metadata_id of these items are used as create/destroy timestamps. This does not really cause any real additional complexity, but can be confusing to explain).

Rod Thompson

Feb 2009

## References

1. Luethi, M., Huerlimann, D., and Nash, A. (2005) *Understanding the Timetable Planning Process as a Closed Control Loop.* Institute for Transport Planning and Systems, ETH Zurich

2. van Oosterom, P. (1997) *Maintaining Consistent Topology including Historical Data in a Large Spatial Database*. in *Auto Carto 13*. Seattle, WA.

3. Snodgrass, R.T., Böhlen, M.H., Jensen, C.S., and Steiner., A. (1998) *Transitioning temporal support in TSQL2 to SQL3.* Lecture Notes in Computer Science, 1399: p. 150-194.

4. Murray, C. (2005) *Oracle® Database Application Developer's Guide - Workspace Manager, 10g Release 2 (10.2).*