Université libre de Bruxelles

Temporal Databases

Implementing a Multi-Version and a Multi-Temporal Database

Redioni Karakashi & Ahmet Anil Pala

Advanced Databases INFO-H-415

Mr. Esteban Zimányi

31 December 2013

TABLE OF CONTENTS

1 INTRODUCTION

1.1 BACKGROUND

1.1.1 Temporality

By using the term temporal we mean an abstract object which existed or exists over a specific time-period. Temporality rules concerns are on how we will define an abstract object into a time-interval and how we should handle that abstract object's changes over the time. A basic rule of temporality is that for each abstract object, there cannot exist more than one of it in the time-line, simultaneously. Additionally, we are assuming that we have only one time-line and that is linear.

1.1.2 Temporal Database

Continuing in the temporal logic, we will replace that abstract object with 2 specific objects, the temporal data existing in a database table and the temporal schema of that database table. The data(temporal or not) and the table schema's temporality have only one direct connection together, and that is that the data we insert in a specific time must agree with schema of the table in that time.

1.1.3 Multi-Version Databases

A today's database management system is constructed in a way to allow the user to make changes on the original schema of the database. The user in his convenience is allowed to rename objects, to add new objects in a database or to drop objects in a database. In our case, we are concerned only for the table object's such as the table attributes. Thus the user is allowed to change the table's schema over time as he wants but there is an aspect which is not yet covered by the today's DBMS: the over time aspect. Today a user may be able to change a table's schema but after this event, he has to continue using the last modified schema of the table, with a consequence the old version of the schema to be erased and not able to be retrieved for information purposes and/or to be used again. It is in this point in our progress in the today's DBMS technology that we have to introduce a Multi-Version DBMS which will keep a record of all the different schema versions a user has made over time. More specifcally, a Multi-Version DBMS allow the user to review all of his changes on the database's schema evolution and to go back in time and use the database from a previous schema state or version.

1.1.4 Multi-Temporal Databases

A common purpose of today's Database Management Systems is to provide space(or tables) where a user can store his data and also tools by which he can handle his data. Such tools are for inserting, deleting, updating and projecting data. The disadvantage of today's DBMS is that the specific tools which handle the user's data do not take into consideration the Temporality of the data but presume them as being constant over time despite multiple changes may happen on those data in the meanwhile. We call this type of table, by temporal terms, a Snapshot table(SN). For the lack of this functionality by a today's common DBMS, we need to introduce and/or implement a Multi-Temporal Database Management

System. In a Multi-Temporal Database, we are concerned for the Temporality of the actual data. In this Database type, we need to keep track of the changes of the actual data over time. In order to achieve this, we cannot delete any actual data from the table neither we can really update any existing data. For this reason, whenever there is an update, we instead perform a new insertion of those data, still keeping the old ones, and whenever there is a deletion of data, we instead just change the Temporal Ending Time of those data(or tuples).

1.2 MOTIVATION

In this project, we focused on implementing a database management API that functionally supports multi-temporal queries with multi-versioning support. Motivation for the idea of bringing two distinctly different paradigms exist in database world together was to provide a flexible temporal database environment that features both backward and forward compatibility. By means of such a platform that we made an effort to keep historical data and historical metadata as well (data that describes the format of the actual data, i.e a table schema) legacy applications can still be using the old data definitions with old data and also new data at the very same time. Moreover, new applications can reach the old data even though it is defined under an older version of schema.

Situations where this kind of a flexible database is useful such as correction of schema deficits in early version schemas, new user requirements, occur frequently in real-life. For example, keeping the historical data regardless of the schema evolution is an important issue (i.e auditing) or a database that number of legacy and modern applications both make transactions to and modify is needed where the importance of the backward/forward compatibility kicks in.

All that said, we were motivated enough to take on this project which by we can investigate the features and the implementation methods of these features of the multi-temporal databases with schema versioning.

2 IMPLEMENTATION

2.1 METHODOLOGY

2.1.1 General Structure

General application structure of our project implementation is composed of a programming application interface (API) which is specifically designed to manage multi-temporal and multi-version databases along with some SQL Server triggers that enforces some of the program logic to be implemented. We call the API we developed Temporal API. Temporal API will provide users with a number of functions that are specifically developed and designed to deal with temporal queries. This API sits on the top of the SQL Server and the catalogue tables (attribute and relation tables responsible for bookkeeping)

Behind the scenes, what is being used to store all the historical and current data of a table (regardless of how many times its schema is changed) is in fact only one table just as how a relation is stored in a classical database environment. In other words, for one relation, only one table exists. However, in order to deliver an illusion that makes user think there are separate tables exist for every schema version of each table, there are two additional so-called 'metadata' tables which are responsible for bookkeeping of different schema versions of every relation defined in the database. These are named as relation catalogue and attribute catalogue respectively. Relation catalogue keeps track of version number of the schemas for the tables and also the time interval in which the schema was valid (note that, only transactional time temporality is used, in other words different schema versions for a table cannot overlap) As for the other 'metatable' namely the attribute catalogue, as its name suggests, it records the attribute information of every attribute for each schema version appears in the relation catalogue.

The abovementioned illusion perceived from a higher level, the user or application that drives our API have the abstract view where there are different tables for different versions of a schema which is important for both legacy and modern applications sharing the old and new tuples without any problems, is in fact delivered by some complex interplay of catalogue tables, triggers defined to avoid illegal operations (see: Triggers defined, Assumptions) and program logic of temporal API functions.

Firstly, in order to accommodate all the data inserted under different schemas in a same table, how the schema change functions update the table and how the new data is inserted to the table should be handled with care. For the purpose of storing the historical data of historical attributes that are dropped in new versions of the schema, the table where the actual data is stored retains every attribute of the old version after addition or removal of some columns (except for the case of a schema update where the temporal dimension is upgraded or downgraded, in some cases, some temporal columns can be renamed. Conversion from Transaction time to the valid time causes this. However, this is an exceptional case along with the column renaming operator and should require elevated permissions from the upper level) with the presence of such a table and a specially designed insertion function which implements some insertion logic, it is possible to keep the data defined under different versions in the same table.

The function responsible for insertion in the API (see: INSERT_INTO) should manipulate the formatted data coming from the upper layer (since upper layer will want to add the tuple in the schema it is using, a naïve insertion operation will result in an error) the way it manipulates the row to be inserted is by injecting 'NULL' values for the missing attributes in the current schema after consulting the relation and attribute catalogues in order to identify which attributes are missing) Once, where the fields which 'NULL' value should be inserted is identified, rest is as straightforward as calling a SQL insert function. This way, the relation table is ready to answer queries that are intended to fetch data defined under different schema.

2.1.2 Assumptions

- Schema versioning keeps track of different schema versions defined over time. However, the way of reaching the historical data (older versions of the data) has more to do with how the most recent schema of the table is defined. If it has at least one temporal dimension, then our API provides ways to reach and process the historical data.

- Only Transaction Time schema versioning is allowed. In other words, our implementation does not support adding a schema proactively or retroactively. Instead application times of new schema is always defined between the end of the previous schema to the time when the schema is changed again.

- Some temporal conversions (generally, temporal downgrading operations) are not possible due to potentially previously defined primary keys of the otherwise affected table. More specifically, The reason for such a restriction is the possible illegal tuples with same keys these conversions might result in. These conversions are listed as follows: Conversion from BT to VT, BT to SN, VT to SN and TT to SN.

- Some temporal analysis operators such as PROACT, RETROACT, TJOIN, SEQDIFF and SEQDIV are not multi-versioning-compliant. This is because, the nature of these operators allow only some specific type(s) of temporality or only the temporality that only the kind of temporality which the current version of the table is. As an example for the former case, PROACT and RETROACT queries strictly requires bi-temporal time relations in order to detect proactive or retroactive changes that involves comparison of the validity time and transactional time of the tuples. For the latter case, SEQDIV is a good example. SEQDIV can be called with any kind of temporality unlike the PROACT or RETROACT. However, it only allows the operation if the current version of the relation is the same as the one specified in the parameter set of the procedure.

- Any table without a key cannot be defined. This is solely because, in the case of future temporal upgrade operation the relation will have only the key for temporal dimension(s), this gives space for illegal tuples with the same keys especially in the presence of multiple tuples are inserted before the conversion.

- Any attribute which is a key or a part of the composite key of a table cannot be removed.

- Added attributes are always non-key attributes.

2.1.3 Technical details of the project Implementation

The purpose of our project is to record changes that happen in our temporal data over the time-line. Because as we said the time-line is linear therefore we imply that whenever a change happens, we will have two separate states of our data, the old state of our data and the new state of our data. As someone might wonder, we need somehow to define which is the old data and which is the new data. For this purpose, each temporal data in our table will have a starting time attribute which keeps the datetime of when the data became to this state and an ending time attribute of datetime type which defines when the data stopped being in that state.

2.1.3.1 Temporal Types of Data

For the purpose of our real world, we will define two different types of temporality for the actual data. The one will be the Transaction Time Temporality and the other one will the Valid Time Temporality. We can also have a combination of those two, too. We have two different types because in practice an event often occurs in a different time than when an event is recorded in the database. Let's consider the example below:

Who?	What?	When?	What our DB knows?
John	Is born	In 20/08/1993	No John is born
John's Father	Registers John in the commune	In 26/08/1993	Now DB knows that John was born(in Brussels)
John	John continues living in Brussels	In 01/01/2013	John is in Brussels
John	Moves in Antwerp	In 05/11/2013	John is still in Brussels
John	Continues living in Antwerp	In 01/02/2014	John is still in Brussels
John	Register's in the commune	In 05/02/2014	John is in Antwerp

Taking into consideration the above table, we see that events happen independently from when is being recorded in the commune's database. For example John in 05/11/2013 moved in Antwerp but according to our database he was still living in Brussels until 5/02/2014, therefore we need to keep track of two different dates. For that, we have the Transaction Time which defines when an event is recorded in the database and the Valid Time which defines when the event actually happened and the combination of both of them as the above table. In occasions that we need to know only the Transaction Time, we create a Transaction Time table. In other occasions we want to know the Valid Time, we create a Valid Time table. In some occasions we need to keep track of both of them, we create a Bi-Temporal table. Also in order to be able to differentiate different tuples from each other, we should apply a primary key to each tuple. In the above example, a primary key could be the identification number of John or John by himself if we assume no other person with the name John exists.

2.1.3.1.1 Transaction Time Table

In this kind of table, every time we have a new tuple in our table, we assign to it the current time of the transaction and for its ending time, we define the keyword “U/C” which means Until Change. Practically this mean that the specific tuple, with this primary key, is the last modified tuple in our table. Whenever an update comes to change this tuple, the ending time of it, from “U/C” becomes the current transaction time and the new arrived updated tuple takes the current transaction time as its starting time and again “U/C” for its ending time. We also know that there cannot be any tuple with a future time because practically a database transaction can happen only in the present. Finally it is obvious that

the TT temporal columns (TST and TET) are being updated automatically from the system, without requiring the user to insert any datetime.

2.1.3.1.2 Valid Time Table

In a Valid Time table, we can have a time from the past, from the present or from the future, thus there is no restriction as in a TT dimension. In a VT dimension, only the user can insert the date interval of a tuple and additionally a tuple cannot be inserted in the VT table if there is not given at least the Starting Valid Time of the tuple. In a case that the Ending Valid Time of a tuple is missing then the system assigns to it the keyword “NOW” because theoretically the tuple's VET is infinite and every time the user access the particular tuple, he sees its current state in the present(“NOW”). Finally every time a new tuple with an existing primary key arrives and if the previous tuple, the existed tuple before the new arrival, has to its Valid Ending Time column the value “NOW” then this tuple(the old one) takes as its VET the current transaction time and the new arrived tuple takes the date interval that the user has assigned.

2.1.3.1.3 Bi-Temporal Table

In a Bi-Temporal database we do not have a new dimension but a co-existence of both previous dimensions, the TT dimension and the VT dimension. Every time a new tuple arrives, the temporality is handle with the respective rules of each dimension. Now as before the user has to assign dates only for the Valid Time dimension while the system will assign the dates for the Transation Time dimension automatically.

2.1.3.2 Metadata Tables

The Metadata tables serve for maintaining a historical archive of every table's schema change in the database. Those two tables are the core objects that allow us to maintain a Mutli-Version Database. Despite they are two different tables, they are cooperating closely together for keeping a table's metadata. The first one is the '*relation*' Metadata table which stores the Temporal information of all the tables in our system. Whenever a database table's Temporal dimension is changed, that fact is recorded into the '*relation*' table and the corresponding Application Times are being updated, too. There are also cases when we update the '*relation*' table whenever an update happens in the '*attribute*' table because a schema change has happened there and therefore we have to increase the Schema Version number by 1 (for both tables). From the point of view of the '*attribute*' table, the events that we record in this table have to do with schema changes of a table in our database. Every time a new column is added or an existing one is deleted, the '*attribute*' table is updated along with the Schema Version number. An additional case of updating the '*attribute*' table is when the Temporal dimension of a table has changed and therefore we have to record the 2 new Temporal attributes of the new schema. In general, we see that if one of the two tables is updated, then it is required to update the other one, too.

Below we have a more detailed explanation for each Metadata table:

RELATION

relation_name: The name of the relation we are making a reference to. This attribute is the Primary Key of this Metadata table.

schema_version_number: Each time we record a new table's Temporal state, we have to increase the schema version number by one. The last schema version number of a table indicates its current state.

schema_version_format: This attribute indicates the current Temporal dimension of a table. The Temporal dimension of a table can be either Snapshot(SN) or Transaction Time(TT) or Valid Time(VT).

app_start_Time: The Application Start Time keeps the initial time when a table obtained a specific Temporal dimension.

app_end_Time: The Application End Time keeps the last time of a Temporal dimension of a table. In case the Temporal dimension is still valid on the table, then it is being assigned a null value in this Metadata attribute. We should also note that the End Time of a Temporal dimension has to be equal to the Start Time of the new Temporal dimension of a particular table.

ATTRIBUTE

relation_name: The name of the relation we are making a reference to. This attribute is the Primary Key of this Metadata table.

schema_version_number: Whenever a schema change is happening, either because the user interaction or because of a Temporal dimension change, we have to update this attribute by 1.

att_name: Indicates the name of a particular attribute of a table in our database.

domain: Indicates the data type of an attribute. We should record a change on the domain of an attribute, as well.

is_key: Indicates if an attribute is a Primary Key of the table. The values(boolean) can either be 1 for an attribute being the PK of the table or 0 otherwise.

att_order_number: Shows us the ordinal number of a particular attribute in the table. This Metadata attribute proved to be helpful for dealing in various ways with table columns in some of our procedures.

2.1.3.3 A 'dump' table: SystemArrayList

We have created this particular table for storing temporal data which we handle during different processes of functions, mainly in the INSERT_INTO procedure. The need for the creation of this table is because the MS SQL Server does not provide any array variable or arraylist but only common variables. Thus in cases when we do not know the exact number of values we need to store in variables, we just insert the values in this 'dump' table for later use and we then immediately delete these values for not interfering with future similar values. The table has 3 attributes which helps a procedure to identify its own attributes(considering there are various procedures using this table simultaneously): the attribute '*Value*' which normally is used to store the actual value, the 'D*estination*' attribute which is normally used to defined the table destination of the value and the '*Other*' attribute for any other purpose. This table is a system table only(not handled as temporal).

2.2 TEMPORAL API

Functions of the Temporal API can be discussed under two different category namely, operational functions and maintenance functions. Additionally, some triggers are defined to enforce the rules established in the assumptions subtitle above.

2.2.1 Operational Functions

Users of the database management system that we have developed will be able to use some operational functions to query over the data reside in their database(s). Supported set of operational functions are mostly the ones that explicitly relate to temporal operations such as COALESCE, PROACT, RETROACT and TJOIN. Additionally, an insertion procedure called INSERT_INTO is also provided to replace standard SQL Server insert function. These functions differ from the maintenance functions in that these do not alter database structure in anyway but make simple transactions or visualize the data or metadata.

INSERT_INTO: A simple definition of this function is that allows the user to assign new data in a table. But besides that, the function takes care for all the Temporal aspects for inserting properly new data in the table. More specifically the function takes 3 parameters: 1st the name of the table in which the data will be assigned, 2nd all the values of the columns separated by the "|" character followed by a value for the particular column, and 3rd the schema version number of the table. For the 3rd paremeter, the logic is that the Multi-Version table has had multiple schemas over time and we have to specify in which schema we want to add the new values and accordingly to define the values of the columns in the 2nd parameter. This features allows user to see the relation as if it was defined under a past schema when inserting new tuple and it is essential to implement backward compatibility. In order for the user to be able to know the exact table schema of a specific version, there exist the GET_TABLE_SCHEMA procedure which is going to be explained in a separated section. Explaining further the 2nd parameter, if for example we have a table schema as below:

RandomTable (A int primary key, B int, C varchar(10), D datetime)

The parameter's values should be: '4|5|randval|2013-12-22 07:40:54.151'

In this point we have to say that we always define the value of the first column without a “|” character and after the first column we always do use the “|” character and the value of the next column. We also should have in mind that we do not have and shouldn't add the “|” character at the end of the query. The only case we can type the “|” character at the end of the query is when we want to leave the last column empty and therefore we do not type anything else afterwarts, as in the example below:

'4|5|randval|' <- we have left the datetime empty(or NULL)

Below is following a full example of this procedure considering again the above table in its 4rth version schema:

INSERT_INTO 'RandomTable' , '10|30|MATHH415|2013-12-22 07:40:54.151' , '4'

We also should mention that there is not any update function and whenever the user wants to update an existing column, he has to use the INSERT_INTO logic and to re-enter the new tuple's values. The function by itself will identify that the user has inserted an already existed tuple(by looking the primary key(s)) and will insert the new tuple following the Temporal rules of the table, if and only if the table has a Temporal dimension otherwise an update in a SN table is not possible because we also assume the SN table being Temporal but without time-crossover 'abilities'.

GET_TABLE_SCHEMA: This procedure is usually called before using the INSERT_INTO procedure. The reason is that the GET_TABLE_SCHEMA proc. receives the table's name and a version number and presents in an ordinal order all the columns of that table's schema in that specific version. Considering this column ordering, the user has to assign the values in the 2nd paremeter of the INSERT_INTO procedure.

Example:

GET_TABLE_SCHEMA 'RandomTable', 2

ADD_COLUMN_TO_TABLE: This procedure adds a new column to the specified table and record the event in the Metadata tables. The most important update happens in the '*attribute*' table where we re-import all the table's columns plus the new one and increase the version number by one. A similar action is performed in the '*relation*' table where we also increase the version number by one.

Example:

ADD_COLUMN_TO_TABLE 'RandomTable', 'NewRandomCol', 'INT'

DROP_COLUMN_TO_TABLE: This procedure makes the particular column unvailable for future use from the user by removing its existence in the new schema version in the Metadata tables but does not remove it from the actual table because in this way we can keep all the values of this column up to its deletion and therefore having a Temporal history of the data, no matter if they have been deleted.

Example:

DROP_COLUMN_TO_TABLE 'RandomTable', 'ExistingRandomCol'

GET_TABLE_CURRENT_SCHEMA: We usually use this procedure in order to retrieve a table's schema attributes in its most new schema version. The only paremeter we have to provide is the table name. The columns shorting is made by the ordinal position of the them.

Example:

GET_TABLE_CURRENT_SCHEMA 'RandomTable'

GET_TABLE_ALL_SCHEMAS : By using this procedure we take all the table schema attributes from all the existing schema version, thus we will get two columns in our results, the first one which states the schema attributes and the second which states the schema version number. The shorting is made by the schema version number and the ordinal position of the columns.

Example:

GET_TABLE_ALL_SCHEMAS 'RandomTable'

PROACT: When a bi-temporal time relation passed to it, this procedure displays the tuples that are inserted proactively (validity starts later than the transaction time) to the database.

RETROACT: Like PROACT, when a bi-temporal time relation passed to it, this procedure displays the tuples that are inserted retroactively (validity starts earlier than the transaction time) to the database.

TJOIN: This procedure basically computes the sequenced (temporal) join of two tables according to the temporality type specified for each which is taken into account when finding the overlapping tuples. Temporality type can be either ‘VT’ or ‘TT’ implying that relation to be joined cannot be snapshot. User can join two tables with different temporality types (e.g one is transactional time whereas the other is valid time) as long as the relation supports the specified temporality. For example, while temporal join of a bi-temporal time relation over validity time and transactional time relation over transactional time is possible, join of these two is not allowed when temporality specified for the latter is ‘VT’

Example:

TJOIN ‘rel1’,’TT’,’rel2’,'VT’

SEQDIFF: This procedure computes the sequenced difference of two relations. The interface of the procedure is pretty much the same as that of TJOIN. However, It takes only one relation name defined in the database and one temporality types associated with it. Then, it finds the sequenced difference over the temporal dimension specified (VT or TT)

SEQDIFF ‘rel1’,’VT’

SEQDIV: This procedure computes the sequenced division of three relations. The procedure takes three relation names and three temporality types which will be considered as the temporal dimension of each of them respectively when computing the division. The first relation should always be the ‘dividend’ In other words, it should be the relation in which the user wants to finds the tuples whose existence over the temporal domain is universally spans over the existence period of the correlated tuples from the second and third relations. Additionally, this correlation of the relations should be defined by some common attributes (foreign keys) More concretely, all these three relations should have mapping that

correlates the first relation to the second, the second to the third, and the third to the first one. A good illustrative procedure call for this function is as follows:

create table aff(SSN int, dnumber int, primary key(SSN,dnumber))
create table controls(dnumber int, pnumber int primary key(dnumber, pnumber))
create table workson(SSN int, pnumber int primary key(SSN, pnumber))

SEQDIV att,'TT',controls,'TT',workson,'TT

After creating the relations, Following SEQDIV call is valid and produces the sequenced division of the tuples in the first relation over the tuples in second and third relations when they all have a valid 'TT' dimension (Note that they can be bi-temporal relations as well as transactional time)

COALESCE_TABLE: In the Coalesce method, we put all the existing dates in an 'linear' order such as they again form a bigger time interval constructed by all the other smaller and spread dates. The bigger interval starts from a random date until the sequence of dates is not interrupted. Below we explain the procedure's parameter:

TABLE_NAME: The name of the table.

COLUMN_NAME: The name of the column which we are going to project over the time-line that is going to be formed.

TemporalDimension: The user should provide the Temporal dimension, he wants to coalesce the table. This Implies that the user must know what is the current Temporal dimension of the table and pass the Temporal dimension. The idea is that in a Bi-Temporal table, there isn't any factor about on which dimension to Coalesce, therefore the user must take this responsibility. There are two options: the 'TT' value for coalescing on the Transaction Time Columns(TST & TET) or the 'VT' value for the Valid Time Columns(VST & VET)

Example:

COALESCE_TABLE 'RandomTable', 'ExistingRandomCol', 'VT'

SEQ_AGGREGATION: The functionality of this procedure is to find all the time intervals that exist in our Temporal table and then compute a specific aggregation value for each one of those intervals. At this point we should have in mind that no coalesce method happens yet in this function, therefore the dates do not form a continues time period. Below we are explaining in details the parameters of this procedure:

TABLE_NAME: The name of the table.

COLUMN_NAME: The name of the column in which we are going to aggregate the value.

AGGREGATION_FUNCTION: Here we have to specify which aggregation method we want to use. There are 5 possible options we can give: 'MAX', 'MIN', 'AVG', 'COUNT' and 'SUM'.

InsertResultsIntoTable: The Boolean value that this parameter accepts indicates whether the results should be saved into a new table or should be outputted to the user's screen. For saving the results into a table, we should pass the value 1 otherwise for just showing the results on the screen, we pass the value 0. In case of saving the results into a table, the name of that table is: the original name of the table we are making the aggregation sequence followed by the aggregation function name: e.g.

Randomtable+MAX= RandomtableMAX . Let us also mention that the purpose of saving the results into a table is made mostly because we it is needed into the 'COALESCE_TABLE_WITH_AGGREGATION' procedure.

Example:
SEQ_AGGREGATION 'RandomTable', 'ExistingRandomCol', 'MAX', 'VT', 0

COALESCE_TABLE_WITH_AGGREGATION: Here we are implementing the Coalesce method and the sequenced aggregation method. More specifically, we are aggregating the values of each date interval that we form and then ordering those intervals in a continues time-interval. Let us explain in details the parameters of this procedure:

Example:
COALESCE_TABLE_WITH_AGGREGATION 'RandomTable', 'ExistingRandomCol', 'MAX', 'VT'

2.2.2 Maintenance Functions

Maintenance functions are some procedures provided in Temporal API for the use of database administrator to manage the database. In other words, they provide an interaction layer with a number of interfaces for applications with elevated permissions to manipulate the temporality and versioning of the database. In particular, these operations include upgrading of temporality, adding and removing non-key attributes. Since, these procedures change the state of relations –changing the current schema- and might add or remove some significant data, they should be used with caution although in principle they can't cause any loss of data due to the schema versioning (except for the temporal dimension, because there are some downgrading operations available which simply chops off some one temporal dimension altogether, see CONVERT_BT_TO_TT, and hence might lead temporal data loss)

CONVERT_RELATION_SN_TO_TT: What this function essentially does is increasing temporal dimension by introducing some temporal columns to the relation it is applied to. Added columns are Transaction Start Time (TST) and Transaction End Time (TET) Furthermore, as soon as this function is executed TST becomes a part of the composite key of the relation to maintain temporal integrity. All the past tuples inserted prior to the conversion will yield the time of the application of the procedure (application time) for the field TST and Until Change (UC*) for the field TET.

Example:
CONVERT_RELATION_SN_TO_TT 'RandomTable'

CONVERT_RELATION_SN_TO_VT: What this function essentially does is increasing temporal dimension by introducing some temporal columns to the relation it is applied to. Added columns are Validity Start Time (VST) and Validity End Time (VET) Furthermore, as soon as this function is executed VST becomes a part of the composite key of the relation to maintain temporal integrity. All the past tuples inserted prior to the conversion will yield the time of the application of the procedure (application time) for VST and 'NOW' for the field VET.

CONVERT_RELATION_SN_TO_BT: What this function essentially does is increasing temporal dimension by introducing some temporal columns to the relation it is applied to. Added columns are Transaction Start Time (TST), Transaction End Time (TET), Validity Start Time (VST) and Validity End Time (VET)

Furthermore, as soon as this function is executed VST and TST become a part of the composite key of the relation to maintain temporal integrity. All the past tuples inserted prior to the conversion will yield the time of the application of the procedure (application time) for the fields of TST and VST, 'NOW' for VET and 'Until Change' for the field TET.

CONVERT_RELATION_TT_TO_VT: This procedure changes temporal dimension by translating tuples defined under Transaction Time to Validity Time. Translation logic replaces columns TST-TET with VST-VET. Primary key constraint is updated so that VST becomes the key or part of it while TST is dropped totally. For the past tuples, VST takes the value of TST and VET takes 'NOW'

CONVERT_RELATION_VT_TO_TT: This procedure changes temporal dimension by translating tuples defined under Validity Time to Transaction Time. Translation logic replaces columns VST-VET with TST-TET. Primary key constraint is updated so that VST becomes the key or part of it while TST is dropped totally. For the past tuples, TST takes the value of VST and VET takes 'UC'

CONVERT_RELATION_TT_TO_BT: What this function essentially does is increasing temporal dimension by introducing some temporal columns to the relation it is applied to. Added columns are Transaction Start Time (VST) and Transaction End Time (VET) Furthermore, as soon as this function is executed VST becomes a part of the composite key. Note that VST was already (part of the) primary key since prior to the conversion relation was Transaction Time. VST of the all the past tuples take the value of TST and VET yields the value 'NOW'

CONVERT_RELATION_VT_TO_BT: What this function essentially does is increasing temporal dimension by introducing some temporal columns to the relation it is applied to. Added columns are Validity Start Time (TST) and Transaction End Time (TET) Furthermore, as soon as this function is executed TST becomes a part of the composite key. Note that TST was already (part of the) primary key since prior to the conversion relation was Transaction Time. VST of the all the past tuples take the value of TST and VET yields the value 'NOW'

CONVERT_BT_TO_TT: This function downgrades the relation passed to it by eliminating the temporal dimension of 'Valid Time' from the relation. Even though one of two eliminated columns namely the 'VST' is part of the composite key, removing this column does not result in tuples with the same values in their corresponding composite key fields since 'TST' which no two of any tuple can necessarily have the same value of (note that, transaction start time is recorded by the insert_into procedure and always equals to the exact system time which the insertion of the tuple is realized) is guaranteed to still be the part of the composite key in this conversion.

2.2.3 Triggers

MetadataEvent_NewTableCreation: This trigger fires whenever a new table is created in the whole database. After the new table is created, the trigger records this event in the Metadata tables: 'relation' and 'attribute'. Every table is created initially with a SN dimension and its initial version is number 1.

DataIntegrityTrigger_TABLERenamingRestriction: This trigger simply forbids the user to rename a table because in our Metadata tables, the name of a table is part of the primary key. Therefore if we would allow a table renaming we would not be able to track tables in the Metadata tables properly.

DataIntegrityTrigger_OR_MetadataEvent_ColumnRenamingRestriction_OR_ColumnRenaimingEvent: This trigger focus on the column renaming of any table. If the user tries to rename a column which is a Temporal column (such as TST or VET) then the trigger blocks the renaming and throws an error message to the user. On the other hand, if the user wants to rename a column other than the Temporal ones then the action is performed normally and is being recorded in our Metadata tables. A column renaming in our Metadata tables is considered as we create a new column, thus the old column cannot be used anymore by the user(despite it still exists) and the new renamed column is used instead. This action, following the Temporal rules, is equal to dropping one column and adding a new column to the table.

3 CONCLUSION

3.1 OBSERVATIONS

During the development of our project, the idea of putting temporality and versioning together was more challenging than we had been thinking before commencing on the implementation of this idea. Eventually, we have faced many problems that we had to come up with some real solutions or workarounds for them.

For example, ideally, a database management system with a native versioning support should always be able to revert the database back to the any state that existed in past, however, in our design and implementation when user wants to downgrade the schema of a relation (note that, there is only one such temporal conversion procedure is provided: conversion from bitemporal time to temporal time) some irreversible information loss occurs. We were tempted to exclude out this procedure to avoid this possible permanent loss in data, but then we thought there can also be some situations where user deliberately wants to lose the data. The consensus we reached at this point was to keep the procedure yet not let everyone use it but some user with elevated permissions. However, since this kind of authentication requirements should be handled on the upper layer (application that drives the programming interface we developed), implementation of such a permission checking mechanism was beyond the scope of the project.

Another challenge was to implement every single temporal procedure in such a way that it can be applied to a past schema of a relation and produce results as if the past schema was the current one. This was possible for at least two main operational functions to insert data and fetch data from database. For insertion, a special procedure named INSERT_INTO is provided. For, displaying data resides in the table, user can first learn the schema of a version he wants to display the data in the form of. Then he can select tuples with the set of attributes from the schema version he wants to work with. However, for many maintenance procedure or complex operational queries such as SEQDIV, SEQDIF or TJOIN,

schema specified should be the current schema. For the former case (maintanence procedure) this restriction was because of the very nature of the procedure to be applied to the relation. For example, nobody can expect temporal conversion function that converts the table from snapshot temporality to bi-temporal temporality to be applied to a transaction time relation while there are temporal conversion procedures available for transaction time relations. For the latter case, this was due to our design choice. Since the data is always kept under the temporal column types that is defined under the 'current' schema, it was not possible to apply some of the operational temporal queries for a past version with less temporal division than the current (this is well possible in the case of temporal downgrading conversion is applied to the relation in question)

There were also some technical challenges that we were able to come up with complete solutions. For instance, sometimes in our procedures we have to disable specific triggers which interfere with processes of the procedure. An example is in the SEQ_AGGREGATION where we may need to create a new table(to save the results into) but this action will fire the 'MetadataEvent_NewTableCreation' and record the event into the Metadata tables. Such as we fist disable this trigger before the interfered processes and re-enable it right after those processes.

In our database when a table gets a Temporal Dimension(TT, VT) then we assign in the existing Primary Key(s), that the user defined, the Start Time of the Temporal dimension(TST or VST), too. The reason is because whenever we need to re-insert an already existing tuple in the table(because of Temporality), keeping only the user defined PK(s), will throw us an error of duplicate Primary Key. For this reason we also assign as part of the PK, the Start Time, in order to avoid duplicate PK error from the system.

On the other hand, we also need somehow to be able to detect the transaction when the user inserts a tuple which conflicts with the initial Primary Key(s) but in this case we don't get any error from the system. For this, in order for us to be able to identify an initial PK conflict without involving the DBMS, we just do not record the Temporal Columns' Start Time into our Metadata table '*attribute*' as part of the table's Primary Key. So now we can identify a PK conflict by the Metadata table and in parallel not having a PK conflict error from the system. This particular issue concerns mostly the INSERT_INTO procedure.

On another observation, we do not provide any procedure for converting a TT table or a VT table to a SN table. The reason is because we do not want this event to happen because in a Multi-Version table there will be tuples with duplicate initial Primary Key which currently do not conflict because of the Temporal Start Time column being part of that PK. But in a case where we want to make the table SN and therefore delete the Temporal Start Time columns, we will have duplicate Primary Key errors from the system which cannot be handled unless we delete them(but we cannot even think about such thing). Therefore such a conversion is not feasible to be implemented.

Another observation is that whenever the user wants to rename a column, he has to rename it by using the MS SQL server statement '*sp_rename'* for the trigger being able to run properly. Otherwise if we use the DBMS wizard, for some reason the trigger will fire twice because in this case the DBMS seems to complete the task into two steps and therefore firing the trigger 2 times.

4 REFERENCES

- Brahamia Z., Mkaouar M., Chakhar S., Bouaziz R., "Efficient Management of Schema Versioning in Multi-Temporal Databases The International Arab Journal of Information Technology, vol. 9, no. 6, pp. 544-552, 2012
- Zimanyi, Esteban. "Temporal Aggregates and Temporal Universal Quantification in Standard SQL."

- Torp K., Jensen C., and Snodgrass R., “Effective Timestamping in Databases,” The VLDB Journal, vol. 8, no. 3-4, pp. 267-288, 2000