Database Design That Can Work By Separating Different Branches

İSMAİL KILIÇ, FIRAT UNIVERSTY

Most database designs are not temporal. If we want old changed data, It will not be possble. Because If we did change data with update operation on non-temporal table desing, we will lose changed data. Temporal designs keep valid time and transaction time data, so even the old data can be accessed after updating. Much of the research on bitemporal databases has focused on the modeling of time-related data with either attribute or tuple timestamping. While the attribute-timestamping approach attaches bitemporal data to attributes, the tuple-timestamping approach splits the object's history into several tuples. Although there have been numerous studies on bitemporal data models, there is no work database design that works on branches and version control system like git system. In this paper, we will talk about codes, interfaces and database desing working like git system on branches and commit. We will analyze a system and this this system has a database. All data in this database on tables. It can revert to any commit on any branch, when we want revert to setected commit.

Categories and Subject Descriptors: I.5.1 [**Pattern bitemporal**]: Desings—*ersion control system on database*

Additional Key Words and Phrases: Bitemporal table desing, bitemporal query, version control system on database,

1. INTRODUCTION

Many techniques for Database desing have been developed and applied. These are usually developed for use in e-commerce systems. Old data protection not as important as for e-commerce applications or websites but It is important for Enterprise resource planning (ERP) applications or website. At this point the applications are using the validity time to access the old data. This design is called the temporal design. In this article we are talking about that revert to old data in selected time and data on branches in a database.The data on this database may have different values in different branches. When requested, it can be revert to desired time and desired branch. Data will be ensure that change on branches, not like a singel line. If I will show an example for this solution, It will be git repositories. Bitemporal design has been accepted for this solution.

1.1 Non-Temporal Databases

Commercial database management systems (DBMS) such as Oracle, Sybase, Informix and O2 allow the storage of huge amounts of data. This data is usually considered to be valid now. Past or future data is not stored. Past data refers to data which was stored in the database at an earlier time instant and which might has been modified or deleted in the meantime. Past data usually is overwritten with new (updated) data. Future data refers to data which is considered to be valid at a future time instant (but not now).

Table I. Non-Temporal Databases

EmpID	Name	Department	Salary
10	John	Sales	12000
12	George	Research	10500
13	Ringo	Sales	15500

A DBMS stores the data in a well-defined format. A relational DBMS, for example, stores data in tables (also called relations). Thus, a relational database actually contains a set of tables. Each table contains rows (tuples) and columns (attributes). A row contains data about a specific entity, for example, an employee. Each column specifies a certain property of these entities, for example, the employee's name, salary etc. The following table stores data about employees:

http://www.timeconsult.com/TemporalData/TemporalDB.htmlTemporal

1.2 Temporal Databases

Temporal data strored in a temporal database is different from the data stored in non-temporal database in that a time period attached to the data expresses when it was valid or stored in the database. As mentioned above, conventional databases consider the data stored in it to be valid at time instant now, they do not keep track of past or future database states. By attaching a time period to the data, it becomes possible to store different database states.

A first step towards a temporal database thus is to timestamp the data. This allows the distinction of different database states. One approach is that a temporal database may timestamp entities with time periods. Another approach is the timestamping of the property values of the entities. In the relational data model, tuples are timestamped, where as in object-oriented data models, objects and/or attribute values may be timestamped.

What time period do we store in these timestamps? As we mentioned already, there are mainly two different notions of time which are relevant for temporal databases. One is called the valid time, the other one is the transaction time. Valid time denotes the time period during which a fact is true with respect to the real world. Transaction time is the time period during which a fact is stored in the database. Note that these two time periods do not have to be the same for a single fact. Imagine that we come up with a temporal database storing data about the 18th century. The valid time of these facts is somewhere between 1700 and 1799, where as the transaction time starts when we insert the facts into the database, for example, January 21, 1998.

Assume we would like to store data about our employees with respect to the real world. Then, the following table could result:

Table II. Temporal Databases

EmpID	Name	Department	Salary	ValidTimeStart	ValidTimeEnd
10	John	Research	11000	1985	1990
10	John	Sales	11000	1990	1993
10	John	Sales	12000	1993	INF
11	Paul	Research	10000	1988	1995
12	George	Research	10500	1991	INF
13	Ringo	Sales	15500	1988	INF

The above valid-time table stores the history of the employees with respect to the real world. The attributes ValidTimeStart and ValidTimeEnd actually represent a time interval which is closed at its lower and open at its upper bound. Thus, we see that during the time period [1985 - 1990), employee John was working in the research department, having a salary of 11000. Then he changed to the sales department, still earning 11000. In 1993, he got a salary raise to 12000. The upper bound INF denotes that the tuple is valid until further notice. Note that it is now possible to store information about past states. We see that Paul was employed from 1988 until 1995. In the corresponding *non-temporal table*, this information was (physically) deleted when Paul left the company.

1.3 BiTemporal Databases

Transaction time and valid time are independent time dimensions that are used for different purposes. Bitemporal tables have both a transaction-time column and a valid-time column. Changes to bitemporal tables that happen automatically as a result of row modifications are independent for the transaction-time and valid-time dimensions. These dimensions must be considered separately when determining what will happen to a row as a result of a modification.

Valid time:. Valid time is the time for which a fact is true in the real world. A valid time period may be in the past, span the current time, or occur in the future.

Transaction time:. Transaction time records the time period during which a database entry is accepted as correct. This enables queries that show the state of the database at a given time. Transaction time periods can only occur in the past or up to the current time. In a transaction time table, records are never deleted. Only new records can be inserted, and existing ones updated by setting their transaction end time to show that they are no longer current.

A bitemporal DBMS stores the history of data with respect to both valid time and transaction time. Note that the history of when data was stored in the database (transaction time) is limited to past and present database states, since it is managed by the system directly which does not know anything about future states.

For example, if a row in a bitemporal table is deleted, the ending bound of the transaction-time period is automatically changed to reflect the time of the deletion, and the row is closed to further modifications. The database reality, reflected by the modified ending bound of the transaction-time period, is that the row has been deleted.

Table III. BiTemporal Databases

EmpID	Name	Department	Salary	Valid Time Start	Valid Time End	Transaction Time End	Transaction Time End
10	John	Research	11000	1985	INF	1985	1990
10	John	Research	11000	1985	1990	1985	INF
10	John	Sales	12000	1990	INF	1985	INF
12	George	Research	10500	1991	INF	1992	1999
12	George	Research	10500	1991	1999	1992	1999
13	Ringo	Sales	15500	1988	INF	1988	INF

the table also includes a transaction-time dimension, another copy is made of the original row, reflecting the original period of validity, but the row is closed in transaction time at the time the terms changed. No further changes can be made to this row, because it is closed in transaction time. It provides a permanent "before" snapshot of the row as it existed in the database before it was changed.

2. ANALYSIS

We will talk about our solition but there are other candidates that we will mention shortly. If a VCS(Version Control Systems) saves database scripts at the 'object' level then each file corresponds to a table or routine. In this case, the task of creating a new database or upgrading an existing one, from what's in source control, is primarily an exercise in creating an effective database script from the component 'object' scripts. These will have to be executed in the correct dependency order. Subsequently, any static data must be loaded in such a way as to avoid referential constraints being triggered. https://www.red-gate.com/simple-talk/sql/database-delivery/database-version-control/

2.1 Analysis of existing database version control systems

We can divide these VCS into 2 categories. These ones;

State-based tools. Generate the scripts for database upgrade by comparing database structure to the model (etalon).

Migration-based tools. Help/assist creation of migration scripts for moving database from one version to next.

These applications is important for backup but these VS system is not keeping table data. You can see what is VS system safekeeping;

Database configuration properties.

DDL scripts to define database users and roles, and their permissions.

Database creation script.

Database interface definition (stored with the application it serves).

Requirements document.

ETL scripts, SSIS packages, batch files and so on.

SQL agent jobs..

Table IV. This is the list of source version control tools for databases.

Redgate SQL Source Control
Plugs into SQL Server Management Studio: SSMS 2017, SSMS 2016, SSMS 2014, SSMS 2012 Type: State-based Supports databases: SQL Server Supports repositories: Team Foundation Server, Subversion, git, Mercurial, Perforce, SourceGear Vault, Working Folder Notable features: Locking objects in Shared model Commercial/Free: Commercial Free edition: no, Pricing from: 495
dbForge Source Control
Plugs into Microsoft SQL Server Management Studio: SSMS 2016, SSMS 2014, SSMS 2012 Type: State-based Supports databases: SQL Server Supports repositories: Team Foundation Server, Subversion, git, Mercurial, Perforce, SourceGear Vault, Working Folder Commercial/Free: Commercial Free edition: no, Pricing from: 250
Source Control for Oracle
Type: State-based Supports databases: Oracle Supports repositories: Subversion, Team Foundation Server Commercial/Free: Commercial Free edition: no, Pricing from: 369
Version SQL
Plugs into Microsoft SQL Server Management Studio: SSMS 2017, SSMS 2016, SSMS 2014, SSMS 2012 Type: State-based Supports databases: SQL Server Supports repositories: Git, Subversion Commercial/Free: Commercial Free edition: yes, Pricing from: 149

2.2 Our Solution

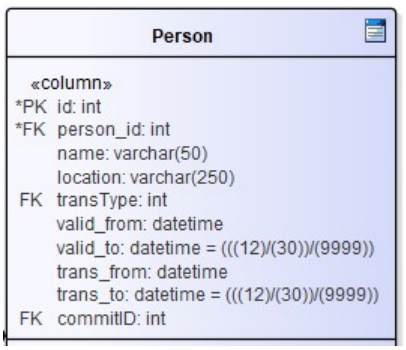
Other database version control systems were examined in the final report. As a result of this review, I analysed whether we arrive our wish. Existing Version Control Systems back up the database structure or commit with do store a specified format it on the Microsoft Foundation Server or Git. In addition, these systems are expensive and paid applications. This system is available for developing a application with team. But it does not has the standards we want.

We developed our desing and our application. This design is working logically similar to GIT.Each new data is added, edited or deleted with a commit number. Each commit is associated with a brach and Each brach is associated with a brach and commit. Thus, we have reached the circular design that we want.

3. DESIGN AND IMPLEMENTATION

3.1 Bitemporal Table Design

Bitemporal tables include valid and transaction time. Once if a data updated , 2 row will be added to table and 1 row will be edited.So It will be duplicate personID. We catch current row with ID column. ID column will change on every commit for current row. We are using transType column because we need facilitate access to these newly added rows. transType is include Insert(indexID: 1), Update(indexID: 2), Delete(indexID: 3).
We use the commitID column for find which data changed on commit. The commitID column associated with the id column in the Commit table. We need add bitemporal desing for every we want use table with this git system.



Id: Primary Key, we need it for return a current time
Person Id: logical ID for determine the person
Name: Person Name
Location: Place of residence
transType: Insert, Update or Delete
valid from: valid start datetime for temporal desing
valid end: valid end datetime for temporal desing
trans from: transaction start of datetime for Bi-temporal desing
trans to: transaction end of datetime for Bi-temporal desing
Commit ID: This data belongs to which commit

Fig. 1. Person Table and Information about table

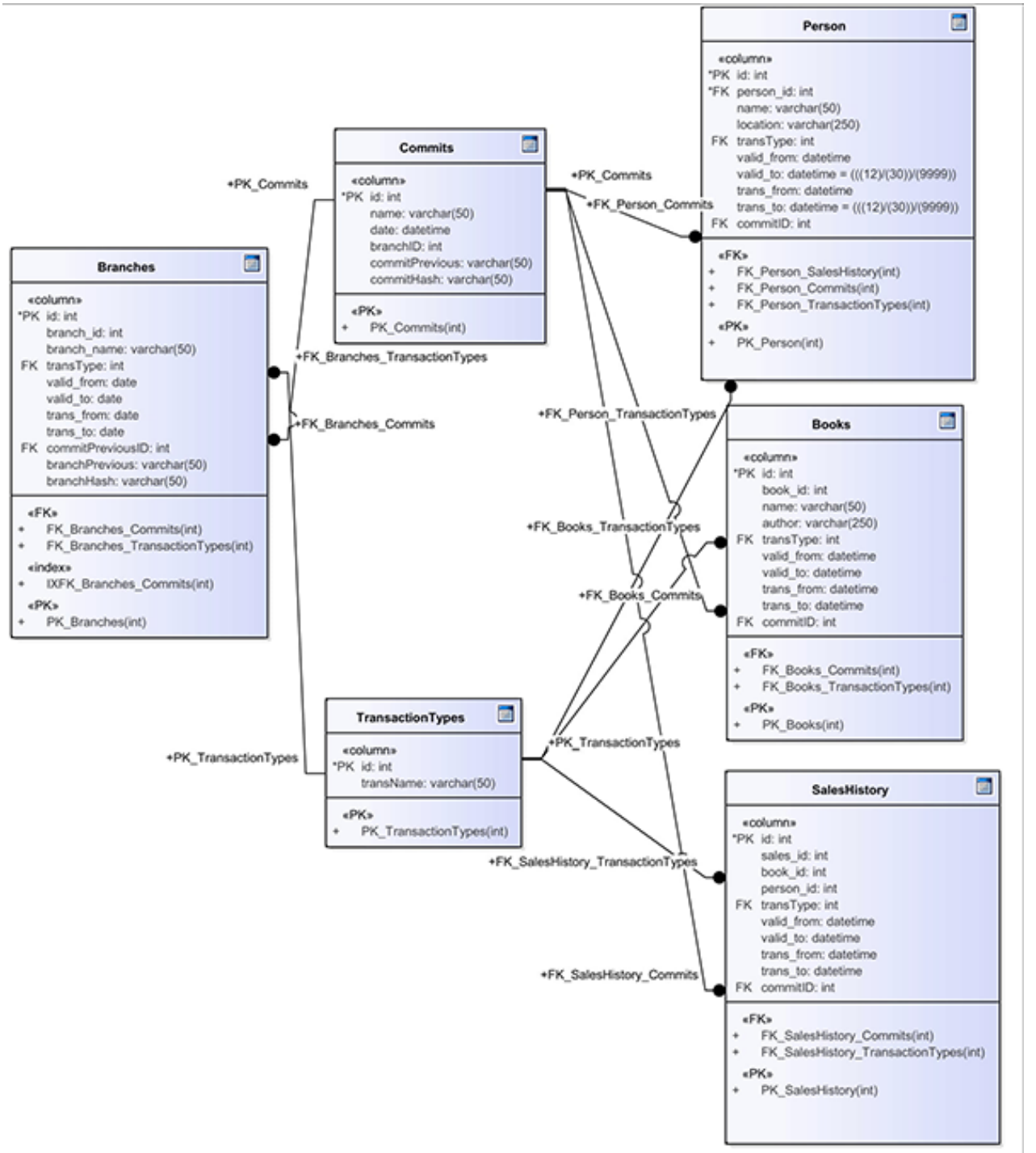


Fig. 2. Database diagram

3.2 Temp Tables for valid data on selected commit

If we changed our selected commit, we are filling current data to temp tables on selected commit.
Our software is working on these Temp tables. Because If we will select all current data every time on selected commit, It will be problem for our DB performance.
We are execute just select operations with these tables. We execute other operations to other tables.

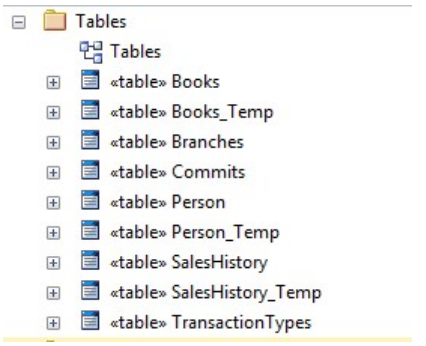


Fig. 3. All tables

Fig. 4. Information about Temp table

3.3 StoredProcedures for fill temp tables when changed selected commit

A **stored procedure** (also termed proc, storp, sproc, StoPro, StoredProc, StoreProc, sp, or SP) is a subroutine available to applications that access a relational database management system (RDBMS). Such procedures are stored in the database data dictionary.
Uses for stored procedures include data-validation (integrated into the database) or access-control mechanisms. Furthermore, stored procedures can consolidate and centralize logic that was originally implemented in applications. To save time and memory, extensive or complex processing that requires execution of several SQL statements can be saved into stored procedures, and all applications call the procedures. One can use nested stored procedures by executing one stored procedure from within another. Stored procedures may return result sets, i.e., the results of a SELECT statement. Such result sets can be processed using cursors, by other stored procedures, by associating a result-set locator, or by applications. Stored procedures may also contain declared variables for processing data and cursors that allow it to loop through multiple rows in a table. Stored-procedure flow-control statements typically include IF, WHILE, LOOP, REPEAT, and CASE statements, and more. Stored procedures can receive variables, return results or modify variables and return them, depending on how and where the variable is declared.
https://en.wikipedia.org/wiki/Stored$_{p}rocedure$

We created stored procedures for sequential operations;

sp getPersonOnMaster New2: This sproc taking 1 paremeter from outsite. It is just commitID.If we will sent commitID to this commit, It will return valid data for selected commit's branch

sp getActualPersonOnSelectedCommit: This sporc calling "sp getPersonOnMasterNew2" If we will sent commitID to this commit, It will fill temp table with valid data for selected commit. This sporc clear deleted data on returned rows and It will add valid data to temp tables.

sp changeVersion: this sproc execute "sp getActualPersonOnSelectedCommit". Because we need execute some sproc and fill all bitemporal temp tables

4. VERIFICATION AND CONCLUSION

CommitHistory page was added to check accuracy of the data and the correctness of the project in the developed application. Changes made to commits can be monitored via commitTree in this page. On this page we can see which commit is associated with which commit. If desired, we can follow step-by-step changes of the data on commits.
3 branches were created on the designed database. One of them 2 level. Test data were entered, including in relational tables. This input was committed on separate times. This inputs was checked step by step. I saw some inconsistency on this test. If we want first commit to any other branch, this data's transTo date is changing, so selected data has different date from the previous branch. If we will need transTo column in the future, it will be problem. We are using transTo column for version control system, so this problem isn't a problem for version control system on database. Because this construction is our main key for our solution.