A Model for Schema Versioning in Temporal Database Systems

John F. Roddick

Advanced Computing Research Centre
School of Computer and Information Science
University of South Australia
The Levels, Adelaide
South Australia 5095
roddick@cis.unisa.edu.au

Abstract

The aim of temporal data models is to accommodate the nature of time naturally and directly and avoid the ad hoc, one-off extensions commonly employed. Schema versioning is one of a number of related areas dealing with the general problem of using multiple schemata for various database related tasks. In particular, schema versioning, and its weaker companion, schema evolution, deal with the need to retain current data and software system functionality in the face of changing database structure. This paper brings temporal and evolutionary models together and proposes an extension to temporal data models capable of accommodating partial schema versioning.

Keywords: Schema Evolution, Partial Schema Versioning, Temporal Data Modelling.

1 Introduction

In recent years there has been a lot of work investigating the accommodation of change in data models. Change in data results from the dynamic character of the real world and has resulted in research into dynamic and temporal modelling techniques. More recently, change in data structure, reflecting changes in real-world object morphology or user requirements, has resulted in research in schema evolution and schema versioning. Within the database systems field, this has resulted in the prototype development of temporal and evolving database systems.

A variety of temporal data models have been proposed in the literature (Ariav et al., 1982, Ariav, 1986, Bubenko, 1977, Clifford and Ariav, 1986, Elmasri and Wuu, 1990, Gadia, 1986, 1988, Lorentzos and Johnson, 1988, Navathe and Ahmed, 1989, Roddick and Patrick, 1991, Segev and Shoshani, 1987, Tansel, 1990) and a comparative survey of those based on the relational model is given by McKenzie and Snodgrass (1991). Substantially fewer models have been proposed accommodating schema evolution or schema versioning (Banerjee et al., 1987, Klishold et al., 1986, McKenzie and Snodgrass, 1987) although an increasing number of papers in this field are appearing (Roddick, 1992). This paper brings these models together and proposes an extension to temporal data models capable of accommodating partial schema versioning.

This paper first briefly outlines the necessary characteristics of a temporal, evolving model. Because of the complexity involved, this necessitates giving a number of definitions which will be used to delineate the scope of the model. In Section 3 the extensions are then presented and evaluated with respect to the pragmatic requirements of such a model. Some discussion of other research issues is then given in Section 4.

2 Characteristics of Temporal, Evolving Data Models

The growth in research in accommodating changing data and data structure has resulted from, above all, a strong user requirement for stable systems in changing environments. Numerous analyses of future database directions has indicated that legacy systems, change management and high availability are currently, if not poorly, at least inadequately supported by databases (Laguna Beach Report, 1989, Selinger, 1993, Stonebraker et al., 1993). From our perspective, this gives a strong pragmatic emphasis to the development of a temporal, evolving model.

In addition, the development of commercial object-oriented database management systems has divided (very broadly) into two groups; the unified architectures which are attempting to build OO functionality on top of a relational engine (for example UniSQL/X (Kim, 1993) and Postgres/Montage (Stonebraker and Kemnitz, 1991, Stonebraker and Rowe, 1986)), and the pure approach which generally does not (for example O2 (Deux, 1991) and ObjectStore (Lamb et al., 1991)). The approach taken here is to extend those temporal data models which are based on the relational model and thus provide a basis that may be able to be utilised by the unified architectures. The decision to extend the relational model is reinforced by recent initiatives to extend the SQL92 standard to support temporal data and, to a more limited extent, evolving schemata (Snodgrass et al., 1994).

Schema versioning is defined as being accommodated when (Jensen et al., 1994):

... a database system allows the accessing of all data, both retrospectively and prospectively, through user definable version interfaces.

Schema versioning subsumes the concept of schema evolution which in turn subsumes that of schema modification. A more complete discussion of some of the general schema evolution issues can be found in (Roddick, 1995). The definition can be further refined by distinguishing between retrieval and update activity as follows:

Partial Schema Versioning is accommodated when a database system allows the viewing of all data, both retrospectively and prospectively, through user definable version interfaces. Data updates...
are allowable through reference to one designated (normally the current) schema definition only.

Full Schema Versioning is accommodated when a database system allows the viewing and update of all data, both retrospectively and prospectively, through user definable version interfaces.

Some of the problems associated with schema versioning are analogous with those associated with data and view integration which aims to merge schemata from different sources. Research in this area has largely been directed at integrating heterogeneous systems, but the applicability to the evolution of schemata is clear. Miller et al. (1993), for example, investigate the concept of the information capacity of a schema to decide whether schema or view integration would be lossless. A taxonomy is given in which the translation ability for two schemata is distinguished by their ability to retain information during update and retrieval. This aspect is also investigated by Orłowska and Ewald (1992) who view schema integration as a schema evolution process in which the integration of two or more schemata is effected by choosing one and applying the facts held in the others. Geller et al. (1992) present a method which allows the integration of structurally similar but semantically dissimilar datasets by using a "dual" model which maintains separate representations for structure and semantics. Although very similar to data and view integration and although many aspects of this work are common, the accommodation of historical data and the ability to retain the evolutionary history of schemata enable and necessitate a different treatment of the problem.

Miller et al. (1993) have shown that in order to update data stored under two different schemata using the opposite schemata, they must be equivalent, i.e. all valid instances of some schema $S$ must be able to be stored under $S_2$ and vice-versa. Since we cannot (generally) foresee the future requirements of the database, and hence the changes required to the data structure, and since neither the active nor the historical schemata can be changed (we may only create a new version which supercedes the old), this is too strong a condition to impose. We therefore adopt the weaker concept of partial schema versioning in which data stored under any historical schema may be viewed through any other schema but may only be updated through the current or active schema. Even this level of support necessitates imposing some restriction on the schema modification which can be achieved without reorganisation and data coercion. This aspect will be discussed in more detail in Section 3.

A set of characteristics for a temporal, evolving data model are now presented. The length of this paper precludes lengthy discussion on these, however references are provided to other sources for further information.

1. The model must support bitemporal data. While a number of valid-time and transaction-time only models have been proposed and while valid-time and transaction-time are orthogonal, it is important that the model be general with the mon-temporal models as special cases

2. The model must degrade, in a conceptually simple manner, to non-evolving, snapshot or temporal models.

3. The model must accommodate retrieval of all data through any defined version interface. Note that version interfaces may cease to be defined under specified circumstances.

4. The model must accommodate the ability to update all (allowable) data, although the version interfaces allowable for this purpose may be restricted. In the context of the definitions above, this model accommodates partial schema versioning and allows update only through the current schema definition.

5. The model must accommodate a set of restructuring operations which must be both complete and pragmatically useful. Completeness in this sense requires not only that any allowable relational structure be able to be created but that this evolution is also able to be tracked. SQL's DDL statements, for example, while able to permit the creation of any relational structure do not permit all forms of evolution to that structure, for example, creating one relation from the outer join of two others.

6. The model must accommodate a set of pragmatically useful, type conversion rules to allow for sensible evolution of attribute domains.

7. The model must facilitate some form of meta-level vacuuming in order to remove unwanted schema versions (Roddick and Snoedgrass, 1995).

3 An extension to temporal models to support versioning

3.1 Completing the Relation

The extension presented here uses, as an example, the bitemporal conceptual data model (BCDM) employed in the TSQL2 language design (Jensen et al., 1995) although the ideas presented are generally applicable to many other temporal models. Under the BCDM, time is conceptually modelled as consisting of bitemporal chronons (elementary rectangles) existing in the two dimensional space between transaction and valid-time. A set of chronons is termed a bitemporal element. Facts are represented by tuples consisting of an arbitrary number of explicit attributes and an implicit timestamp attribute which represents a bitemporal element. Thus a tuple in an instance of a relation comprises an arbitrary number of attributes representing a fact and a temporal element indicating the temporal validity of the fact.

Consider the example in Figure 1 adapted from Jensen et al. (1995). Flight XX025 to Sydney was to leave at 1:00pm and arrive at 3:30pm. This was recorded at 11:15am that morning. At 12:30 a delay of an hour to the flight was reported. At 2:15pm an actual departure time was recorded of 1:30pm as well as a new estimated time of arrival of 3:30pm. Finally at 4:00pm the actual arrival time was recorded as 3:45pm.

The introduction of schema versioning affects the composition and the method of retrieval and update of the explicit attributes (in the example above, flight number and destination). Given a bitemporal relation scheme $R = (A_1, \ldots , A_n, T)$, a tuple $x$, without

---

1Terminology in this paper will adhere to the consensus glossary presented by (Jensen et al., 1994).

2Although it will be assumed in this paper that all changes to a database schema result in a new version interface, this need not be necessary. This is discussed further in Section 4.

3Bitemporal and transaction-time models do not allow changes to data stored with a transaction-time end date which is not set to the open value until changed. This is analogous with events on the transaction-time axis being linked to the system clock.
schema versioning would have the form \((a_1, \ldots, a_n)\).
With the introduction of schema versioning, the concept of a non-temporal completed relation scheme \(C\) is introduced which contains the minimal union of all explicit attributes which have been defined during the life of the relation. Moreover, the domain of each attribute in \(C\) is considered syntactically general enough to hold all data stored under every version of the relation scheme \(R\) and the implicit primary key of \(C\) is defined as the maximal set of key attributes for the scheme over time. Versions of the schema (denoted \(S_{time}\)) can be seen to be views of \(C\). Thus the relation scheme active during the interval \(t_1, R_{t_1} = (S_{t_1}, T)\). The current schema through which updates may be performed is denoted \(S_{now}\).

### 3.2 View Functions

A view function \(V_t\) maps \(C\) to a subset of the attributes in a schema \(S_{t}\) active during \(t_1\). The converse function \(W_{t}\) maps from \(S_{t}\) to \(C\). An example is shown in Figure 2.

![Figure 2: Versions of Schemata over time](image)

Thus the data stored during \(t_2\) may be mapped to the format specified during \(t_5\) through invocation of \(V_{t_5}(W_{t_2}(S_{t_2}))\). Note that the view functions \(V\) and \(W\) are syntactic devices only and can be considered as complex casting mechanisms.

As an example, consider the following structural history for an Employee relation:

1 Jan, 1995  Relation Employee defined as:

- **Id**: NUM(6),
- **Name**: CHAR(30),
- **Salary**: NUM(4)

1 Feb, 1995  Attributes

- **Gender**: CHAR(1),
- **Maritalstatus**: CHAR(1)

1 Mar, 1995  Attribute **MaritalStatus** deactivated.

1 Apr, 1995  Attribute **Salary** redefined as:

- **Salary**: NUM(5,2)

2 Apr, 1995  Attribute **Salary** redefined as:

- **Salary**: NUM(5)

The completed scheme for this relation would be:

- **Id**: NUM(6),
- **Name**: CHAR(30),
- **Gender**: CHAR(1),
- **Maritalstatus**: CHAR(1),
- **Salary**: NUM(5,2)

At all points in time the view functions \(V\) and \(W\) are available to convert from the stored schema to the completed schema \(C\) and then from \(C\) to the required schema.

Given a completed relation \(C = (A_1, \ldots, A_n)\) and a given version \(S = (B_k, \ldots, B_m)\), with instance \((b_1, \ldots, b_m)\), a view function \(V_t\) can be considered to be composed of attribute view functions \((v_{t_1}, \ldots, v_{t_n})\) as follows:

\[
\forall A_i \in S : v_{t_i} = \Gamma \tag{1}
\]
\[
\forall A_i \notin S : v_{t_i} \text{ is undefined} \tag{2}
\]

Furthermore the function \(W\) is defined as composed of the attribute view functions \((w_{t_1}, \ldots, w_{t_n})\) as follows:

\[
\forall B_i \in S \wedge \Psi(b_i, A_i) : w_{t_i} = \Gamma \tag{3}
\]
\[
\forall B_i \in S \wedge \neg\Psi(b_i, A_i) : w_{t_i} = \Delta_i(B_i) \tag{4}
\]

where \(\Gamma\) is one of a set of standard type conversion functions,

\[
\Psi(\beta, \alpha)
\]

indicates whether the application of \(\Gamma\) is lossless between each instance of \(\beta\) and the type of \(\alpha\),

and \(\Delta_i(\alpha)\) is a default value function for \(\alpha\) at \(t\) to be used in cases where \(\Gamma\) is not lossless.

For example, consider the following data stored on 1 Apr, 1995 under the example outlined earlier:

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Gender</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Smith</td>
<td>F</td>
<td>12,532.00</td>
</tr>
<tr>
<td>46</td>
<td>Brown</td>
<td>M</td>
<td>12,532.50</td>
</tr>
</tbody>
</table>

If this data was retrieved through \(S_{1(Apr, 1995)}\) the application of \(\Gamma\) is lossless. If this data was retrieved through \(S_{2(Apr, 1995)}\) the application of \(\Gamma\) for the first tuple is lossless but in the second is not. Thus in the second case the default value function is used (which in this instance may be a simple truncation or rounding).

If, however, this data was retrieved through \(S_{2(Jan, 1995)}\) the application of \(\Gamma\) in both cases not lossless and the default value function would again be used (which in this instance may return some pre-defined value indicating overflow).

In TSQL2 (Snodgrass et al., 1994), \(\Delta\) is defined through a new INAPPLICABLE clause which provides for a simple function restricted to an SQL function of attributes in that relation.

---

*It should be remembered that in a temporal database data may be marked as old but is never deleted (although see 4.2). Similarly, in evolving temporal databases, structure is superceded rather than deleted.*
### 3.3 Restrictions imposed on the completed scheme \( C \)

It can be seen that the retrieval of information requires a set of view functions which translate the data from any schema definition \( S_t \) through \( C \) to any other schema definition \( S_{now} \). Since schema modification is generally lossless, updates cannot be guaranteed not to violate key and other constraints. We therefore impose the following rule:

*The current schema must be an updateable view of the completed schema.*

This is required so that the current schema is always available for data update. Note that this does not imply that all attributes in \( C \) may be updated through \( S_{now} \); rather any update through \( S_{now} \) does not result in a violation of specified functional and foreign key dependencies for the relation and therefore, implicitly, in \( C \).

Given that when the database is initially defined \( S_{now} \equiv C \), and leaving aside relation composition and decomposition for a moment, \( S_{now} \) will be an updateable view unless the following has occurred:

- The domains of the primary key attributes have been reduced.
- The composition of the primary key of the relation has been reduced.

Note that many of the conditions normally specified for update through views do not apply here as schema updates can ensure that no existing data contradicts the new structure (for example, the new key definitions) and, under partial schema versioning, data will not be able to be added under any other schema.

### 3.4 The Completed Scheme \( C \) and Rules for Relation Composition

Schema versioning commonly allows the formation of one or more relations through the composition or decomposition of one or more original relations. \( q.v. \) (Roddick and Snodgrass, 1995, Shneiderman and Thomas, 1982). For example, two relations might be combined horizontally and then decomposed vertically in a database restructuring exercise.

In cases where the relation has been composed from other relations (either vertically or horizontally) the completed relation \( C \) must apply to two or more sets of historical schemata. Given the maximal nature of \( C \) and the fact that the view functions convert between schemata by converting to and from \( C \) this does not cause further problems. Note that it is the users responsibility (or the responsibility of a different part of the schema formation mechanism) to ensure that past data does not result in duplicate primary keys\(^6\).

For decomposition of relations, the completed relation is duplicated to each of the new relations. However, problems are encountered in the case of horizontal decomposition where relations have been horizontally decomposed and new data has been added as it is not clear which of the two old relations (if any) the new data is implicitly a member of\(^6\). Three options are suggested:

1. Disallow queries through historical versions,
2. Assume new data belongs to neither obsolete relation, or
3. Allow the implicit or explicit specification of a relation to resolve the problems during relation construction.

One overarching solution (essentially adhering to 1 and 2 above) would be to define the *life* of a relation to be since its *explicit* creation and not to include possible antecedent relations from which it may have been composed. However, given that neither vertical decomposition nor vertical nor horizontal composition suffer from this ambiguity, and thus this may be too restrictive a general rule to use.

The resolution of this problem is currently the subject of future research but at present we are investigating the use of the following rule:

*Data may be retrieved through antecedent relations except in cases where there could be ambiguity of the implied data source.*

Thus all schema changes are allowed but some will result in data ceasing to be retrievable from the source relations. This rule also allows for the possible incorporation of a disambiguation function.

Finally, note that the completed scheme can be simplified by removing old schema definitions (meta-level vacuuming).

### 4 Discussion and Future Research Issues

Apart from the data sourcing problem outlined in Section 3.4, many issues remain to be solved, some of which are listed briefly below. Other issues, including query language construction issues, are discussed in more detail in (Roddick, 1995).

#### 4.1 Meta-level transaction Processing

Many schema change requirements involve composite operations and thus a mechanism for schema level commit and rollback functions are suggested which could operate at a higher level to the data level commit and rollback operations. For example, since data updated to a revised schema may be inapplicable if the schema change is not itself committed, it may not be considered appropriate to allow data to be committed outside of the scope of the current session until the schema-level commit is issued. This allows for the definition and population of attributes to be completed as one molecular operation.

As an example, the following sequence of database operations may be issued to test a program:

1. Add attribute(s) to relation;
2. Populate attribute(s);
3. Data level commit;
4. Run test programs;
5. If tests successful
   - Schema-level-commit;
6. Else
   - Schema-level-rollback;

#### 4.2 Schema Vacuuming

In temporal databases the concept of vacuuming allows for the physical deletion of temporal data in cases where the utility of holding the data is outweighed by the cost of doing so (Jensen, 1993). Similar consideration must be given to the retention of old schema definitions, especially in cases where no data exists adhering to either that version (physically) or referring, through its transaction-time values, to the period in which the definition was active. Roddick and Snodgrass (1995) propose two pragmatic positions:

---

\(^{6}\)We take a pragmatic approach to schema versioning. Schema changes are generally required whether the database systems facilitates them or not. Our aim is to provide maximal resilience to schema change but we advocate providing warnings rather than prohibitions in cases where dataloss may occur.

\(^{6}\)We term this the *implicit data source* problem.
1. All schema definitions which pre-date all data (both in format and in transaction-time values) are to be considered obsolete and should be deleted;

2. Old schema definitions are considered valuable independent of whether data exists and may only be deleted through a special form of vacuuming.

Acknowledgments

Some of the ideas contained here resulted from work on a temporal extension to the SQL92 standard, TSQL2 (Snodgrass et al., 1994, Snodgrass, 1995). The commentaries giving background discussions on this initiative are available on-line in the tsq/directory at FTP.cs.arizona.edu via anonymous FTP or in the recently published (Snodgrass et al., 1994, Snodgrass, 1995) while further information on this initiative may be obtained from Prof. Richard Snodgrass at the University of Arizona.

References


McKenzie, L. and Snodgrass, R. (1987), Scheme evolution and the relational algebra, Technical Report 87-003, Department of Computer Science, University of North Carolina, Chapel Hill, NC.


