Keywords: relational database, temporal data, program state, alternative states

Tomasz RYBAK[1]

USING TEMPORAL RELATIONAL DATABASE TO STORE AND MANAGE PROGRAM'S EXECUTION PATHS

Article describes idea of system able to store alternate solutions of problem. It is based on temporal data model described by Snodgrass and extends this model by adding concept of time lines. Each time line stores alternate values of data. Article begins with short description of cooperation between databases and programming languages. Next it describes temporal model, and idea of program as sequence of changes of its state. Description of ways that database servers use to durably store changes made to data by client applications follows, together with discussion how it influences ability to restore state of data from the past. Next section introduces concept of time lines, first by describing their implementation in PostgreSQL database by using Point In Time Recovery mechanism, and then limitations of this mechanism and custom implementation of time lines inside temporal relational database. Detailed discussion of implementation together with analysis of emerging problems follows and finishes article.

1 INTRODUCTION

We can observe trend to integrate databases and programming languages. On the one hand most of the currently available relational databases allow for writing stored procedures, and to do it not only in pure SQL. Most databases offer extended SQL dialects; for example Microsoft SQL Server offers T-SQL, Oracle offers pl/SQL, PostgreSQL has PL/pgSQL. Also, all of mentioned DBMS allow for writing stored procedures in other languages. Microsoft SQL Server provides ability to execute inside database server code written in .NET languages; Oracle has ability to run code written in Java. PostgreSQL offers the widest range of languages: there exist ability to run procedures written in C, sh, Perl, Python, Tcl, Java. If this is not enough, there is possibility to provide server with custom language interpreter.

On the other hand, just as databases try to incorporate programming languages, those languages try to include databases, or at least ways of accessing databases, into them. Almost all languages have libraries that can be used to access databases; database access libraries can be tailored for single database like libpq for PostgreSQL, or able to connect to many different databases, for example JDBC for Java, DBI for Perl, ADO.NET for C#. Those libraries create abstraction layer that hides complexity connected to operating on data stored in relational database. They allow for changing chosen database server without requiring to change program, but for the price of ignoring features offered by only one database.

The next step in incorporating data access into languages is mapping between object-oriented and relational model, and ability to store objects in database. Java Persistence API (based on Hibernate library, described in [2]) is example of Object/Relational Mapping (ORM) — a way to join object-oriented and relational worlds, build on different theoretical foundations. It allows for treating database the same way as any other storage space, and to have some abstract general way of storing data. Similarly, LINQ in .NET allows for treating all data sources similarly to relational databases. It allows for writing SQL-like constructs directly in language and to use them to get data from relational database, file, or any data structure.

[1]Faculty of Computer Science, Bialystok Technical University, Bialystok

Border between database and programming language, and between storing data and operating on it is more blurred as integration of languages and databases is progressing. It allows for using features offered by database in programming languages and to use programming languages in databases.

2 TEMPORAL DATABASES AND CHANGES OF STATES

In article introducing transactions as a way of organising operations made on data stored in relational databases ([5]) Jim Gray noticed that similar attitude can be used to reason about changes made in any programming contexts. Programs can be seen as sequence of changes of states. This would allow for using techniques known from databases, like aborting and omitting transactions, in all programs. Using relational database as storage for program makes it much easier, as it allows to use all already implemented transactional features offered by databases, without need for implementing custom one.

Relational model used in most of the databases can be extended to the temporal one. Temporal databases add time as one of data dimensions and allow for analysing changes as function of time ([10]). They allow for going backward in time and observing data at any moment in the past. Ability to look at exact state in any particular moment means that it is possible to observe changes in state, and to predict how each change may result in later changes.

To avoid creation of many incompatible implementations of temporal databases, standardisation steps were made. SQL Multimedia and Applications Packages (SQL/MM) is set of ISO standards describing extensions of SQL. One of its parts is "Part 7: History" ([1]), describing temporal data. It describes recommended way of creating history tables, by presenting types, functions, and stored procedures for working with temporal data. Its idea and presented solutions are very similar to described by Snodgrass ([10]), but more formalised and presenting more details of used functions. Snodgrass focused of ideas and theory, ISO on implementation.

As way of avoiding conflicts with existing names, names of all objects described in standard start with letters HS and underscore (HS_). For each table additional table storing historical data is created. Historical table contains all columns that original table contains, plus additional column with type HS_Hist, consisting of two attributes, HS_HistoryBeginTime and HS_HistoryEndTime. Those attributes describe start and end of period when row was valid. History and original tables have also triggers that are used to update history table whenever original is changed. Additional triggers preventing any changes in history table other than inserting new rows and changing HS_HistoryEndTime attribute from NULL to other value can also be created.

Currently no full implementation of SQL/MM Part 7 exists, but some implementations of concepts presented there can be found. Those are limited to particular database servers; for example contrib/timetravel module implements temporal tables in PostgreSQL, but it uses custom types and naming conventions, not those described in [1].

When used as backward storage, temporal databases allow for storing history of program's states in databases. This allows for easy implementation of undo/redo capabilities. Those capabilities allow for experimentation with data, checking different hypothesis and compare alternative solutions. But the most common implementation of undo forgets all commands after going back in time and issuing any command other than redo. This means that returning to original state after trying alternative solution requires going back to the last shared state and repeating all original commands again. It discourages tinkering with data and slows down process of getting intuition about problem space and possible solutions.

3 STORAGE MECHANISMS USED BY DATABASES

As written by Jim Gray ([5]), databases change their state from consistent to another consistent one. But usually changing state means making changes to many different tables. And here Duration, one of four ACID (Atomicity, Consistency, Isolation, Duration) properties of transactions, comes into our attention. When transaction is committed, it must be stored in durable way. As operations are done on data in volatile memory, power outage, operating system or hardware failure means that all changes are lost. So usually before committing

File name 000000010000000000000001.00000068.backup

```
START WAL LOCATION: 0/1000068 (file 000000010000000000000001)
STOP WAL LOCATION: 0/10000CC (file 000000010000000000000001)
CHECKPOINT LOCATION: 0/1000068
START TIME: 2007-05-10 20:36:24 CEST
LABEL: copy-one
STOP TIME: 2007-05-10 20:40:44 CEST
```

Figure 1: Information about initial copy of database

transaction DBMS saves data into non-volatile memory, usually hard drive. But as said, changes may be made in many tables. Problem occurs, when one table is updated and another is not. This may lead to inconsistency in data.

To protect against this problem, Write Ahead Log is used ([3]). WAL is special file (or set of files) storing all changes made to database. Each database implements WAL differently; PostgreSQL uses 16MB files named with successive natural number, e.g. 000000010000000000000006, 000000010000000000000007, ... As storing changes made by one transaction very rarely requires more than one file, it is less probable that there will be outage during disk operation. Also, in case of failure, only transaction log files are corrupted, not files with actual data. Using log lessens requirements for memory used for operations on data, as there is no need to operate only on data held in RAM. Information about all transactions may be saved in WAL files. In case of abortion of particular transaction, there is no need to revert tables to previous state, but only to mention in WAL that this transaction has been aborted.

Described process means that information about all changes are stored only in WAL files. So in convenient moment, usually when there is no other activity, DBMS will transfer all changes made by committed transactions from log files to real tables. This means that files with tables always contain consistent state from the past; current state of database is stored in WAL, but this state may be inconsistent.

The second function of Write Ahead Log is to help with restoring database in case of crash. This requires state of database from the past, and all WAL files that were created after this moment. During restoring DBMS will perform all operations that it would do during normal mode of work: transfer all committed transactions from WAL to tables, and ignore all aborted transactions.

Mechanisms used by crash recovery can also be used to provide way of travelling into the past. According to Author's knowledge, only PostgreSQL and Oracle 10g allow for database administrator to go back to any moment in the past. Oracle calls this ability Oracle Flashback Query, PostgreSQL calls this very interesting way of creating backup Point In Time Recovery (PITR).

PITR is based on creating full backup and then storing all WAL files. To create backup it is required to save initial state of database by simply copying entire data directory. During this initial copy DBMS creates special file with information about state of itself, especially which WAL files were used and which weren't at this very moment. This file will be used during restoring to find first command to restore. This approach is very interesting. During creating initial backup state database need not to be in consistent state. All changes, and transactions that were active during creating initial copy will be saved later. Sample content of such file with information DBMS state is presented in Figure 1. Then all changes in database are saved in the backup. All changes are saved in order they were executed by the database.

If there is need to restore data, DBMS must be brought to state it was in during the initial copy, so all old files must be copied back to the data directory. Then all WAL files saved by PITR must be replayed again to redo all commands and all changes that were made in database since the initial backup copy was made. All saved files and all commands are played in order they were saved, which is the chronological order. Details of this process are described in [3] and in [6].

The most interesting feature is ability to stop recovering at any moment between initial backup and last issued command. As PostgreSQL uses transaction numbers to identify time, and PITR uses time, it is possible to identify time to which replay commands either as number of the last transaction or as time.

Oracle also has similar feature, allowing for going back to the past state of any table. This feature is called Oracle Flashback. However implementation is different from described PITR. Database administrator creates special area on the disk, where files with previous states should be stored. During creation of this space

File name 00000002.history

```
1 000000010000000000000008 before transaction 8075 at 2007-05-17 17:52:17 CEST
```

Figure 2: Information about new time line

maximum period of time (default value is 5 days) old data is preserved is entered. When there is no empty space left in this area, server removes the oldest data. This way there is no risk filling up entire hard drive (this risk exists when using PostgreSQL and PITR), but it limits time we may go into the past. Unlike in PostgreSQL, where PITR is connected to backup, in case of Oracle backup and Flashback are separated. This solution requires more space on disk, as flashback space and backup are separated. But Oracle does allow for going back in the time without stopping server by issuing command FLASHBACK TABLE.

So these two solution, somewhat similar, emerged as response to different needs, and came from different philosophies. Also none of them will be sufficient as solution for our problem with storing alternative values of data.

4 TIME LINES

Ability to stop recovery at any given moment is very interesting, but PostgreSQL uses it to provide user with even more interesting functionality. If recovery files were not played to the end and then in the database were done changes, PostgreSQL creates a new "time line". Each time line has it's own identifier, so it is possible to choose a time line during recovery. It means that during PITR activity it is possible to change data in the database, than go to the earlier point in time, and make other changes. It is even possible to switch from one time line to another, with entirely different state of data in database. But going back in the time or changing active time line requires stopping server, restoring backup, and replaying appropriate WAL files containing correct time line. It is administrative burden and does not allow for fast going into the past. But it allows not only for looking at previous database state, but also changing state, while previous is preserved in the old time line.

Creation of new time line requires storing information about it, i.e. from which line it was created, what was last WAL file, last transaction identifier and time of last transaction, that was shared between new and old time line. Identifier of new time line is stored in form of name of file with information about this line. Those files are used during restoring to compute relations between time lines. Sample file is shown in Figure 2.

Because each time line is identified by number and this number is part of name of each WAL file, even in case of creation new time line old WAL files are preserved. This means that there can be many existing time lines, and it is easy to distinguish files belonging to different time lines without reading them, only by looking at their names.

At any moment there is only one active time line, so changes are made only on one of them. Other are inactive; they are only stored in files on hard drive and not even touched by database. Each time line but first one has one parent. So time lines create tree, where first one is root, and other are children.

Ability to travel in time and switch between alternative time lines offered by PostgreSQL, although very interesting can not be used in programs, as it requires large amounts of administrative work with restoring database server. But some ideas from implementation of PITR can be used during implementation of custom time lines.

As shown in previous article by Author ([8]), and as described in SQL/MM:7 ([1]), creation of simple temporal database does not pose many technical challenges. It requires adding two columns for each temporarily enabled table and extending primary key, as described in SQL/MM:7. Few additional objects (stored procedures, view, triggers) will allow for old programs to work without changes, but without ability to use temporal features.

Temporal model assumes that there is only one value for each row at any moment. This means that going into the past and changing anything will create at least two conflicting rows, both containing values valid at that moment. It creates inconsistency in data. To avoid this all changes made after the moment of initial change of the past situation are removed. This inconvenience exists because temporal model assumes existence

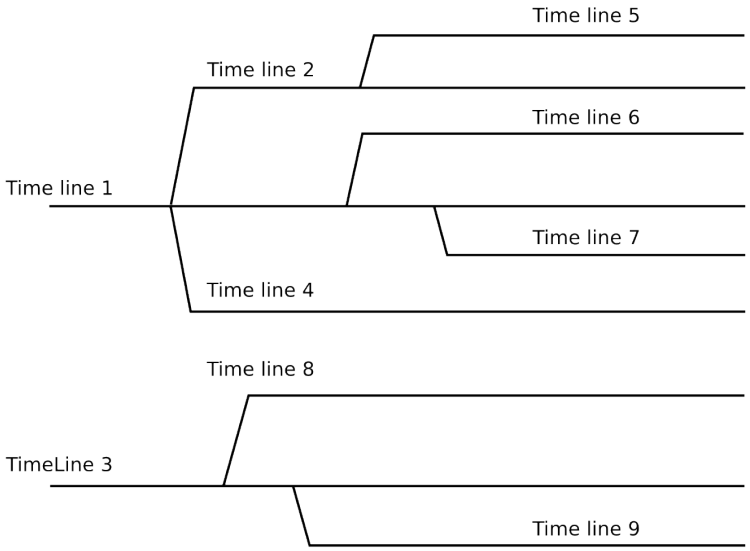

Figure 3: Time lines

of only one time line, so it can not distinguish changes made at two different moments. So, we can go to the past but only to observe. Each change of the past will result in situation that there is no presence to return to.

So to be able to go back in time and make changes and do not lose previous changes, combination of temporal model and solution similar to PITR is needed. Using database as storage space means that all features offered by DBMS can be used, and server is responsible for managing data. This will give programs ability to store many time lines with alternative solutions. Of course it will also mean that more disk space is required to store all alternative lines, but drives offer large amounts of space nowadays.

5 IMPLEMENTATION

Described solution is partially based on implementation described in previous article of Author ([8]).

Implementation of time lines is based on extending tables with additional columns. First two columns describing first and last moment of validity of row, similarly to proposition of Snodgrass ([10], or SQL/MM:7 ([1]), are added. Snodgrass describes two types of tables with time columns. One was state table, containing times at which data was true. Other is logging table, containing times when data was stored in database. These two periods are not necessary the same. State table is about reality, logging about when we know about some facts. As described solution uses triggers to update times stored in additional columns, and those triggers insert current time into time columns, it is logging, not state table. On the other hand, applications can insert custom time into those columns, so it may treat table as state table. However, mixing state and logging tables may lead to problems with consistency of data, and with analysing it. So in certain cases it may be necessary to extend table once more, by adding state tracing columns.

Storing data from the same period but belonging to different time lines means that time is not the only factor distinguishing rows. So just as temporal tables are created by extending ordinary tables by adding columns (and extending primary key), time line tables will require similar extension. Additional column storing time line to which each row belongs need to be added. As each row must belong to some time line, this new column is NOT NULL, and is part of the primary key.

Relationships between time lines can be seen as a tree or a forest. New time line with starting point at first moment of difference is created when there is change incompatible with any existing line. To be able to go forward and backward in time and to switch between time lines, all relations between them must be traced. To

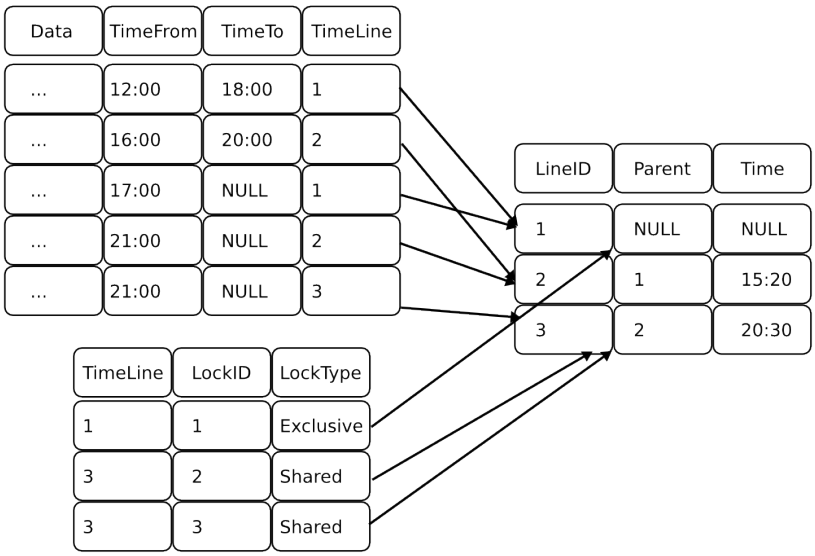

Figure 4: Tables with implementation of time lines

be able to store those relationships, additional table is used. As relationships between time lines are parent-child ones, one to many table is sufficient to store them. If database initially contains only one line, all time lines create tree; when there is more than one state initially in database, forest emerges. Sample forest is shown in Figure 3. Tables and relations between them are shown in Figure 4.

To be able to use time line database with older software, that can not use temporal and time lines features, layer of abstraction is needed. There are two possible layers. Application may know about temporal aspect, but not about time lines. To solve this, one table storing only one row, with main time line, is needed. This also requires creating view that will present only data coming from main time line, and triggers that will add information about time line to all changes made by such application. This situation is similar to PostgreSQL's PITR, which also has one time line. But although here is one main time line, other may exist as well, and other applications may operate on them.

If application can not use temporal aspect nor time lines, it need to be presented with ordinary database. In this case solution presented in previous paragraph may be used to get temporal database. To present temporal database as ordinary one, few additional database object like views and triggers are needed. Details (and also discussion of two possible solutions) are described in [8]. In summary, to present extended database as ordinary one we must hide previously added columns.

To fully use time lines additional other objects are needed. Besides views and triggers as layer of abstraction for older applications, the most important database object is stored procedure (or a view, or function) returning state of any time line at any time. To present access consistent with other tables this procedure may be wrapped in a view. Procedure allows for getting historical data, and data for comparison between time lines.

But travelling into the past is rather complicated when taking time lines into consideration. So hiding complexity is another reason why it is better to put entire algorithm inside procedure, without requiring implementation of it in all applications. Sketch of algorithm (full code is shown in Figure 5) is as follows:

1. If desired moment is after the beginning of wanted time line, return data from this time line similarly to returning ordinary temporal data
2. If not, find which time line is parent of currently analysed
3. Go into this parental time line
4. Repeat from the beginning
5. If desired moment is before beginning of the earliest time line, report error

```
CREATE TYPE row_t AS ( x INTEGER, y INTEGER, value INTEGER);

CREATE FUNCTION getBoard(timeline INTEGER, when TIMESTAMP WITH TIME ZONE)
RETURNS SETOF row_t LANGUAGE plpgsql VOLATILE STRICT EXTERNAL AS
$$
DECLARE
        moment TIMESTAMP WITH TIME ZONE;
        line, parent INTEGER;
        i row_t%ROWTYPE;
BEGIN
        line := timeline;
        SELECT parent_id, creation_time INTO parent, moment FROM timelines WHERE id = line;
        IF FOUND THEN
                WHILE moment > when LOOP
                        line := parent;
                        SELECT parent_id, creation_time INTO parent, moment
                        FROM timelines WHERE id = line;
                END LOOP;
                FOR i IN SELECT x, y, value FROM board WHERE timeline = line AND
                time_from <= when AND when < time_to LOOP
                        RETURN NEXT i;
                END LOOP;
        END IF;
        RETURN;
END
$$;
```

Figure 5: Function returning state of time line at any moment

Thanks to storing state in database program can reside on machine other than database server. When storage exists on separate machine and entire state required to run program resides inside database, it is easy to move running program from one machine to another. Besides, clients can be implemented using any technology and programming language as long as it has libraries allowing for communication with database.

By storing all time lines in database and having ability to run program on separate machine it is possible to run many programs on many computers and to use them to run alternate calculations. As each of clients stores all changes in database, failure of one of them will not cause troubles for entire system.

Not all clients must operate on distinct time lines. Many clients may calculate the same results, probably to check validity of obtained result, or to check behaviour of different algorithms. Other may just read data from database to visualise changes done by applications performing calculations. This is similar to Model (database) View (application displaying) Controller (application operating) pattern described by Gang-of-Four ([4]).

Having different programs working on the same table (whether they work on different time lines or not) requires having way of communicating between client and database, and to notify all clients that database state has changed. This gives clients chance to update their states. Description of few possible solutions, with advantages and disadvantages, is presented in next subsection.

Object-relational PostgreSQL was chosen as a database server to present implementation of proposed solution. PostgreSQL allows for creating custom types, functions and stored procedures. Functions and stored procedures can be written in languages other that SQL; currently interpreters of PL/pgSQL, PL/Python, PL/Perl, PL/Tcl, and PL/sh exist. PostgreSQL contains LISTEN/NOTIFY SQL extension, which can be used to implement communication between clients and between database and clients. However, implementation of this system should be possible in other database servers. Implementation specific details and differences between different servers from the point of view of presented system will follow.

5.1 DISCUSSION OF IMPLEMENTATION DETAILS

Unlike in case of PITR in PostgreSQL, where database both manages and stores time lines, proposed system stores knowledge about lines in applications, but management of those lines and their storage is responsibility of the database. PostgreSQL knows that there is process of restoring of database, and if it has stopped before reaching last of the WAL files new time line must be created. But it is not possible to automatically create new time line in case of described system. Application uses data and it decides if it is necessary to create new line. For database there is no difference between application that used data from the past and started making changes basing on that data, and application that just loaded data from the past, but continued making changes at the present moment. It is just SELECT from the past followed by the INSERT or UPDATE. So database do not know whether creation of new line is needed or not. Application knows what to do, because this knowledge is sealed inside it. So creation of new time line must be explicit, initiated by the application. To make it easier

and to avoid necessity of implementing all steps required to create new time line in all programs database may offer function for creation of new line.

Having knowledge about usage of lines outside database can also create problems with running many client applications. Database server has no information about which program is using which time line. Operating on one time line by more that one client may lead to inconsistency in database. To prevent from this way for locking time lines is needed. However, ordinary database locks may not be used for this purposes. Database locks are fully controlled by database server, not by application. They are used for transaction control, and are only used (and active) during transaction; when transaction ends, all locks are released. When control of the time line must be maintained through entire session, transaction-only locks are not enough. Moreover, some database do not offer locks; when database uses MultiVersion Concurrency Control (MVCC), line PostgreSQL, Oracle in some settings, Microsoft SQL Server since version 2005, it may not have locks available for transactions, or only allow their usage in special circumstances.

PostgreSQL offers advisory locks, implemented as rows in special table; all locks in PostgreSQL are visible through system view pg_catalog.pg_locks. They are managed by using stored procedures to acquire and release locks; their usage is described in Chapter 12.3.4 and Table 9-50 of PostgreSQL documentation ([6]). Advisory locks can be used for purpose of controlling time lines; in case of other database systems they can be implemented by programmer. Advisory locks are controlled by application, no by database server. They can be either exclusive or shared. They are identified by integer numbers, so time line's identifier may be also used as lock identifier. They ignore transaction semantics, and remain active for entire session, until client disconnects.

If program is to use time line, it needs to acquire advisory lock on it. Of course, as they are (as name suggests) only advisory locks, nothing prevents other application from operating on data (and potentially distroying it) without having lock. So applications are assumed to behave according to rules, and always acquire lock before changing data.

Using locks allows for simple communication between applications. If program is prepared to cooperate with other applications, it may get shared lock, so other programs can also operate on the same time line. If program needs time line for itself, it acquires exclusive lock, which is signal for other "this time line is mine and no one can change it". However, using time line need not require having lock; programs only reading data do not need locks. For example application visualising state of time lines only reads and never changes data, so it is not necessary to get locks for this purposes.

Communication by using locks only allows for informing which time line can be used and which one can not; however this does not inform clients when data has been changed. Using the same database by many clients requires having way of notifying programs about occurrence of change of state held in database. If there is no such possibility, applications use old state and compute results basing on old, potentially improper data, or are forced to reload entire state on regular basis — which is large performance loss. On the other hand, notification about change should be treated similarly to exceptions in programming languages. It should occur infrequently, as it requires reloading state of all clients affected by that change. Optimal solution is hierarchy of changes, similar to hierarchy of exceptions, which allows for clients to respond only to events interesting to them.

PostgreSQL offers LISTEN and NOTIFY commands that allow for notifying clients about various events. Source of such event may be any of the active clients, or database (for example by using triggers or rules). Database distinguishes event by name and does not give semantics for events; this is application dependent. Any application can listen for any named event; when this event occurs, application is notified by DBMS.

The easiest solution is to create event notifying about change in each time line-enabled table. However no data besides name of event is send with the event to the application, so such event does not inform which time line has been changed. Of course application with exclusive lock need not to reload anything (assuming that all other applications use locks in proper way), but all other have to. So application still reload their state from database to often in such case. Better solution is to create as many NOTIFY events as many there is time lines, and generate both events: for entire table and for single time line. Name of such event can be created as concatenation of name of the table and name of the time line. However this creates administrative work to manage all events, and requires that all application know to which events to listen to, and to manage all dynamic names of events. On the other hand, proper implementation may connect acquiring lock for the time line with registering to all events that are connected to this time line.

Allowing to manage creation and usage of time lines by applications, and ability to communicate between clients in case of change shows the more complicated problem. Now many applications can work on different, alternative solutions of particular problem, and it is necessary to split work between them. For example finding the best chess game strategy requires finding the best path on the tree of all possible moves. This is excellent problem to solve by distributed systems, because all paths are independent, so there is no need to share data

between clients. However splitting paths to check between clients requires some sort of cooperation between them. Each client must know whatever particular path was checked by other application or not. If this knowledge is not available, there is risk that some solutions will be considered more than once. And while such double (or triple) checking may be beneficial, it must come as result of conscious decision, not as a result of lack of communication between clients.

Unfortunately there is no universal way to solve this problem. Before choosing alternative to consider client can check all currently active (those that has locks on them) and past time lines by watching for first few changes done in them. But this can not always distinguish different solutions. Another possibility is to describe solutions, whether by changing their name or, what is preferred, by adding column to table describing time lines. Such column can contain application specific information about operations performed in this time line. But this is specific for application and may cause troubles for other application — all programs must share and understand semantic of additional column. Another solution can be using managing application, not creating new time lines by clients ready to check part of the solution. Managing program can create all necessary time lines, and possibly add some description to row describing time line. In this case if application finds unused time line, it starts operating on it. But this solution cannot be used for problems where number of alternative solutions is not known in the beginning, or may change as a result of calculations. Another solution, outside of the scope of this article, is to allow for clients to communicate and to reach consensus without using of the database.

Information about state of entire system is held inside database, so it is central point of system; it may become single point of failure and can be performance bottleneck. However, single server is easier to protect again failures and it is easier to make backup copy of one machine, as there is no need to explicitly save state of all clients. There is many literature positions on increasing performance of databases ([9], [7]). In case of performance problems it is possible to use features offered by different databases. They offer table spaces, which allow putting different tables on different hard drives. This makes possible to concurrently access many tables. Many high-end DBMS offer partitioning, allowing to split single table between different disks. If this is not enough, there is always possibility of creating cluster of databases, and to split tables between different machines. However, splitting single time line between machines may create access problems during travelling in time.

6 SUMMARY

Storing different, alternative solutions of problem does not require sophisticated databases. Although presented implementation is far from perfection, it allows for creation of programs. However few open questions remain. One is division of responsibility between database and client programs. Writing entire code managing time lines inside database as stored procedures makes easy unification, and does not require writing sophisticated client libraries. On the other hand it binds library to particular database, and makes it harder to switch to another DBMS. There is no escape from the need of writing client library, even if this is just calling database procedures. As parts of code must exists on both client and server sides, decision should be left to person implementing system. However some guidelines must be made so there is no conflict between different implementations, for example client libraries written for different languages.

Currently no existing data analysis program is able to work with time lines and alternative data presented by them. But there is many ways to export single time line from relational database, as is possible to create views just for analysis purposes. Open question remains, whether to create views for all existing time lines, or just to used ones. Currently databases do not offer possibility of lazy creation of objects, only when there is need to use it. On the other hand, creating one view for each time line, for example by using trigger during creation of such line means that there can be many database objects, which may cause low performance.

But in my opinion, the most significant is problem with understanding alternate solutions and and relationships between data. It may require thinking in terms different from what programmers know and are used to in case of existing languages and programming environments; it may even require creating new data management and programming models.

References

[1] ISO/IEC 13249-7:2005. Information technology – database languages – sql multimedia and application packages – part 7: History, 2005.

[2] James Elliott. *Hibernate. A Developer's Notebook*. O Reilly Media Inc., Sebastopol, CA, USA, 2004.

[3] Michael J. Franklin. Concurrency control and recovery. In Tucker [11]. Published in Cooperation with ACM, The Association for Computing Machinery.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1995.

[5] Jim Gray. The transaction concept: Virtues and limitations. Technical report, Tandem Computers Incorporated, Cupertino, CA, USA, 1981.

[6] The PostgreSQL Global Development Group. Postgresql 8.2.4 documentation, 2006. http://www.postgresql.org/docs/8.2/static/index.html.

[7] Yannis E. Ioannidis. Query optimization. In Tucker [11]. Published in Cooperation with ACM, The Association for Computing Machinery.

[8] Tomasz Rybak. Using object/relational mapping system for analysis of program execution. In *Forum-Conference on Computer Science*, Smardzewice, Poland, 2006.

[9] Dennis Shasha and Philippe Bonnet. Tuning database design for high performance. In Tucker [11]. Published in Cooperation with ACM, The Association for Computing Machinery.

[10] Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, San Francisco, California, USA, 2000.

[11] Allen B. Tucker, editor. *Computer Science Handbook, Second Edition*. Chapmann & Hall/CRC, 2004. Published in Cooperation with ACM, The Association for Computing Machinery.

This work was partially supported by the grant KBN 3T11F011 30, and by the grant W/WI/7/06 from the Białystok University of Technology.