Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 55 (2015) 523 – 528

Procedia Computer Science

Information Technology and Quantitative Management (ITQM 2015)

Database versioning 2.0, a transparent SQL approach used in Quantitative Management and Decision Making

Cosmin Cioranu[a], Marius Cioca[b,1], Carmen Novac[b]

[a]*Executive Unit for Higher Education, Research, Development and Innovation Funding, Bucharest, Romania*
[b]*"Lucian Blaga" University of Sibiu, Sibiu, Romania*

Abstract

Managerial decisions are based on accurate information and in today's time raw data is produced even with a stroke of a key. Regardless of the data creating process one needs to know how the information was extracted and which pool of data was used. One important factor is time therefore we need to structure it in layers of data history in such a way that it can be analyzed, (post)process, in order to be able to retrieve valuable information. The simplest way is to use a Database Management System (DBMS), but even with such a management system we face the issue of making it a self-contained database on each version of data added. Our proposed system, a continuation of previous work, aims toward creating a database versioning system which keeps the natural dependency between data on each internal revision, a basis of security and alteration control mechanism, trend analytics, without sacrificing(within acceptable levels) speed, flexibility and the cost of implementation be kept as minimal as possible.

Keywords: DBMS, SQL, Database Logical Version Control, Decision Making;

1. Introduction

Managerial decisions are based on accurate information, but in today's time raw data is produced even with a stroke of a key. No matter how the raw data was created (automated or manual systems) one needs to know how the information was extracted and which pool of data it was used. One important property of data is time, but because we are living in a fast and ever growing space, an informational space, we face the need to slice it and restructure it in historic layers[7]. If process requirements do not ask explicitly for historic layer mechanism(tracking information in contextual blocks) in long lived databases [10], produced by certain algorithm or advanced applications such as CAD/CAM, Geographic Information System (GIS) [10][5][6], simple but not simplistic architectural design of tracking modification is enough. Most radical decisions are based on informational trends therefore it makes sense to create and invest in such storage engine/architecture. One simple and yet robust approach in achieving historic layering is a file system,

*Corresponding author. Tel.: +4-021-302-3899.
E-mail address: marius.cioca@ulbsibiu.ro.

but in most designs it lacks the necessary flexibility, partitioning of data and/or security. Some of the issues can be addressed by using control version systems (such as Concurrent Versions System - CVS, Subversion - SVN) but it also lacks the necessary granularity in security and data access. Another way is to lean towards a Database Management System (DBMS) which allows all the above requirements in terms of data access, data partitioning it up to the primitive slices, security and most of needed granularity[11], but here we yet another issue, we moved the issue of data storing structure but not the issue of historic layering. Systems based on Common European Research Information Format(CERIF) standard [12] have been employing a time-stamping storing strategy but without any means of keeping track of the modification between various sessions of writing. Most today's DBMS have a built in system of versioning data with or without redundancy, automatic or manual backup [3] controlled by administrator and even am advanced support as a combination between all of them(Oceanstore, PASIS) [3], but in most cases it is not enough, in terms of data history and dependency retrieval. The proposed solution of data versioning is based on previous work [1] and aims towards informational history and dependency, alteration control, trend analytics, without sacrificing(within acceptable levels) speed.

2. General overview over the solution

The paper presents a revised version of the previous work, taking into consideration the experience gained in production environments. The proposed version is aimed toward solving issues regarding various versions of the data(such as foreign key constraints). Data comes in various forms (internal/external process point of view) therefore it has to be marked in certain ways (e.g. valuable, intermediary). Of course, the versioning data producing process could be implemented at the application design level but it breaks the concept of data versioning and from decision making process and software engineering point of view it brings unwanted overhead. This issue is solved by the proposed solution in two flavors an automatic and manual engine . The first involves employing a validation scheme (formal) and the manual engine lets the decision at application level to decide if the information stored is considered a checkpoint (e.g. valuable and intermediary). Another issues solved is the access of any version of data in a transparent fashion, by employing a global version data stamping mechanism, see Fig.1

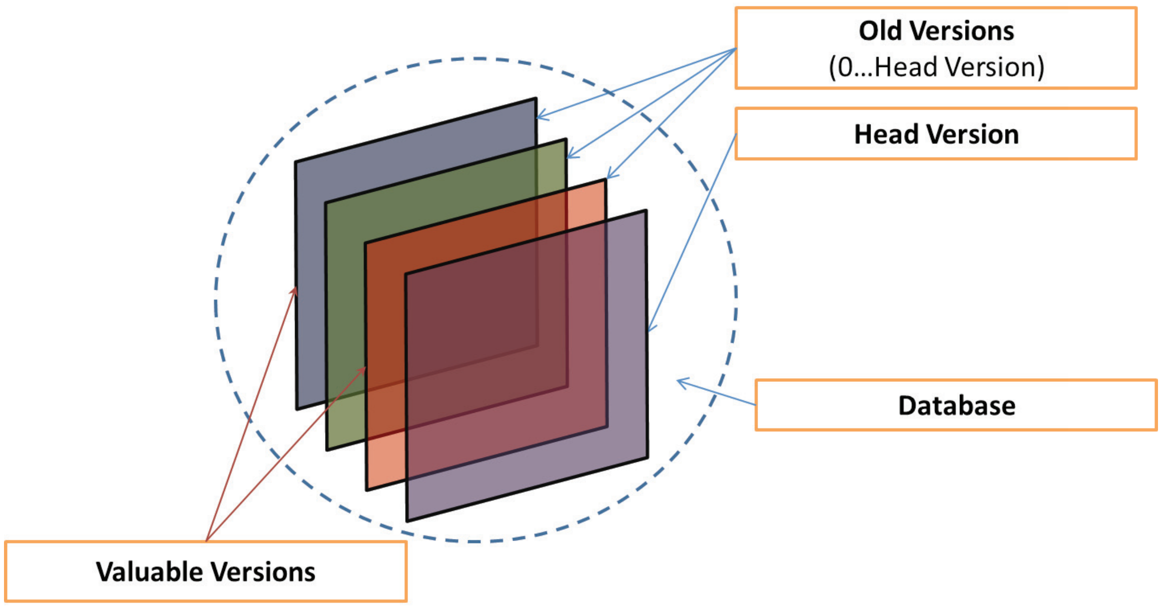

Fig. 1. Basic representation of the solution.

3. Design issues of data versioning

Any versioning system, in our case a transparent Structured Query Language (SQL) approach, has to take into considerations issues like performance, security, maintenance costs and hardware resource consumption [3]. In terms of performance and security, the previous and the proposed solution is mainly a rewriting of the queries sent to the DBMS. The driver or the database abstract layer intercepts them and rewrite them to solve the above issues. The overhead added is reflected directly into the performance and hardware resource consumption. To diminish the introduced impact, a number of improvements can be added, like caches (at driver and application level) [9] but we will not approach them at this point. Also the overall performance is affected by the cost of accessing data versions (e.g. at checkpoints), total number of versions stored in the database and of the completeness of data [3]. From the security point of view, being a query rewriter, the resulting request cannot circumvent the in-place rules (assuming there are no flaws at the implementation level). However, a new security layer has to be considered in terms of the consistency of the layered concept of the stored data, therefore a self-check of the produced request has to be enabled and also and architectural permission rights has to be considered [8]. The hardware consumption is closely related to the performance overhead introduced by the rewriter, issues addressed in this paper as a flavor of the previous paper. Also we should keep in mind that the cost per-byte and processing power, in later years, is becoming lower, for small to middle systems (e.g. small on-line shop, small to middle organization, financing agency) this will not be an issue[4]. Maintenance effort, is minimized, due to the fact 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 [1].

4. Approach

The general approach of the proposed solution in this paper and also in the previous one, in terms of architecture, is the *man-in-the middle or proxy* see Fig.2, but with some additions. We also have kept the general rule of flexibility and we started from a technology of Open Database Connectivity (ODBC). In the left we see the generic or usual approach, every query, or request sent to the DBMS as it was written, in the right, our solution, transforms it, by adding certain parameters to the general request to permit the layering architecture shown in Fig.1.

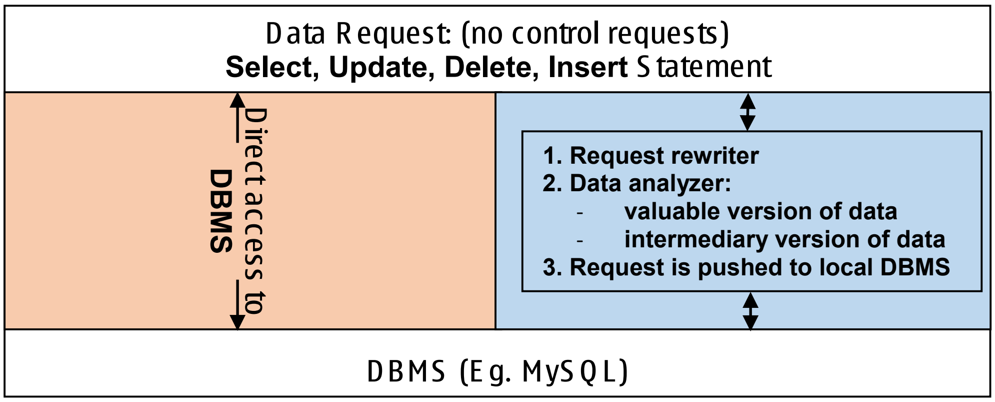
Data Request: (no control requests)
Select, Update, Delete, Insert Statement
Direct access to DBMS
1. Request rewriter
2. Data analyzer:
- valuable version of data
- intermediary version of data
3. Request is pushed to local DBMS
DBMS (Eg. MySQL)

Fig. 2. Generic approach.

4.1. SQL Prerequisites

The SQL implemented in most relational database systems offer a small range of statements aimed toward data manipulation. Our solution is closely related to those statements, the select, update, insert and delete structures. The implementation of the current solution, as mentioned before, is a rewriting mechanism [Table 1], the transformations are different than in the previous solution, aimed toward performance, in an attempt to isolate and eliminate bottlenecks. As we could see we are not taking into consideration the modification at schemata level.

The current approach does take into consideration the following:

- it is aimed at long-term database systems [10][5];
- it is meant as a transparent way to solve the issues related to versioning or the historic layering of data;
- it is been built 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;
- it is been developed in such a way that limits the modification/cost required to be implemented into an existing software architecture;
- it provides a multi-level data granularity [2] made possible by introducing the valuable vs. intermediary versions of data) which may be useful in terms of security, data-mining and information projection[3];
- 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, giving the necessary flexibility to add modules or various extensions.

Table 1. Transformation requests.

Request/Operation Type	Previous SQL Transformed Engine	Proposed SQL Transformation Engine
Select	Step 1: Creates the current layer documents/items, D Step 2: Run original Statement, R - results Step 3: R= R D	Step1: Add transformation to the SQL Statement using requested version (or last)
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	Step1: Transform delete statement into update statement Step2: Check local version and add conditions to the update statement
Insert	Step1: create a new layer of data, using old data. Step2: adds the current elements to the current layer	Step1: add data regarding current version Step2: analyze data completeness to evaluate the type of version (valuable vs. intermediary)
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	Step1: Transform statement into an insert statement Step2: invalidate previous statement by using update statements

In order to achieve all of the above we had to develop tools and methods to meet the required performance and compatibility to the existing systems. The solution was developed using a MySQL DBMS, using InnoDB (must be able to use key constraints [9]) as storage engine, but the solution proposed can be used on any DBMS which follows SQL as query language.

4.2. Algorithmic approach

As seen in Fig.1 and Table1 we have been taking into consideration only the (main) data manipulation statements of the SQL language: select, update, insert and delete. The other statements that relates mostly to the Data Definition Language (DDL) are passing through our proxy without modification, therefore we do not take into account the alteration of the schemata, as we specified above [1]. 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, it 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 way;
- Revision Types, valuable and intermediary versions;
- Revision space that takes into account the global database version;
- Revision logger, which keeps tracks of all modifications made in the database. The Document Space seen in the previous version was replace by this logger.

The actual implementation 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 Fig.2, right side. In the table Table2 we will present the actual transformation of a query that it is sent to the targeted DBMS.

Table 2. SQL Statement Transformation(s).

Statement Type	Statement Type	SQL Statement
Select	Basic	select * from SomeTable where field1=value
	Transformed	select * from
		(select d.*
		from SomeTable d, Revisions r
		where r.id=d.idRevision
		and r.internalName=[RevisionName]
		and(
		([requestVersion]=-1
		and d.rVIn¡=r.version
		and r.version¡d.rVOut)
		or
		([requestVersion]!=-1
		and d.rVIn¡=[requestVersion]
		and [requestVersion]¡d.rVOut)
		)
		) rSrc field1=value
Delete	Basic	Delete from SameTable where field1='value1'
	Transformed	a. update SameTable rDest set rDest.rVOut=0 where
		field1='value1' and rDest.rVIn¡=[version] and
		rDest.rVOut¿[version] and rDest.idRevision=[RevisionId]
		b. update Revisions r set version=version+1 where r.id=[RevisionId]
		c. insert RevisionVersions set idRevision=[RevisionId],
		version=[version]
Insert	Basic	Insert Into SomeTable set field1='value1', field2='value2'
	Transformed	a. update Revisions r set version=version+1 where r.id=[RevisionId]
		b. insert into ProiectePCCA2013 set rVIn=[version[,
		idRevision=[RevisionId], field1=value1, field2=value2
		c. insert RevisionVersions set idRevision=[RevisionId],
		version=[version]
Update	Basic	Update SomeTable set where field1='value1', field2='value2' where id=value
	Transformed	a. Create a temporary table with all rows to be update (using select)
		b. Delete all fields using delete statement
		c. Run the update statement open the temporary created table
		d. Insert the rows from the temporary table into target table

As it can be seen there is considerably overhead but the real advantage is the manageability of the resulting data and we gain the following:

- Transparency to the application level;
- Historic layering achieved;
- Security is inherited.

5. Results and tests

The results shows that the version 2.0 of the transparent SQL approach Table2 is better in terms of performance, we gained features like: history value, better offload of DBMS Resources by distributing it across various storage areas. In the previous version all the queries had to go through a center registry.

Table 3. Results at 1000 repetitions.

Request Type	Time, direct access to DBMS	Time, access though our proposed solution
insert	0.7627	4.4032
select	0.545	1.9872
update	11.6997	12.9992
delete	0.8141	6.0321

As it can be seen, see Table3, the *insert, select* and *delete* operations are considerably slower then the basic ones, but the *update*, which in our case, interests us are is only 9% 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].

6. Conclusions and further developments

The presented solution has been used in the software solution that hosts one of Romanian R&D proposal management system (UEFISCDI) for over five years and has reached his potential, but there are a few enhancements that can be implemented:

- The need to use in place caches to retrieve the information on select operations;
- Better optimization to the rewriting algorithm.

References

[1] Cosmin Cioranu, Marius Cioca, *Database Versioning, a Transparent SQL Approach*, Journal of Mobile, Embedded and Distributed Systems; vol. V, no. 1, 2013, ISSN 2067-4074.

[2] Filip F.G., *A decision-making perspective for designing and building information systems*, International Journal of Computers Communications and Control, 7(2), 264-272, 2014.

[3] Ninging Zhu,*Data versioning systems*. Technical report, Stony Brook University.

[4] Craig A.N. Soules, Garth R. Goodson, John D. Strunk, Gregory R. Ganger, *Metadata Efficiency in Versioning File Systems*, Proceedings of FAST 03: 2nd USENIX Conference on File and Storage Technologies, San Francisco, CA, USA March; 2003.

[5] Edward Sciore, *Versioning and Configuration Management in an Object-Oriented Data Model*, VLDB JournaL3; p. 77-106, 1994.

[6] Michail D. Flouris, *Clotho: Transparent Data Versioning at the Block I/O Level*, Twelfth NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with Twenty-First IEEE Conference on Mass Storage Systems and Technologies; p. 315-329.

[7] Ana-Maria Suduc, Mihai Bizoi, Florin Gheorghe Filip,*Usability in Scientic Databases*, Computer Science Journal of Moldova; vol. 20, no.2(59), 2012.

[8] L.I. Cioca, Marius Cioca,*Using distributed programming in production system management*, Transactions on Information Science and Applications; No4(2), p. 303-308, 2007.

[9] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko, *High Performance MySQL, Third Edition*, Lecture Notes in Computer Science Volume 1861; p. 1048-1062, 2000.

[10] Enrico Franconi, Fabio Grandi and Federica Mandreoli, *A Semantic Approach for Schema Evolution and Versioning in Object-Oriented Databases*, Computational Logic - CL.

[11] Marius Cioca, Andrada-Iulia Ghete, Lucian-Ionel Cioca, Daniela Gifu, *Machine Learning and Creative Methods Used to Classify Customers in a CRM Systems*, Applied Mechanics and Materials; vol. 371, 2012.

[12] CERIF 2008-1.3, *Full Data Model: Model Introduction and Specification*; http://www.eurocris.org/Uploads /Web%20pages/CERIF-1.3/Specifications/CERIF1.3_FDM.pdf.