Database Versioning, a Transparent SQL Approach

Cosmin CIORANU¹, Marius CIOCA²

¹Executive Unit for Higher Education, Research, Development and Innovation Funding
²“Lucian Blaga” University of Sibiu
ROMANIA

cosmin.cioranu@gmail.com, marius.cioca@ulbsibiu.ro

Abstract. In today’s times every piece of software no matter the complexity, produces data, redundant or otherwise, that needs or should be stored in a permanent storage. The simplest way is a file system, but where the need of complexity and flexibility is required, most system designers prefer to use some kind of Database Management System (DBMS). Even with such a management system, on large amounts of data, combined with information updates, it raises the need to retrieve and track the history of modifications of a specific information alongside the dependencies that makes it coherent. Considering the above the natural solution is to have a database revision or versioning system in which the existing algorithms will still perform without or little modifications and the cost of implementation of such a solution is minimal.

Key-Words: DBMS, SQL, Versioning, Data Versioning.

1. Introduction

In today’s software, no matter the complexity, the need of storing the produced data is paramount in most cases, therefore the management system holding it makes the difference between good software versus a bad one. In most cases a simple software architecture combined with a database design where the requirements does not ask explicitly for the tracking modifications (versification of data) of the information in long lived databases [2], produced by certain algorithm or advanced applications (CAD/CAM, GIS) [2][5][6], is enough, but there are cases when such a solution has to be taken it into consideration. Data versioning in general can be use as basis for security purposes, or tracking information movements, but we can see other uses like maintaining an incremental backup, or data redundancy [3][4]. Most DBMS used today, have a built in system of versioning with or without redundancy, automatic backup [3] or manual backup controlled by administrator and even advanced support as a combination between all of them (Oceanstore, PASIS) [3], but in most cases is not enough, in terms of data history and data dependency retrieval. Usually a backup maintained by a user or administrator in order to be effective can act as versified data management but it has to be accessed and installed in some query able way, which implies time and costs, and for some applications (e.g. Bank systems) where the data has to be rapidly accessed is not acceptable [5][6].

2. General overview over the solution

The paper presents an approach designed especially for this purpose, in which the access to the information history is transparent to the end user (which can be an algorithm) or other kind of process. The presented concept of database versioning is aimed at long term databases [6] and at the data level. The schemata level versioning is supported, but not as an explicit feature, changes reflected in data can be accommodated but only by addition and not by deletion or alteration of elements of current elements. The proposed approach of versioning can be seen in many flavors, but all have in common the fact that from an architectural point of view, the information is stored in layers of data identified by am unique identifier [Figure 1].

Any versioning system needs to take into consideration the following issues: performance, hardware resources consumption, maintenance effort [3].
In terms of performance, the solution is mainly a rewriting of a certain request sent to the DBMS server. So every issue regarding performance will be related mostly with the cost of the server to accommodate this new request. Of course certain caches can be built, at the software level [8], or even at the storage server [9] but we will not approach the optimization methods now. Moreover, at the performance level we have to take into consideration the following: the performance costs of accessing old version of data, how many versions are available and the completeness of data versioned [3]. Again the performance costs are related to how many versions are available and in terms of completeness the data is fully versioned, no data is lost, which implies increase of storage capacity.

In terms of hardware resource, the overhead is mainly given by the increase number of version available kept in alive state (keeping them accessible) [4] and on the number of concurrent accesses to those data. Keeping in mind that the cost per-byte and processing power, in later years, is becoming lower, for small to middle systems (like middle online shop or a small to middle organization, or even financing agency) will not be an issue. Maintenance effort, is minimized, due to the fact that that the proposed schema is built upon an existing architecture, in our case a DBMS, which limits the problems related to the storage of data.

3. Approach

The general approach in building such a solution, in terms of architecture, was the man-in-the-middle, or proxy, as shown in [Figure 2], giving up the flexibility of an implementation starting from a simple ODBC (Open Database Connectivity) or one similar. In the left we see the generic, or usual approach, every query, or request sent to the server as it was written, in the right, our solution, does transforms, by adding certain parameters, to the general request to permit the layering architecture shown in [Figure 1].

In any relational database there are only a few data basic operations: select, update, insert and delete. These transformations are shown in [Table 1].

<table>
<thead>
<tr>
<th>Request/Operation Type</th>
<th>SQL Transformed Engine</th>
</tr>
</thead>
</table>
| Select                 | Step 1: Creates the current layer documents/items, D;  
                         Step 2: Run original Statement, R – results; Step 3: R= R ∩ D |
| Delete                 | Step 1: Run "Select" statement using the deletion condition as filtration  
                         Step 2: mark all elements from R to be <end of life> in this layer  
                         Step 3: create a new layer of data, importing all data available from previous layer |
| Insert                 | Step1: create a new layer of data, using old data.  
                         Step2: adds the current elements to the current layer |
| Update                 | Step1: Run "Select" statement using the update condition;  
                         Step2: Run “Delete” statement using the update condition  
                         Step3: Run “Insert” statement using the update data |
An example implementation can be seen in [Table 2]. The current approach does take into consideration the following:

- it is aimed at long-term database systems[2][5];
- it is aimed as a transparent way to access the stored data and its dependencies;
- it has been developed to limit the amount of existing algorithms/procedures/software modifications which in turns reduces developments costs, and down time upgrade;
- provides a multi level way to access the data, transforming the database in a pool of information evolution of data, which in turn can provide useful information in terms of security, data-mining and information projection[3];
- it provides the necessary access granularity to certain roles (beneficiary, designer, manager etc) [1], giving an edge in terms of software design and data manageability, by observing trends in coherent data. Usually algorithms are build using one layer of data without history attached to it [7], therefore using a transparent database versifier will give them an edge in terms of manageability an ultimately on general costs;
- it is been built to serve as a gateway between the request of data and the actual representation of it, giving it the necessary “space” to build an internal model;
- it is been build upon SQL (Structured Query Language) with no engine preference what so ever, providing us the tools to be implemented in various engines starting from MSSQL, PostgreSQL, Oracle or MySQL.

In order to achieve all of the above we had to develop the here by solution in such way a developer can use it, provide the performance needed and be as compatible as possible with the existing developed software. In our test implementation we have been using as a base database engine a MySQL server (InnoDB is must in choosing the table storage, which provides the necessary key constraints [9]), without any special customization, but it can be ported to any SQL supporting DBMS (and a table storage that support the InnoDB properties) with little modification, if any.

### 3.1 Algorithmic approach

In any relational database there are only a few data basic operations: select, update, insert and delete. Given the nature of our approach we do not take into account the alteration of the schemata, as we specified above. In order to achieve the database versioning first we need to meet certain perquisites in terms of tables to be built in the host database, as follows:

- Revision Space, here we specify the revision descriptors;
- Revision space version, here the algorithms stores the date and de revision version of the Revision Space, giving the ability of the revision system to acquire and retrieve the information stored in a layering kind of way;
- Document Space, the data which has a revision has to have a document descriptor, which in turn holds information about the revision (revision identifier and revision version), user rights (if it is necessary), other security information.

In actuality the solution presented, consists in four different algorithms that treats each of the basic operation in a special manner, basically it rewrites the request sent to the DBMS Server, a man-in-the-middle kind of approach [Figure 2, right side].

### 3.2 Request Transformation, Test Case

In the Table 2 is shown how a simple operation (insert, delete, update, select) are transformed.
Table 2. Transformation Statements. Test case scenario, a MySQL Implementation

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select</td>
<td>select * from SomeTable where field1=value</td>
</tr>
<tr>
<td></td>
<td>select t.* from SomeTable t , (select d.id idDocument from Revisions r, RevisionVersions rv, Documents d where r.id=rv.idRevision and rv.version=&lt;RequestedVersion&gt; and (d.validVersionIn&lt;=rv.version and rv.version&lt;d.validVersionOut) and d.idRevision=r.id) r where t.idDocument=r.idDocument and field1=value</td>
</tr>
<tr>
<td>Delete</td>
<td>Delete from SameTable where field1='value1'</td>
</tr>
<tr>
<td></td>
<td>a. The delete statement can be run only on last version</td>
</tr>
<tr>
<td></td>
<td>b. Select all fields, using select and the condition provided in delete</td>
</tr>
<tr>
<td></td>
<td>c. For each element retrieved using document apply : update Document set versionValabilityFinal=&lt;lastVersion&gt; where idDocument=&lt;IdDocument&gt;</td>
</tr>
<tr>
<td></td>
<td>d. Create a new layer of data</td>
</tr>
<tr>
<td>Insert</td>
<td>Insert Into SomeTable set field1='value1', field2='value2'</td>
</tr>
<tr>
<td></td>
<td>a. Create a document for each insert executed;</td>
</tr>
<tr>
<td></td>
<td>b. For each element retrieved using document apply : Insert Into SomeTable set field1='value1', field2='value2', idDocument=&lt;IdDocument&gt;</td>
</tr>
<tr>
<td></td>
<td>c. Creates a new layer of data</td>
</tr>
<tr>
<td>Update</td>
<td>Update SomeTable set where field1='value1', field2='value2' where id=value</td>
</tr>
<tr>
<td></td>
<td>a. Select all fields, using select and the condition provided in update statement</td>
</tr>
<tr>
<td></td>
<td>b. Delete all documents using previous statement</td>
</tr>
<tr>
<td></td>
<td>c. Insert elements using insert statement described previously</td>
</tr>
</tbody>
</table>

As it can be seen, there is considerably overhead, considering that all basic statements, with exception insert is using a select statement at its core, but we gain:

- No modifications regarding your basic algorithms is needed (with the exception of providing the version on which one is working, but the default version is the head one);
- Layered structure of the database is achieved;
- Security is provided, been recorded all the modifications.

4. Results and tests

In this chapter we will present a time comparison between our transparent SQL approach against operation sent on normal channels. It takes into consideration the execution time of static algorithm and the time spent on an actual DBMS server, in our case MySQL.

Table 3. Results at 1000 repetitions

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Time, direct access to DBMS</th>
<th>Time, access though our proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>0.7627</td>
<td>4.894</td>
</tr>
<tr>
<td>Select</td>
<td>0.545</td>
<td>2.0786</td>
</tr>
<tr>
<td>Update</td>
<td>11.6997</td>
<td>13.0644</td>
</tr>
<tr>
<td>Delete</td>
<td>0.8141</td>
<td>7.0818</td>
</tr>
</tbody>
</table>
As it can be seen, from the [Figure 3][Table 3], the insert, select and delete operations are considerably more slow then the basic ones, but the update, which in our case, interests us are is only 10% slower. It can be considered a good result taken into consideration the gains that offers such a schema in representing data at the database level. We achieved by using all levels of data optimization in creation of table structures and by indexes added to the core tables [9].

5. Conclusions and further developments

The presented solution has been used in the management of National Plan funding Agency (UEFISCDI) for over four years and has reached its potential, but there are a few enhancements that can be implemented: a). using cached tables when a first select operation is issued to the server, however there is a need of implementing a way to clean up this structures upon update, delete, operation, not to mention alter schemata [9]; b). removing Generic Document Storage, we use a central governance of all items in our database. It can accommodate databases up to 15 - 20 millions of item (row/table) data versification but there is a need for more, a more distributed architecture is needed.

References