Study and Comparison of Schema Versioning and Database Conversion Techniques for Bi-temporal Databases

Han-Chieh Wei and Ramez Elmasri
Department of Computer Science and Engineering
The University of Texas at Arlington
{wei, elmasri}@cse.uta.edu

Abstract

Schema evolution and schema versioning are two techniques used for managing database evolution. Schema evolution keeps only the current version of a schema and database after applying schema changes. Schema versioning creates new schema versions and converts the corresponding data while preserving the old schema versions and data. To provide the most generality, bi-temporal databases can be used to realized schema versioning, since they allow both retroactive and proactive updates to the schema and database. In this paper we first study two proposed database conversion approaches for supporting schema evolution and schema versioning: single table version approach and multiple table version approach. We then propose the partial table version approach to solve the problems encountered in these approaches when applied to bi-temporal databases.

1. Introduction

Database and software applications usually evolve over time. This is due to several reasons, such as changes to the modeled reality, the application requirements, or the design specifications. Hence, a database schema is also subject to change even after careful design. In [17], quantitative measurements in an actual commercial relational database application are described, which show that changes to the database schema are not only unavoidable but also a significant maintenance concern during and after the process of system design. It consumes a lot of time and effort to convert the database and application programs whenever such changes occur. As a result, a mechanism to manage schema changes and maintain the database consistency after these changes would be a valuable addition to database management systems. *Schema evolution* [1,9,11,12,13,15,22,23] and *schema versioning* [4,7,14] are two of the most discussed techniques for managing database evolution in current database systems.

Basically, the difference between schema evolution and schema versioning is that schema evolution only keeps the current schema and corresponding data, whereas schema versioning preserves versions of schema and data during the evolution. Most current database systems are categorized as snapshot databases, where there exists only one version of a schema and database at any time. Therefore, whenever a schema is changed, the new schema becomes the current schema and thus the old schema is obsolete and will not be kept. The legacy data that followed the old schema become inconsistent and must be converted to the new schema. The problem of schema evolution is that the application programs written against the old schema and affected by the schema changes may need to be modified or at least need to be recompiled. Obviously this is very time consuming.

Another problem of this technique is that some applications, such as banking, auditing, medical records, and reservation systems require not only current but also past or even future information. These problems give rise to the approach of schema versioning. Basically, whenever the schema is changed, a new schema is created and defined as a new version of the schema. Accordingly, the new version of data is converted from the old data to be consistent with the new schema. Both the legacy schema and data are still stored in the catalog and databases system as old versions. As a result, the legacy data is preserved and also the applications defined on the old schema need not be rewritten nor recompiled. However, the concepts of time and history are still not included in this approach. In another words, the schema changes can only apply to the current version. Changes to the past or plans for the future schema still can not be captured.

Bi-temporal databases [18,19], incorporating both transaction time and valid time, can fulfill the requirements of schema versioning not only because they allow the users or application programs to access temporal information but also because they allow retroactive and proactive updates. Therefore, if the database schema is stored in a bi-temporal catalog, it can provide the most

flexibility for database schema evolution. However, there is a cost tradeoff between the flexibility of retroactive and proactive schema changes and the cost of implementing these mechanisms. The complexity is high because changes not only affect the current versions of data but also the past and even the futute versions which makes the database conversion much more complicated than for conventional snapshot databases.

Currently, there are two main appraoches to convert the database structure after applying schema changes in temporal relational databases: *single table version* (or *complete table*) [3,4] (**STV**) and *multiple table version* [4] (**MTV**). Most previous research [14] discusses versioning for transaction-time only databases. In [4], bi-temporal database versioning is studied but they only consider the simple cases, especially for the multiple table version approach. Also there is also no space analysis for the database conversion process for these two approaches. In this paper we will fully explore the effect of schema changes involving both retroactive and proactive updates in bi-temporal relational databases by comparing these two approaches. We focus on two basic schema update operations, attribute addition and attribute deletion since these two operations affect the storage space and table structure. Other operations can be executed using sequences of these two operations. Algorithms for schema update and database conversion are also defined for these two operations. The problems and complexity of the two approaches are also discussed. We then propose a compromise approach, *partial multiple table versioning* (**PMTV**), which not only will solve the problems associated with the previous two approaches but also can largely reduce the complexity.

In the next section, we briefly describe bi-temporal databases and the data model used for our presentation in this paper. The approaches of single table version and multiple table version are discussed in sections 3 and 4. An example is then given to show the problems and complexity. The approach of partial multiple table versioning that we propose is presented in section 5. The comparison with the previous two approaches is given in section 6. Section 7 concludes this paper.

2 Temporal databases

Temporal databases store current information, as well as past information and even information about the future events which are planned to occur. To serve the requirements of storing and retrieving temporal data, two time dimensions are usually incorporated in the database: *transaction time* and *valid time*. Depending on the time dimensions the database supports, temporal databases can be categorized as valid-time (or historical) databases, transaction-time (or rollback) databases, and bi-temporal databases [6]. Valid-time databases incorporate only the valid-time dimension which records the history of the information in the real world. Any error corrections and plan changes can be made by modifying the values of temporal data and/or the timestamps. However, after the modification, the previous values will not be retained and thus the database cannot rollback to the state before the change. Transaction-time databases incorporate only the transaction-time dimension which records the history of the database activities. It is impossible to make changes to past information. Any changes can only apply to the current data versions. Bi-temporal databases incorporate both transaction-time and valid-time dimensions. Although this leads to greater complexity, it allows the generality of both retroactive and proactive changes.

We now briefly define the bi-temporal data model adopted in this paper:
A bi-temporal relation schema R is defined as follows,

$$R = < ID, A_1, A_2, \ldots, A_n, VS, VE, TS, TE >,$$

in which $\{ A_1, A_2, \ldots, A_n \}$ is the union of the time-varying and fixed-value attributes. Attribute ID is a system generated entity identifier which has the same value for all versions of a particular entity. Time in this model is assumed to be discrete and described as consecutive nonnegative integers. Accordingly, time attributes [*VS, VE]*, and [*TS, TE*] are atomic-valued timestamped attributes containing a starting and ending valid-time point and a starting and ending transaction-time point, repectively. The domain of transaction-end time includes a time variable *UC* (Until Change) [18] and the domain of valid-end time includes time variable *now* [2]. The valid-time interval[1] [*VS, VE*] consisting of valid-time points associated with each tuple means the entity version in the tuple is valid in the modeled reality from *VS* to *VE*. The transaction-time interval [*TS, TE*] associated with each tuple defines when the tuple is logically existing in the database. The tuples that have *UC* as the value of transaction-end time (*TE*) represent the *current versions* of an entity. If the value of valid-end time of some current version of an entity is the time variable *now*, these current entity versions are valid currently and also are valid in the future until some change occurs.

A tuple is *logically deleted* by replacing the value of transaction-end time *UC* with the current time. If a tuple is modified, the tuple version will first be logically deleted and then a new version is inserted with the new attribute values and with *UC* as the value of *TE*.

An example of a bi-temporal database is shown in Figure 1. This is a bi-temporal catalog for maintaining schema information concerning the database relations and attributes. Such a bi-temporal catalog needs to be created

[1] This concept of a fixed time interval is now called a time period in SQL.

to maintain the history of schema changes and to be able to update the schema both retroactively and proactively. Figure 1a is defined for single table version approach, and Figures 1b and 1c are defined for multiple table version and partial multiple table version approaches respectively.

(a) **Relation**

ID	Name	VS	VE	TS	TE

Attribute

ID	Name	Domain	Rel	VS	VE	TS	TE

(b) **Relation**

ID	Name	Version	Derived_From	VS	VE	TS	TE

Attribute

ID	Name	Domain	Rel	Rel_Version	VS	VE	TS	TE

(c) **Relation**

ID	Name	Attrbute_of	VS	VE	TS	TE

Attribute

ID	Name	Domain	Rel	VS	VE	TS	TE

Figure 1. (a) Catalog relations for single table version approach. (b) Catalog relations for multiple table version approach. (c) Catalog relations for partial multiple table version approach.

Although a full discussion of catalog implementation is outside the scope of this paper, we would like to briefly discuss the relationship between catalog and database. The catalog itself can be considered a bitemporal database[2] to provide the full generality of schema changes. The bi-temporal database must be changed to conform to the schema changes as stored in the catalog. If both catalog and database are bi-temporal, the consistency between them can be maintained since both will allow retroactive and proactive changes. For the remainder of the paper, we will concentrate only on the database changes (not the catalog).

In this paper we will use the following example to explain and compare the three versioning techniques. For simplicity, the example only shows the versions of one entity in the relation.

Example 1 Assumes that the bi-temporal relation Employee is created at time 10 with valid-time interval [10, *now*] and with the attributes employee ID, name, salary, and position. Figure 2 shows the current state of the relation at time 40, for the versions of Employee 'John'.

[2] Although in many cases, a transaction-time database may be sufficient for the catalog implementation.

Employee

	ID	Name	Salary	Position	VS	VE	TS	TE	
t1	1	John	30k	P1	10	30	10	20	
t2	1	John	30k	P1	10	20	20	UC	*
t3	1	John	35k	P2	20	50	20	35	
t4	1	John	35k	P2	20	30	35	UC	*
t5	1	John	40k	P3	30	65	35	40	
t6	1	John	40k	P3	30	60	40	UC	*
t7	1	John	45k	P4	60	80	40	UC	*

Figure 2. The state of the Employee relation at time 40. The tuples with '*' are the current versions.

Assume that the following three schema changes are applied to the relation Employee:

*SC*1: At time 50, a new time-varying attribute Bonus is added to Employee, which is valid from time 25 to 65, and John's bonus is recorded as 5% and is valid during [25,65].

SC2: At time 60 another time varying attribute Phone is added to Employee, which is valid during [15,55], and the value of Phone for employee John is 3334567 with valid-time interval [15,55].

*SC*3: At time 70, attribute Salary is dropped from Employee from time 15 to 50.

3 Single Table Version approach

In single table version, each table has only one version throughout the lifetime of the database. This idea is proposed in [11] as *complete schemata*, which follows the idea of *complete table* in [3]. A complete schema consists of tables defined over the union of attributes that have ever been defined for them, each with the least general domain which can include all the domains' values. In cases where a general domain cannot be used, it is necessary to duplicate the attribute with enough domains to hold all necessary values. For instance, if attributes are added to a relation or the domain size of the attributes is enlarged, the table needs to allocate more space to accomodate the new attribute or the change of the domain. However, if attributes are dropped, the dropped attribute will still be retained in the database since in append-only temporal databases data will never be deleted. Therefore the record size, and hence the table size for this approach will only grow but never shrinks.

After the schema update operation *SC*1, which adds a Bonus attribute, is applied at time 50. The state of the catalog (which also consists of bi-temporal tables) is shown in Figure 3.

Relation_catalog

ID	Name	VS	VE	TS	TE
1	Employee	10	now	10	UC

Attribute_catalog

ID	Name	Domain	Rel_ID	VS	VE	TS	TE
1	ID	string	1	10	now	10	UC
2	Name	string	1	10	now	10	UC
3	Salary	real	1	10	now	10	UC
4	Position	string	1	10	now	10	UC
5	Bonus	real	1	25	65	50	UC

Figure 3. The catalog status at time 50 after the attribute Bonus is added

The new Bonus attribute has a default value 5% which is valid from time 25 to 65. However, before time 50 there is no value for Bonus, so the unknown value *null* must be assigned, as shown in Figure 4(a). In Figure 1, we notice that three of the four current entity versions of employee 1 have their valid-time interval overlap with the valid-time interval [25,65] of the 5% value of Bonus. For the entity versions whose valid-time intervals *partially overlap* (P) with [25,65] (namely t_4 with [20,30] and t_7 with[60,80]), two new tuples need to be derived from each version: these are t_8 and t_9 from t_4, and t_{11} and t_{12} from t_7. For the entity versions whose valid-time is *included* (I) in [25,65] (namely t_6 with [30,60]), one new tuple needs to derived from each version: this is t_{10} from t_6. The newly derived tuples are inserted as shown in Figure 4(b). The transaction-end time of the overlapped versions is changed from *UC* to the time when the new attribute value is assigned, which is time 50.

To continue the example, the effect of applying schema change operation *SC*2 (adding a Phone attribute) is shown in Figure 5. There are 2 current entity versions whose valid-time intervals are partially overlapping, and two others fully included in the valid-time interval of *SC*2. Therefore, six new tuples need to be inserted. As in *SC*1, the transaction-end time of the overlapping versions needs to be changed to the current time, which is 60 in this example.

Employee

	ID	Name	Salary	Position	Bonus	VS	VE	TS	TE	
t_1	1	John	30k	P1	null	10	30	10	20	
t_2	1	John	30k	P1	null	10	20	20	UC	
t_3	1	John	35k	P2	null	20	50	20	35	
t_4	1	John	35k	P2	null	20	30	35	~~UC~~ 50	P
t_5	1	John	40k	P3	null	30	65	35	40	
t_6	1	John	40k	P3	null	30	60	40	~~UC~~ 50	I
t_7	1	John	45k	P4	null	60	80	40	~~UC~~ 50	P

Part (a)

t_8	1	John	35k	P2	null	20	25	50	UC
t_9	1	John	35k	P2	5%	25	30	50	UC
t_{10}	1	John	40k	P3	5%	30	60	50	UC
t_{11}	1	John	45k	P4	5%	60	65	50	UC
t_{12}	1	John	45k	P4	null	65	80	50	UC

Part (b)

Figure 4. The state of relation Employee at time 50 after the value of Bonus is assigned. Part (a) shows the relation state before the attribute value is assigned. Part (b) shows the new inserted tuples after value for attribute Bonus is assigned.

Employee

ID	Name	Salary	Position	Bonus	Phone	VS	VE	TS	TE	
1	John	30k	P1	null	null	10	30	10	20	
1	John	30k	P1	null	null	10	20	20	~~UC~~ 60	P
1	John	35k	P2	null	null	20	50	20	35	
1	John	35k	P2	null	null	20	30	35	50	
1	John	40k	P3	null	null	30	65	35	40	
1	John	40k	P3	null	null	30	60	40	50	
1	John	45k	P4	null	null	60	80	40	50	
1	John	35k	P2	null	null	20	25	50	~~UC~~ 60	I
1	John	35k	P2	5%	null	25	30	50	~~UC~~ 60	I
1	John	40k	P3	5%	null	30	60	50	~~UC~~ 60	P
1	John	45k	P4	5%	null	60	65	50	UC	
1	John	45k	P4	Null	null	65	80	50	UC	
1	John	30k	P1	null	null	10	15	60	UC	
1	John	30k	P1	null	3334567	15	20	60	UC	
1	John	35k	P2	null	3334567	20	25	60	UC	
1	John	35k	P2	5%	3334567	25	30	60	UC	
1	John	40k	P3	5%	3334567	30	55	60	UC	
1	John	40k	P3	5%	null	55	60	60	UC	

Figure 5. The state of Employee relation after time 60.

The effect of *SC*3 is shown in Figure 6. As the figure shows, five more tuples are inserted after the change as there are 3 fully including and 1 partially overlapped current versions.

Employee

ID	Name	Salary	Position	Bonus	Phone	VS	VE	TS	TE	
1	John	30k	P1	Null	Null	10	30	10	20	
1	John	30k	P1	Null	Null	10	20	20	60	
1	John	35k	P2	Null	Null	20	50	20	35	
1	John	35k	P2	Null	Null	20	30	35	50	
1	John	40k	P3	Null	Null	30	65	35	40	
1	John	40k	P3	Null	Null	30	60	40	50	
1	John	45k	P4	Null	Null	60	80	40	50	
1	John	35k	P2	Null	Null	20	25	50	60	
1	John	35k	P2	5%	Null	25	30	50	60	
1	John	40k	P3	5%	Null	30	60	50	60	
1	John	45k	P4	5%	Null	60	65	50	UC	
1	John	45k	P4	Null	Null	65	80	50	UC	
1	John	30k	P1	null	null	10	15	60	UC	
1	John	30k	P1	null	3334567	15	20	60	~~UC~~ 70	I
1	John	35k	P2	null	3334567	20	25	60	~~UC~~ 70	I
1	John	35k	P2	5%	3334567	25	30	60	~~UC~~ 70	I
1	John	40k	P3	5%	3334567	30	55	60	~~UC~~ 70	P
1	John	40k	P3	5%	null	55	60	60	UC	
1	John	null	P1	null	3334567	15	20	70	UC	
1	John	null	P2	null	3334567	20	25	70	UC	
1	John	null	P2	5%	3334567	25	30	70	UC	
1	John	null	P3	5%	3334567	30	50	70	UC	
1	John	40k	P3	5%	null	50	55	70	UC	

Figure 6. The state of Employee relation after time 70.

There are three problems that need to be addessed if the single table version approach is adopted: **space overhead**, **search overhead**, and **database availability**. The problem of space overhead results from excessive *data duplication* and *null values* whenever the database is converted to conform to the schema change. Data duplication is a common problem in temporal databases if

the relation is not in temporal normal form [10], i.e., the attributes in a relation do not change their values synchronously. This is the case here, since every time an attribute is added to or dropped from the relation, all the current entity versions whose valid-time interval overlaps with the valid-time interval of the schema change have most of their attributes duplicated. As shown in the examples, the information of employee's ID, name, and position are duplicated in all the newly inserted tuples. In a database with thousands of records, this situation causes a large amount of duplication. This problem not only wastes storage space but also makes the temporal JOIN operation [5,16,20,21] more complicated.

Another space overhead for this approach is the space of *null* values in the old tuple versions after the schema change of attribute addition, as shown in Figures 4 and 5(b). Again, if there are thousands of records in the relation at the time of attribute addition, it will introduce a large quantity of null values. This will not only waste space at the storage level but also lead to the problems with understanding the meaning of the attributes and with specifying JOIN operation at the logical level. Moreover, nulls can have multiple conflicting interpretations:

1. The attribute is valid but its value is unknown.
2. The attribute is valid and its value is known but not recorded yet.
3. The attribute is invalid.

In the case of the example, the third interpretation of *null* above is the correct one.

Searching for current versions of entities that must be changed because of a schema change can be expensive. These are the versions whose valid time overlaps with the valid time of the schema change. As a result, in the single table version approach, every time an attribute is added to or dropped from a relation, all current versions need to be checked to see if their valid time interval overlaps with the schema change. According to [17]'s measurement, the number of attributes in the commercial application used for their experiment increased 274% in only 18 months. For temporal databases, which usually contain a large amount of data, the search time will be a large overhead.

In addition to the space and time overhead, database availability is another concern. When a new attribute is added or the type domain of an attribute is generalized, part of (or even the whole) database will not be available for a period of time due to the process of database conversion, which requires augmentation and reorganization of the storage space.

4. Multiple table version

In multiple table version, every time a relation schema is changed, it creates a new table version. Data inserted to the relation after the change will be stored in the newly created table. For conventational nontemporal databases or transaction-time databases, since schema updates can only be applied to the current schema version, the implementation of multiple table version is less complicated than for bi-temporal databases, where schema and database both can be updated retroactively and proactively. We use the following two figures, 7 and 8, to illustrate the difference between bi-temporal and transaction time databases for this problem.

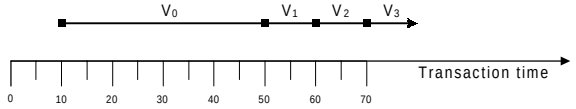


Figure 7. Multiple versioning in the transaction-time relation Employee

For transaction-time databases[3], as shown in Figure 7, there always exists only one current schema version at any time point. Any schema change will create only one new version which is derived from the current version. However, in a bi-temporal database, at a certain time point, there may exist more than one current version defined for different valid-time intervals, as illustrated by the following example.

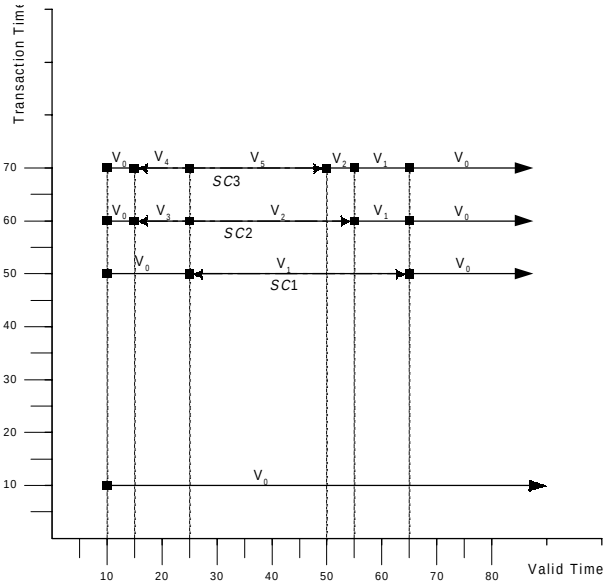


Figure 8. Multiple versioning in the bi-temporal relation Employee

In Figure 8, we see that version V_0 of the relation Employee is defined at (transaction) time 10 and is valid from time 10 to *now*. At time 50, *SC*1 is executed. Since there is only one current version, V_0, that overlaps

[3] We assume that the schema versions in nontemporal databases are identified by the transaction time when they are defined, and thus can also be categorized as transaction-time databases.

[25,65], only one new version V_1 is created. Now at time 50 after *SC*1, we have two current table versions, version V_0 which is valid during [10,25] and [65,now], and version V_1 which is valid during [25,65]. At time 60, *SC*2 is executed. However, since the effective time of *SC*2 is from 15 to 55, it overlaps with both versions V_0 and V_1. Two new versions, V_2 and V_3, need to be created where V_2 is derived from V_1 and V_3 is derived from V_0. SC3 is executed at time 70 and it affects versions V_2 and V_3. Therefore, two new versions V_5 and V_4 are derived from source versions V_2 and V_3 respectively. We will call table version V*i* a source to a later table version V*j* if V*j* is derived from V*i* because of overlapping valid time intervals.

After new table versions are created, the current entity versions in the source table whose valid-time interval overlaps with the schema change need to be copied into the newly created table along with the value of the new attribute (if the schema change is attribute addition).

To illustrate the problems that may occur, we first give the algorithms for new table version creation and data insertion to the newly created table versions for both attribute addition and attribute deletion.

Attribute addition

At time *t*, a new attribute A_x is added to table *R* with value $a_{x,i}$ for entity *i*. The valid lifespan of the schema change is I_{vsc}.

Create new table versions

The parameters:
v : table version number.
I_{vsc}: a temporal element which is the valid lifespan of the schema change.
I_{vv} : a temporal element which is the valid lifespan of table version R_v.
cur : latest table version number.

$v = cur$;
AddAttribute(A_x, v, I_{vsc})
// compare I_{vv} with I_{vsc}
if $I_{vv} \supseteq I_{vsc}$ // I_{vsc} included in I_{vv}
then $cur = cur + 1$;
create new table version R_{cur} by adding A_x to R_v with valid lifespan I_{vsc};

else if $I_{vv} \cap I_{vsc} = \varnothing$
then $v = v - 1$;
AddAttribute(A_x, v, I_{vsc}); // recursive call to the older versions
else if $I_{vv} \cap I_{vsc} = I_{vy}$ // I_{vy} is temporal element of the intersection
then $cur = cur + 1$;
create new table version R_{cur} by adding A_x to R_v with valid lifespan I_{vy};
$I_{vsc} = I_{vsc} - I_{vy}$;
$v = v - 1$;
AddAttribute(A_x, v, I_{vsc}); // recursive call to older versions

After the vew table versions are created, data in the source table versions need to be converted to the newly created table versions with the value of the new added attribute A_x.

Inserting tuples in the newly created table versions

For each newly created table version $R_{new,v}$ after adding the new attribute A_x

1. Identify the source table version R_v from which $R_{new,i}$ is derived.
2. In R_v, find a set of current versions S_i for each entity *i*, $S_i = \{V_{i,1}, V_{i,2}, \ldots, V_{i,n}\}$.
3. For each entity, compare the valid-time interval, $I_{i,k}$, of each current entity versions $V_{i,k}$ in S_i with I_{vsc}.
   If $I_{i,k} \cap I_{vsc} = [VT_1, VT_2]$
   then insert $V_{i,k}$ as a new tuple into table version $R_{new,v}$ along with the new attribute value $a_{x,i}$, valid-time interval $[VT_1, VT_2]$, transaction start-time *t'*, and transaction end-time UC.

Figures 8 and 9 show the newly created table versions and data converted from the source versions after *SC*1 and *SC*2 are executed. In Figure 9, after *SC*1, version V_1 is created, the data are converted from the current entity versions marked with '*' in table version V_0. Versions V_2 and V_3 are created after *SC*2 is executed. Version V_2 is derived from V_1 and its data is converted from entity versions marked with '†' in table version V_1. Version V_3 is derived from V_0 and its data is converted from entity versions marked with '✓' in table version V_0.

(a) **Employee_V$_0$** (Valid lifspan: $I_{v0} = [10,now]$)

ID	Name	Salary	Position	VS	VE	TS	TE		
1	John	30k	P1	10	30	10	20		
1	John	30k	P1	10	20	20	UC		✓
1	John	35k	P2	20	50	20	35		
1	John	35k	P2	20	30	35	UC	*	✓
1	John	40k	P3	30	65	35	40		
1	John	40k	P3	30	60	40	UC	*	
1	John	45k	P4	60	80	40	UC	*	

(b) **Employee_V$_1$** (Valid lifspan: $I_{v1} = [25,65]$)

ID	Name	Salary	Position	Bonus	VS	VE	TS	TE	
1	John	35k	P2	5%	25	30	50	UC	†
1	John	40k	P3	5%	30	60	50	UC	†
1	John	45k	P4	5%	60	65	50	UC	

(c) **Employee_V$_2$** (Valid lifespan I_{v2}= {[25,55]})

ID	Name	Salary	Position	Bonus	Phone	VS	VE	TS	TE
1	John	35k	P2	5%	3334567	25	30	60	UC
1	John	40k	P3	5%	3334567	30	55	60	UC

Employee_V$_3$ (Valid lifespan I_{v3}= {[15,25]})

ID	Name	Salary	Position	Phone	VS	VE	TS	TE
1	John	30k	P1	3334567	15	20	60	UC
1	John	35k	P2	3334567	20	25	60	UC

Figure 9.(a) The original table. (b) After *SC*1. (c) After *SC*2.

Attribute deletion

At *t*, attribute A_x is dropped from relation R during I_{vsc}.

Create new table versions

Find a list of table versions L_R in which all the versions include attribute A_x.

The list is ordered by the version number in descending order.

DropAttribute(A_x, I_{vsc}, L_R)

The first version R_i in L_R has valid lifespan I_{vi};
// compare I_{vi} with I_{vsc}
if $I_{vi} \supseteq I_{vsc}$ // I_{vsc} included in I_{vv}
then $cur = cur + 1$;
create new table version R_{cur} by dropping A_x from R_i with valid lifespan I_{vsc};
else if $I_{vi} \cap I_{vsc} = \varnothing$
then $L_R = L_R - R_i$;
DropAttribute (A_x, I_{vsc}, L_R); // recursive call to older versions
else if $I_{vi} \cap I_{vsc} = I_{vy}$ //I_{vy} is the temporal element of the intersection
then $cur = cur + 1$;
create new table version R_{cur} by dropping A_x from R_i with vlaid lifespan I_{vy};
$I_{vsc} = I_{vsc} - I_{vy}$;
$L_R = L_R - R_i$;
DropAttribute (A_x, I_{vsc}, L_R); // recursive call to older versions

Inserting tuples in the newly created table versions

For each newly created table version $R_{new,i}$ after dropping the attribute A_x

1. Identify the source table version R_v from which $R_{new,i}$ is derived.
2. In R_v, find a set of current versions S_i for each entity i, $S_i = \{V_{i,1}, V_{i,2}, \ldots, V_{i,n}\}$.

// insert tuples to new table versions

3. For each entity, compare $I_{i,k}$, the valid-time interval of each current entity version in S_i, with I_{vsc}.
   If $I_{i,k} \cap I_{vsc} = [VT_1, VT_2]$
   then insert $V_{i,k}$ as a new tuple into table version $R_{new,v}$ with valid-time interval $[VT_1, VT_2]$, transaction start-time *t*, transaction end-time UC, and without the dropped attribute A_x.

Figures 8 and 10 show the newly created table versions and data converted from the source versions after *SC*3 is executed. In Figure 10, versions V_4 and V_5 are created after *SC*3 is executed . Version V_4 is derived from V_3 and its data is converted from entity versions marked with '✓' in table version V_4. Version V_5 is derived from V_2 and its data is converted from entity versions marked with '*' in table version V_2.

(a)**Employee_V$_2$** (Valid lifespan I_{v2}= {[25,55]})

ID	Name	Salary	Position	Bonus	Phone	VS	VE	TS	TE	
1	John	35k	P2	5%	3334567	25	30	60	~~UC~~ 70	*
1	John	40k	P3	5%	3334567	30	55	60	~~UC~~ 70	*
1	John	40k	P3	5%	3334567	50	55	70	UC	

Employee_V$_3$ (Valid lifespan I_{v3}= {[15,25]})

ID	Name	Salary	Position	Phone	VS	VE	TS	TE	
1	John	30k	P1	3334567	15	20	60	~~UC~~ 70	✓
1	John	35k	P2	3334567	20	25	60	~~UC~~ 70	✓

(b)**Employee_V$_4$** (Valid lifespan I_{v4}= {[15,25]})

ID	Name	Position	Phone	VS	VE	TS	TE
1	John	P1	3334567	15	20	70	UC
1	John	P2	3334567	20	25	70	UC

Employee_V$_5$ (Valid lifespan I_{v5}= {[25,50]})

ID	Name	Position	Bonus	Phone	VS	VE	TS	TE
1	John	P2	5%	3334567	25	30	70	UC
1	John	P3	5%	3334567	30	50	70	UC

Figure 10. (a) The table versions before *SC*3. (b) After *SC*3.

From the algorithms and the example above, we can see that the MTV approach does not have the problems of *null* value and database availability as in the approach of STV. However, there are still some problem with this approach: **data duplication**, **multischema queries** [4], and **mandatory version creation**.

Although the MTV approach does not introduce null values, it still has the problem of data duplication. As we can see from the example, if the current entity versions in the source table versions have valid-time interval that overlaps with the schema change, the unchanged attributes need to be duplicated and inserted into the new table versions.

The MTV resolves the problem of database reorganization in STV resulting from attribute addition by creating new table versions. However, the trade-off is the number of table versions. As shown in Figure 8, in bi-temporal databases, the schema modification is not limited to the latest table version as in the transaction-time databases. Therefore, the number of newly created versions may be more than one depending on the valid-time interval of the schema change. As the number of table versions increases, more temporal JOIN operations will be needed to process queries. For example, suppose that at time *t*, an attribute *A* is added to table *R* and the

effect of this schema change creates three new table versions of *R* at time *t*. Later if a query requires a join operation between table *R* and *S* through attribute *A* as of time *t*. Then table *S* needs to be joined with three different table versions of *R*. Another concern is querying the entity's history. The system has to access all the different table versions to get the information. As a result, the output of the query includes different tuple types which makes it difficult to use by applications. This is called the problem of *multischema queries* [4].

Another problem of the MTV approach is *mandatory version creation*. Let us consider the schema change *SC*1. In *SC*1, the valid lifespan of the newly added attribute Bonus is from time 25 to 65. At time 45, a new table version V_1 is created with attribute Bonus. However, if the current time becomes 66, the attribute Bonus is not valid any more. A new table version must be created at this time to maintain the system consistency. This extra work must either be done by the user or by specifying appropriate triggers.

5. Partial multiple table version

From the example and discussion in the previous sections, we notice that the complexity of the schema change is highly dependent on the current state of the database. In this section, we propose the *partial multiple table version* (PMTV) approach which can make the updates independent from the current state of the database and thus can largely reduce the complexity and also solve the problems with STV and MTV.

For the schema change of attribute addition, instead of enlarging the record size required for the extra storage as in STV or creating a new table version as in MTV, PMTV create a bi-temporal relation with only the new attribute, plus the key attribute (ID) of the relation being modified[4]. The complete relation can be later reconstructed by applying the *temporal NATURAL JOIN* operation[5] [20,21]. Techniques for efficient execution of this operation are discussed in [20,21], but they apply to valid-time only databases. For our application, in fact, the type of join needed is more like a TEMPORAL OUTER JOIN. We are currently working on efficient techniques for this join for bi-temporal databases, since this is crucial to the partial multiple table version technique.

Suppose at time *t*, attribute A_x is added to relation *R* with valid-time interval $[vt_1,vt_2]$. A new relation R_A_x is created. Following the data model defined in section 2, we have the following schema:

$R = < ID, A_1, A_2, \ldots, A_n, VS, VE, TS, TE >$

$R\_A_x = < ID, A_x, VS, VE, TS, TE >$

[4] This is similar to the technique of vertical fragmentation in distributed database systems.

[5] This has also been called temporal intersection join.

Applying this approach to the example given in Section 2 after *SC*1 and *SC*2 are executed, the results are shown in Figure 11.

Employee

ID	Name	Salary	Position	VS	VE	TS	TE
1	John	30k	P1	10	30	10	20
1	John	30k	P1	10	20	20	UC
1	John	35k	P2	20	50	20	35
1	John	35k	P2	20	30	35	UC
1	John	40k	P3	30	65	35	40
1	John	40k	P3	30	60	40	UC
1	John	45k	P4	60	80	40	UC

(a) Original Employee relation

Emp_Bonus

ID	Bonus	VS	VE	TS	TE
1	5%	25	65	50	UC

(b) New created relation for attribute Bonus after *SC*1

Emp_Phone

ID	Phone	VS	VE	TS	TE
1	3334567	15	55	60	UC

(c) New created relation for attribute Phone after *SC*2

Relation_catalog

ID	Name	Attrbute_of	VS	VE	TS	TE
1	Employee	null	10	now	10	UC
2	Emp_Bonus	1	25	65	50	UC
3	Emp_Phone	1	15	55	60	UC

(d) The state of Relation catalog after *SC*1 and *SC*2

Figure 11. Applying *SC*1 and *SC*2 using partial multiple table version.

For the schema change of attribute deletion, there are two different cases: the dropped attribute is in the original relation or it was added later.

1. If the dropped attribute A_x is in the original table *R*, then the process is the same as single table version approach.
2. If A_x is an attribute that was added later, then apply the process for single table version approach for table R_A_x only.
   - 2.1 In table R_A_x, for each entity *i*, find a set of current versions S_i where
     $S_i = \{V_{i,1}, V_{i,2}, \ldots, V_{i,n}\}$
   - 2.2 For each entity *i*, compare I_{vsc} with $I_{i,k}$, the valid-time interval of each current entity version in S_i
     If $I_{i,k} \cap I_{vsc} = I_{vy}$ then
     insert $V_{i,k}$ into R_A_x with valid-time interval I_{vsc}–I_{vy};
     TS = *t*; TE = UC;
     change TE of $V_{i,k}$ from UC to *t*;

The following figure shows the result of *SC*3:

Employee

ID	Name	Salary	Position	VS	VE	TS	TE	
1	John	30k	P1	10	30	10	20	
1	John	30k	P1	10	20	20	~~UC~~ 70	P
1	John	35k	P2	20	50	20	35	
1	John	35k	P2	20	30	35	UC	I
1	John	40k	P3	30	65	35	40	
1	John	40k	P3	30	60	40	~~UC~~ 70	P
1	John	45k	P4	60	80	40	~~UC~~ 70	
1	John	30k	P1	10	15	70	UC	
1	John	null	P1	15	20	70	UC	
1	John	null	P2	20	30	70	UC	
1	John	null	P3	30	50	70	UC	
1	John	40k	P3	50	60	70	UC	

Now if the schema is changed again at time 75 that the attribute Bonus is dropped from time 30 to 60. The result will be:

Employee_Bonus

ID	Bonus	VS	VE	TS	TE	
1	5%	25	65	50	~~UC~~ 75	P
1	5%	25	30	75	UC	
1	5%	60	65	75	UC	

Employee_Phone

ID	Phone	VS	VE	TS	TE
1	3334567	15	55	60	UC

As we can see from the example, for the schema change of attribute addition, there is no space needed for null values and data duplication. Searching for the current versions of entities and the valid timespan of their versions is not necessary. Therefore, this approach largely reduces the space and time complexity. Futhermore, the problems of database reorganization in the single table version approach and the mandatory version creation in the multiple table version approach do not exist in this approach. For the schema change of attribute deletion, although in one case the process of database conversion is the same as in the STV approach except that it may be applied to smaller tables. However, most of the attribute change are attribute addition according to [17]'s measurements. If the attribute to be deleted was an added attribute, as in the example above, it is much simpler than the other two approaches.

6. Comparison of three approaches

In this section, we compare the space complexity of the three versioning approaches. Because of the space limitation, we define the cost formulas and only show the results based on the given example. From the discussion in the previous sections, we can see that the database conversion for schema changes of attribute addition and attribute deletion is nearly the same. Here we only discuss the space complexity of adding attributes. The following are the definitions of the parameters:

Parameters

S_a : average attribute size.

I_{vsc} : a temporal element which is the valid lifespan of the schema change.

N_a : number of attributes in original relation R.

N_{tp} : number of tuples in table R before any schema changes.

N_e : average number of distinct entities in relation R.

N_O : average number of current versions of all the entities whose valid-time intervals overlap with I_{vsc}. where $N_O = N_P + N_F$

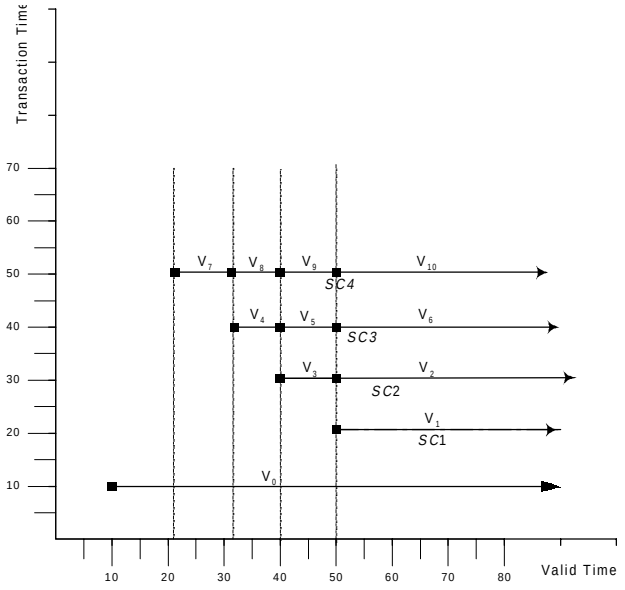
N_P : average number of current versions of all the entities whose valid-time intervals partially overlap with I_{vsc}.

N_F : average number of current versions of all the entities whose valid-time intervals included in I_{vsc}.

N_{add}: the number of attribute addition operations.

Assumptions

1. For the approach of MTV, we use the worst case scenario when computing the number of newly created table versions. That is, the valid-time interval of the next schema change covers the valid time of the previous schema change, as shown in the following figure:



Therefore, a new schema change creates one more table version than the number of table versions created from the previous schema change.

2. Since the three approaches were compared using the same example, the value of N_O (number of overlapped entity versions) is the same in STV and MTV. For MTV, we assume the newly created entity

versions are evenly distributed into each of the newly created table versions.

Single Table Version approach

From Section 3, the space required for attribute addition by STV is the space for the null values S_{null} and for the duplicated attributes S_{dup}.

$$S_{null} = (S_a * N_{tp}) \quad \text{// the space for null after the first attribute addition}$$
$$\text{// the space for null after the 2nd attribute addition}$$
$$+ S_a * [N_{tp} + (2N_P + N_F) * N_e]$$
$$+ S_a * [N_{tp} + (2N_P + N_F)* N_e + (2N_P + N_F) * N_e]$$
$$+ \dots$$
$$= S_a * (N_{add} * N_{tp} + (2N_P + N_F) * N_e * \sum_{i=0}^{N_{add}-1} i)$$

$$S_{dup} = S_a * N_a * (2N_P + N_F) * N_e + S_a * (N_a + 1) * (2N_P + N_F) * N_e + S_a * (N_a + 2) * (2N_P + N_F) * N_e + \dots = Sa * \left[N_{add} * N_a + \sum_{i=0}^{Nadd-1} i \right] * \left[(2N_P + N_F) * N_e \right]$$

$$S_{STV} = S_{null} + S_{dup}$$

Multiple Table Version approach

From Section 4, the space required for attribute addition by MTV is for the overlapped data version duplicated in the new table versions.

$$S_{MTV} = S_a * (N_a + 1) * (N_O * N_e) + S_a * \left[(N_a + 1) + (N_a + 2) \right] * \frac{N_O}{2} * N_e + \dots = S_a * \sum_{i}^{Nadd} \left[\left(\sum_{j=1}^{i} (N_a + j) \right) * \frac{N_O}{i} * N_e \right]$$

Partial Multiple Table Verison approach

From Section 5, the space required for attribute addition by PMTV is only the size of two attributes for each entity.

$$S_{PMTV} = N_{add} * [(S_a * 2) * N_e]$$

Now we can compare the space cost for these three approaches based on the given example: N_a is 4, N_{tp} is 7, N_e is 1, and N_{add} is 2. We also have N_P for $SC1$ is 2 and N_F is 1, and N_P and N_F for $SC2$ are both equal to 2.

$$S_{null} = S_a * 7 + S_a * (7 + (2*2 + 1) * 1) = 19S_a$$

$$S_{dup} = S_a * 4 * (2*2 + 1)*1 + S_a * 5 * (2*2 + 2)*1 = 50S_a$$
$$S_{STV} = 69S_a$$

$$S_{MTV} = S_a*((4+1) * (2+1) * 1)+S_a*[(4+1) * 2 * 1] + Sa*[(4+2) * 2 * 1] = 37S_a$$
$$S_{PMTV} = 2 * [(Sa * 2) * 1] = 4S_a$$

For this example, there is a factor of 17 difference between the approaches of STV and PMTV. Since the space required for the PMTV approach is independent from the number of the attributes, tuples, entities, and the overlapping entity versions. If we have real a bi-temporal database with reasonable number of entities, tuples, and overlapped entity versions, the difference obviously will become much larger. Let us consider another simple example:

A relation R originally has 5 attributes with average size of 8 bytes, and 15 tuples including 3 entities, i.e., $N_a = 5$, $S_a = 8$, $N_{tp} = 15$, and $N_e = 3$. If later 7 attributes are added to R, $N_{add} = 7$, and we assume N_P and N_F are both equal to 2. From the result, we found 40 times difference between STV and PMTV and 30 times difference between MTV and PMTV.

$$S_{STV} = S_{null} + S_{dup} = 8 * (7 * 15 + (2*2 + 2) * 3 * \sum_{i=0}^{6} i) + 8 * (7 * 5 + \sum_{i=0}^{6} i) * ((2*2 + 2) * 3) = 3864 + 8064 \approx 12KB$$

$$S_{MTV} = 8* \sum_{i=1}^{7} \left(\sum_{j=1}^{i} (5+ j) * \frac{4}{i} * 3 \right) \approx 9KB$$

$$S_{PMTV} = 7 * (8 * 2) * 3 \approx 0.3KB$$

7 Conclusion

In this paper, we present the study of two schema versioning approaches in a bi-temporal database environment, single table version and multiple table version. In most of the current literature, only transaction time is considered on schema versioning. The research that discusses schema versioning involving both transaction time and valid time does not consider some of the more complex problems concerning schema version creation and database conversion. We first discuss the problems associated with these two approaches when

applied to bi-temporal databases, then propose a third approach, partial multiple table version, which makes the database conversion much simpler and does not have the problems found in the previous two approaches. Furthermore, we also specify formulas to analyze and compare the space cost for the three approaches.

For our proposed partial multiple table version approach, when a new attribute is added, it creates a new bi-temporal relation with only the new attribute, plus the key attribute of the relation being modified. This way, no null values will be introduced, no searching for the overlapped current versions is needed, no database restructuring and data duplication is required, and no extra effort is needed for the problem of mandatory version creation. In addition, from section 6, compared with the two previous approaches, the space cost has been largely reduced.

We are currently analyzing the costs of various types of temporal queries when applied to the three methods discussed here.

References

[1] J. Banerjee, H-T Chou, H. J. Kim , and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-oriented databases. *SIGMOD RECORD*, 16(3):311-322, 1987.

[2] 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*, 22(2):171 – 214, June 1997.

[3] J. Clifford and D.S. Warren. Formal Semantics for Time in Databases. *ACM Transactions on Database Systems*, pages 214-254, 1983.

[4] Christina DeCastro, Fabio Grandi, and Maria Rita Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, pages 249-290, July 1997.

[5] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proceedings of 7th International Conference on Data Engineering, IEEE*, 1991.

[6] C. Jensen *et al.* A consensus glossary of temporal database concepts. SIGMOD RECORD, 23(1):52-64, 1994.

[7] W. Kim and H-T Chou. Versions of schema for object-oriented databases. In *Proceedings of the 14th International Conference on Very Large Databases*, pages 148-159, 1988.

[8] B. S. Lerner and A. N. Habermann. Beyond schema evolution to database reorganization. *SIGPLAN Notices*, 25(10):67 – 76, 1990.

[9] N.G. Martin, S. B. Navathe, and R. Ahmed. Dealing with Schema Anomalies in History Databases. In *Proceedings of the 13th International Conference on VLDB*, 1987.

[10] S. B. Navathe and R. Ahmed. A Temporal Relation Model and a Query Langue. *Information Sciences*, pages 147-175, 1989.

[11] J. F. Roddick. Dynamically Changing Schemas within Database Models. *Australian Computer Journal*, pages 105-109, 1991.

[12] J. F. Roddick. Schema Evolution in Database Systems – An Annotated Bibliography. Technical Report No. CIS-92-004, School of Computer and Information Science, University of South Australia, 1992.

[13] J. F. Roddick. SQL/SE – A Query Language Extension for Databases Supporting Schema Evolution. SIGMOD RECORD, pages 10-16, Sep. 1992.

[14] J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7), 1995.

[15] M. R. Scalas, A. Cappelli, and C. De Castro. A Model for Schema evolution in Temporal Relational Databases. In *Proceedings of 1993 CompEuro, Computers in Design, Manufacturing, and Production*, pages 223 – 231, May 1993.

[16] A. Segev. Join Processing and Optimization in Temporal Relational Databases. Chapter 15 of *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, 1993.

[17] D. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35 – 44, 1993.

[18] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, pages 247 – 298, June 1987.

[19] R. T. Snodgrass, editor. The TSQL2 Temporal Query Language, chapter 10. Kluwer Academic Publishers, 1995.

[20] M. D. Soo, R. T. Snodgrass and C. S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proceedings of the 10th International Conference on Data Engineering*, IEEE, 1994.

[21] D. Son and R. Elamsri. Efficient Temporal Join Processing Using Time Index. In *Proceedings of the 8th International Conference on Scientific and Statistical Database Management*, pages 252 – 261, June 18 – 20, 1996.

[22] M. Tresch and M. H. Scholl. Schema transformation without database reorganization. *SIGMOD RECORD*, 22(1):21 – 27, 1993.

[23] R. Zicari. A framework for schema updates in an object-oriented database system. In *Proceedings of the 7th International Conference on Data Engineering*, pages 2 – 13, April 1991.