Using temporal relational database to store and manage program's execution paths

Tomasz Rybak

Applied Systems Division
Software Departament
Faculty of Computer Science
Bialystok Technical University
rybak@ii.pb.bialystok.pl

Table of content

Relational databases

- As computers have more memory, programs operate on larger sets of data
- Relational model is very popular way of organizing large sets of data
- Relational databases are in common use, and it is easy to use them databases in programs
  - Libraries dedicated for single databases
  - Universal libraries
    - DBI/DBD in Perl
    - DBI in Python
    - ADO.NET in .net
    - JDBC in Java
    - ODBC

Databases in programs

- Change of state of program, like proposed by Gray
- So it is natural to see database as generalised (or sometimes specific) way of storing data
- ORM links object-oriented world and relational one
- Object relational mapping
  - SQLObject in Python
  - Hibernate in Java
  - NHibernate in .net
- LINQ in .net 3.0 as a way of providing SQL-like language for all possible data sources

Programs in databases

- Microsoft SQL Server
  - Transactional SQL
  - Programs in .net from Microsoft SQL Server 2005
- Oracle
  - PL/SQL
  - Java
- PostgreSQL: C, Java, Mono, Perl, Python
  - PL/pgSQL
  - C
  - Python
  - Perl
  - Java
  - R

Change of state

- Program's execution changes state of memory (variables) ot some external devices
- Transactions, as sequence of SQL commands, change state of database
- Each transaction transfers database from one (consistent) state to another (also consistent)
- Jim Gray noticed similarities between those two models

Temporal aspect of change

- Changes in databases occur in time
- In one moment database has one state, in the next it has another
- So we can store history of those changes
- Temporal databases allow for storing history of changes
- They store all previous values
- This is easy extenstion of relational model
- Add two columns to each table and voila
- We can also add some triggers and views to disguise programs and users and to always present current state of database

Temporal aspect of data

- Undo and redo capabilities
- Ability to make predictions and analysis
- Logging of changes, as required by law

Durable storage

- D (Duration) from ACID
- Two types of memory, volatile and non-volatile
- Only data stored in non-volatile memory is preserved
- Writing changes to all tables at once can be problematic
- Data must be reliably stored after transaction is commited
- It must be moved from volatile to stable memory, and consistency must be preserved
- To have performance it is better to avoid unneccesary changes, when transaction is aborted and we must undo all changes made by it

Write Ahead Log (WAL)

- Is used to save all changes made by all transactions in database
- WAL can be modified without risking changes in real tables, which makes easy to abort transactions
- Later, when not busy, server moves all changes made by commited transactions into tables; server can write to table changes made by more than one transaction
- Changes made by aborted transactions are just left in WAL
- PostgreSQL have WAL stored in 16MB files, and checkpoint when 3 files are created, or every 5 minutes
- Files storing tables present consistent state at some moment in the past and WAL contain changes necessary to move database to the bleeding-edge state
- However, this bleeding-edge state may be inconsistent

Recovering database

- WAL can be used in case of restoring database in case of crash
- State of database from the past is needed — and ordinary copy of files is sufficient; it even need not to be consistent
- But DBMS must know that copy is being made, so it can store information about which transactions are active during copying
- WAL files will be used to make later changes, and to assure that DB is consistent
- All transactions written in WAL are checked
  - Commited transactions are moved to the tables
  - Aborted ones are ignored, or changes made by them are reverted

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: Information about initial copy of database

Point In Time Recovery (PITR)

- Replay does not need to go to the end of WAL
- Restoring of the database can be stopped at any moment between initial state and the last processed transaction
- Administrator or operator can just give number of transaction or moment at which restoring must stop
- Server will stop, even if there are later WAL files
- This allows for going back to the past, and presents undo capabilities
- This undo is rather problematic: server must be stopped, all data directories must be removed, initial database state restored, and all WAL files reprocessed
- However, when in the past, any changes can be made to the database; in other words, we can stop at any point and then make any commands
- This allows for experimentation, but usually after undo and some commands previous resulats are lost

Time lines

- When stopping in the past, server assumes that changes will be made
- As those changes may be different to those made earlier, they may destroy all data
- Old WAL files must be preserved
- PostgreSQL offers time lines
- When during recovering backup not every WAL file is used, server creates new timeline
- All subsequent changes are made in this new line; old one is unaltered
- During recovery it is possible to point to which time line restore to; of course only timelines that started after initial backup copy may be chosen
- This allows for switching between time lines and doing alternate operations, but only during recovery operation

File name 00000002.history

```
1 000000010000000000000008 before transaction 8075 at 2007-05-
```

Figure: Information about new time line

Time lines relations

- Time lines are numbered by natural numbers
- Time line number is part of WAL file number, so WAL file name looks like: TIMELINE-NUMBER WAL-FILE-NUMBER, e.g.: 000000010000000000000006
- It is easy to distinguish files belonging to different time lines without reading them, only by looking at their names
- There is only one active time line, at which changes are being made
- Each trip into the past creates new time line
- Each time line (except the first one) is created from other one
- They create tree of time lines, where first is root and all other are children

Changes in tables

- Creating time-enabled (temporal) database is not difficult
- Two columns must be added tot each table, pointing when period of validity started and ended
- There can be many rows with the same primary key; new PK must be extended by one of the time stamp columns
- At any given moment at most one row with given id can be valid
- But such temporal database can store only one time line
- Any trip to the past and making changes will result in more than one row valid at any moment
- Time lines are needed

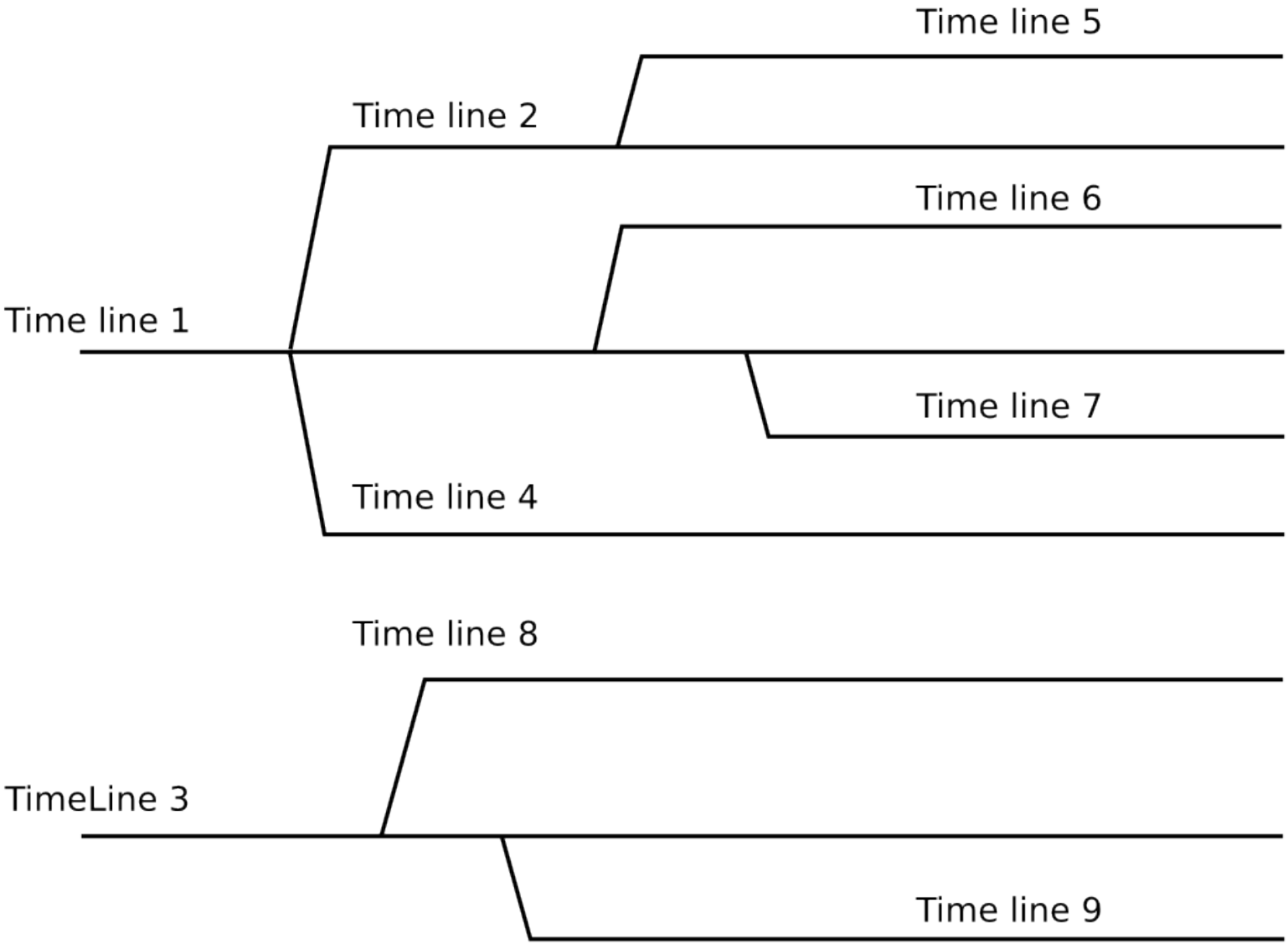

Figure: Time lines

Changes in tables

- Add to temporarly-enabled tables ID of the time line row belongs to
- Create table with description of time lines and relationships between them
- Create table with information about usage of time lines
- Maybe add one more table with one row, holding identifier of main time line; this table serves as compatibility layer for older applications
- Applications aware of time lines concept do not need this table, just like temporal-aware applications do not need views build on top of temporal tables

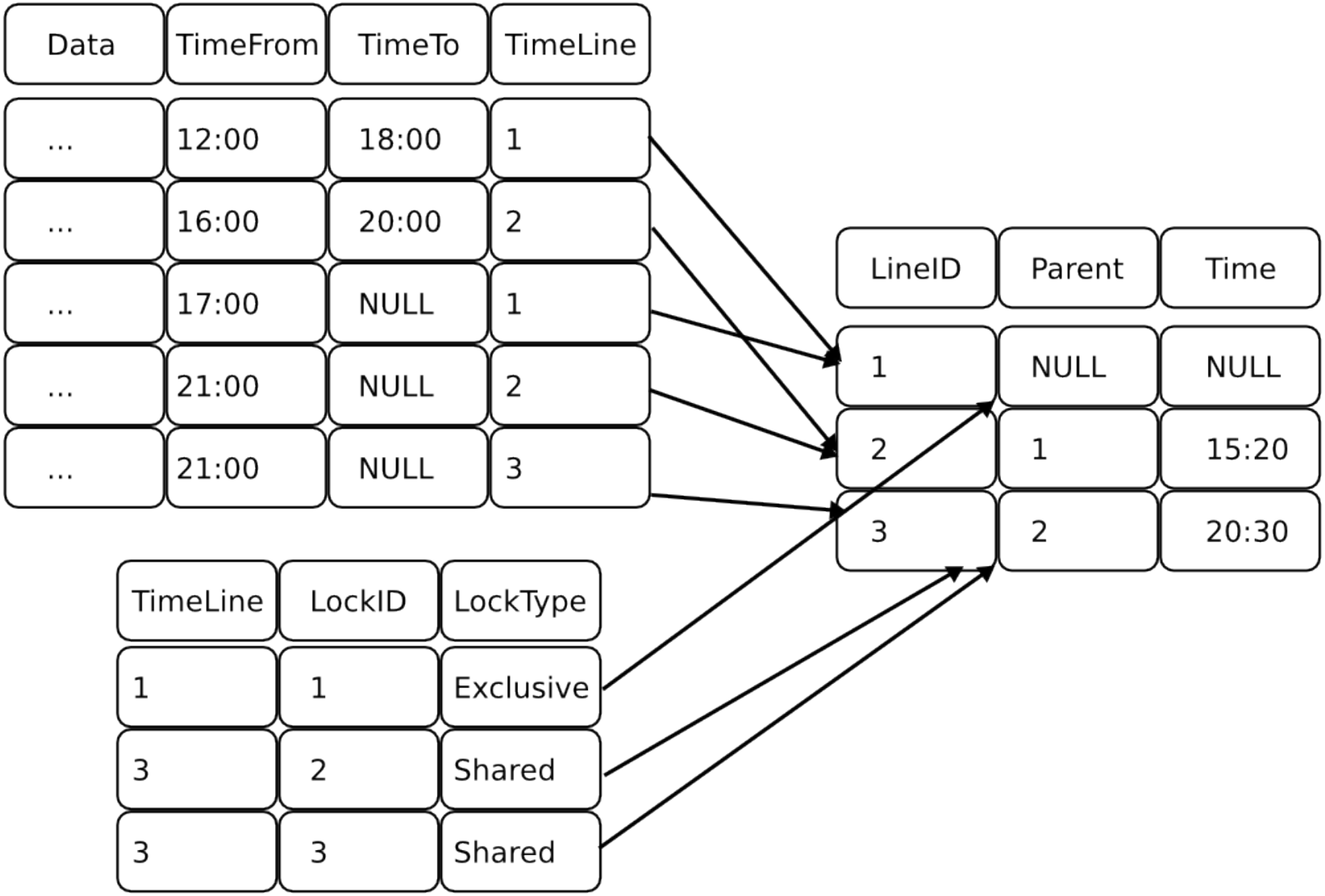

Figure: Tables with implementation of time lines

Additional database objects

- Views for presenting current time line at current moment
- Stored procedures for returning state of current time line at any moment
- Stored procedures to give state of data at any moment at any time line
- When travelling back to the past, it is possible to travel before current time line was created, so it may be neccessary to go to the parent time line
- Algorithm
  - Until searched moment is in the given time line:
  - If searched moment is before start of the current time line, go to the parent time line
  - Check if current time line contains this moment
  - If not, repeat
  - If so, return data as in temporal database

```
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: Function returning state of time line at any moment

Application types

- Many applications can operate on many time lines
- Some may change data, for example by looking for solution in different time lines
- Other may just visualise changes done by other applications
- Similar to Model (database) View (application displaying data) and Controller (application operating on the data) pattern from Desing Patterns book

Time line creation

- Automatic creation of time lines is not possible
- During restoring database server knows that not every WAL file has been used, so new time line must be created
- Here database has no knowledge whether application want to operate on the data from the past (and new line must be created), or is just using data from the past to current operatons
- This knowledge is inside application
- So application must create new time line explicitly
- Database may provide it with stored procedure to do it

Advisory locks

- Applications operating on the same time line may interfere with each other if they are not prepared for cooperation
- There is no way to prevent this without serious performance problems
- However application may be required to acquire locks for time lines it wants to operate on
- PostgreSQL advisory locks can be used to do it
- If application want to have full control over time line, it acquires exclusive lock
- If it can share the same time line, it gets shared lock
- If application do not want to make changes (for example only analyse data, or draw charts) it does not need lock
- However advisory locks are only gentleman's agreement, and rogue application can change data without acquiring them

Notification about changes

- Applications working with the same time line need to be able to know whether data in that line has been changed
- It can be unwise to reload data in each program every N seconds; on the other hand using old version of data can be even worse
- LISTEN and NOTIFY commands can be used to pass messages between database clients
- NOTIFY can be used to send message about change; any client that LISTENs to this message will receive it
- LISTEN/NOTIFY commands use only name of the event, and do not allow for any payload
- Create one event for every table and reload data in case of any time line in this table has been changed
- Or create as many events as there is time lines, and then generate NOTIFY 'table' and NOTIFY 'table+time line'
- This can lead to problems with managing such dynamic labels and channels of communication

Performance of the system

- Database is central server in the system
- It can become single point of failure
- It can also be performance bottleneck
- Performance tuning of the databases is well-known area
  - Divide time lines between disks by using table spaces or inherited tables
  - Use different databases and syndicate them
  - Close time lines should be held on the same servers

Conclusions

- Storing alternate states of programs' execution is possible, but few questions are still open
- Library or framework should reduce problems with using this solution in custom programs
- Visualising tools, especially for showing dependencies between time lines, are needed
- No existing tool for statistical analysis is prepared to work with tables having many alternate (sometimes contradicting) data
- But exporting data from one time line is straightforward
- Lazy creation of (materialized?) view for time lines

Remarks about the future

- Integration with programming languages

Remarks about the future

- Integration with programming languages
- Perl 6 and Transactional Memory

Example

```
async {
maybe {
# Condition
defer if
}
contend {
# Transaction
}
}
```

Questions

Questions?
Thank you for your attention.
Tomasz Rybak
rybak@ii.pb.bialystok.pl