Tomasz Rybak[1]

PROBLEMS WITH STORING TEMPORAL DATA

Abstract: Article shows problems with storing temporal data. It describes tree structures used to store it, and problems with them. Then it describes using relational databases for storing temporal data, and how features provided in relational databases can be used to overcome problems present when trees are used to store temporal data.

Keywords: temporal data, relational databases, data structures, B-Tree

1. Introduction

Thanks to advances in computer technology each user can store more data each year. As Adelstein noted in [2], in last fifteen years disk space available to single person increased over 1000 times. So we store more and more data using enormous disk spaces. But storing large amounts of data makes it harder to retrieve necessary files when needed. We may either remember where each file is located, what is impossible as soon as number of files is more than few dozens, or we may search for them. As noted by Cutrell et al. in [6], users use search programs when they are easy to use yet provide users with enough functionality. When searching, over 60% of searches results were sorted by date. Authors noticed, that we remember files by associating them with time of activities we were doing when creating those files. So being able to access temporal data is more and more necessary to be able to cope with enormous amounts of stored data.

Also, we store not only personal data but also data associated with observed processes, which we try to understand. We want to be able to make assumptions on data, to get information and then knowledge from it. To do it, we must be able to observe it's values, both currently and in the past and on those premises discover what are dependencies in data and predict future values or make theory explaining values present in set. To be able to do it we must store facts together with time it held its values.

Some of problems of storing temporal data are described by Snodgrass in [17] and partially by Gray in [10]. Gray mentioned that introduction of magnetic tapes and

---

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

disks as computer memory make storing history of data more difficult. Previous types of persistent memory were able to be written once and then couldn't be changed, so it was natural to store all previous values of data. It was true both in computers, and in human crafts, like accounting, medical records, and so on. But magnetic memory was expensive and there was temptation to use it again by overwriting old values with new ones.

So now we need to return to storing old values. It puts much request on computers, algorithms, memory and computing power. We need to discover efficient ways of processing large and larger amounts of data.

This article present some of the ways of dealing with problems with storing and accessing temporal data. Section 2 describes data structures used to store temporal data. Section 3 discusses using relational databases to store temporal data. Section 4 describes ways of solving performance problems in databases.

2. Data types

Adelstein noted in [2] that we store so many data, that it is impossible to load it at once into memory, or manually search. Even automatic operation on files which size is measured in terabytes takes time. So we need to use data structures and algorithms that are efficiently operating on data.

Salzberg and Zhang in [15] shown four properties that must be met for a system to be able to efficiently operate on data. They called them *properties of good access methods*, I will call them *principles*. They describe both operations and way of storing data:

1. Data should be clustered on disk according to anticipated queries. Usually data is stored in order governed by their domain. This property is called clustering
2. Additional data (i.e. auxiliary data, used to describe stored values) should use the least possible space on disk.
3. During operations we should minimize number of accessed disk pages with data not relevant to performed operation.
4. Operations performed on local tuples should not change many pages, i.e. operation on small amount of tuples should never require accessing large amount of disk pages.

In other words, principles require that amount of stored data and number of operations should be minimal. Also, algorithms should minimize number of accessed pages during any operation.

Of course these rules, especially 1st principle connected with 4th, assume that there is difference in cost of accessing local and non-local disk pages. It is true when data is stored on magnetic tapes and magnetic and optical disk drives. However when using Solid State Memory disks cost of accessing any disk page is constant and does not depend on previously read page. However magnetic memory offers more capacity and is cheaper than disks produced in different technology so they will remain main storage for data for long time.

2.1   Storing changes of state

Whether we try to store state of entire system or we use data structures to store state, we need a way of saving values and changes that occurred in time. Gray proposed ([10]) two solutions for that problem. Both of them are being used today.

First is to store values together with time of their validity. It allows for easy access to each value at any given time, as it requires accessing appropriate memory fragment representing requested time. This solution is used in temporal databases.

The second one is to store initial values, and then save all changes with time of their occurrence. As this solution stores only changes, usually it requires less memory than first solution. But if one wants to get value at one particular moment he or she has to redo all changes up to this moment.

Implementation of second solution is described in [9], and it's approach for storing data in databases in [14].

In some situations these two solutions are used together. In regular intervals full state is stored, and between them only differences are saved. It joins advantages of both solutions; data takes less space, and fast forwarding in time is possible without computational overhead. It is mostly used in multimedia; in the movie every few seconds there is so called primary frame, which contains information about all pixels, and then there are differential frames, containing only information about differences to the next/previous one. This solution shows most benefits when there is not too many differences between frames.

2.2   B-Tree

B-trees are described in Chapter 19 of [5]. It stores structures (sometimes called tuples), where attributes are used to name it's parts. One or more attributes are grouped together to serve as key. Key can be used to distinguish two tuples. Tuples are stored in order given by their keys, so function able to compare two keys and tell which of two keys is greater is necessary, especially if key is made from more than one

attribute. Sometimes all keys must be unique, but it is not required by algorithms operating on B-tree.

We try to minimize number of operations according to principles. Number of operations is measured as function of number of tuples stored in tree, noted as N. Ways of calculating and describing cost is described in Chapter 2 of [5]. The most common operations are:

- inserting tuple into tree
- removing tuple from tree
- finding tuple in the tree
- finding elements whose key holds some property (related to previous one)

Each node in B-tree contains between $B/2 - 1$ and $B - 1$ objects stored in order according to key values. Each node except leafs has from $B/2$ to B child nodes. Each child node contains tuples with key values between greater than key before this child, and less than following key in parent node.

In B-tree all operations for only one element are proportional to height of the tree. It's height is $O(log_B N)$, where base of logarithm B depends on number of keys stored in each node. So the more keys are stored in the node, the less tall is tree, and the less is the cost of operation. To make sure that tree has regular size, i.e. all leafs have the same distance from the root, operations on nodes that will ensure this are performed during inserting and removing values. But changes in tree are done only on one path from root to leaf, so their cost is $O(log_B N)$. If inserting values results in node having more than $B - 1$ children, it has to be split into two nodes. After split each of nodes has $B/2 - 1$ objects, so they are divided between new and old node. If, as result of removing object, node has less than $B/2 - 1$ objects, it is merged with other node.

Because all values are ordered, searching for keys in range may be implemented by finding first and last key in range and then traversing through all tuples between them. It requires less operations than accessing all keys. However it is not always possible to use such optimization. In many cases there is need to find some tuples when exact key value is not known, but only conditions to be met by key will require going through all tuples.

B-tree is used to store large amounts of data, usually so large that it cannot be put in main memory and must be stored on disk. As each node to be processed must be read from disk and each disk operation is much slower than CPU operation, we try to minimize number of disk pages read. Because reading entire disk block takes the same amount of time as reading fragment of page, in B-tree node size is equal to disk block size. As noted earlier, time of most operations is logarithmic with base

equal to the number of tuples is node. As size of node is constant, number of tuples depends only on size of tuple. So there is tendency to have more tuple stored in one node.

B+-tree allows for storing more values in node. It stores full tuples only in leafs; inner nodes store only keys. It allows for even bigger values of B, so tree has less height, hence less reads from disk. But such tree uses more disk space, as keys are repeated. They are stored in inner nodes, and also in child nodes, together with the rest of object. When keys have the same length it is possible use binary search instead of linear search to make searching for particular key faster. It is the more efficient the more keys are in one node. Sometimes, to be able to put as many keys in one node, keys with the same prefix are compressed, and only their different part is stored, and common prefix is stored once for node. But this attitude makes algorithms more complicated and resulting tree are not so regular. Those changes are described in [15]

In some cases hashing structure, described in details in Chapter 12 of [5] and partially in [15] is used for storing data. It is, however, not used as frequently as B-tree. It uses special functions, called hashing functions and because of way of using it, is extremely slow when there is too many tuples stored. To be faster it would require another hash function, and it would require rebuilding entire hash. But main reason why it is not so frequently used is because it stores data in unordered fashion. It can be used for searching only for equal values, not for ranges. Also, because of properties of hash functions, similar values are stored in different disk fragments, so principle 1 is not met. More detailed description of reasons that hashes are less popular is in [15].

2.3 R-Tree

Adding time to data set adds one or more attributes to stored structure. This additional attributes are used to search data (e.g. find values of all tuples from given moment) so it can be seen as key. In result we have two keys: a main key, previously present in data, and a temporal key, being time in which tuple had those particular values. So we have got a multidimensional key, and using B-tree is not enough as it is able to deal only with one-dimensional key. It will be shown, that meeting four Principles is very hard when having data that cannot be ordered according to only one key.

R-Tree, described in [15], can be used to store tuples that can be accessed by using many keys; it doesn't prefer any of keys and treat each of them in the same way. Because of that it is mainly used to store multidimensional, spatial data. But even though R-trees are not used to store temporal data, they present interesting problems similar to those present in storing time-oriented data.
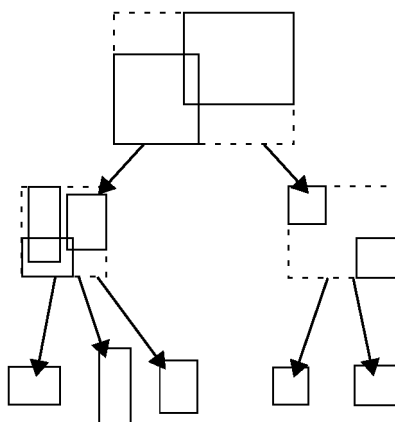
Fig. 1. Sample R-Tree

R-tree assumes that key is point or rectangle (box in higher dimensions) in M-dimensional space, and each point can be described by M numbers. Each node of R-tree contains multi-dimensional rectangle (hypercube) in which all tuples, both belonging to this node and it's all descendant nodes, can be put. If one of keys is non-numerical, it must be made possible to perform the same operations as for numerical values, in particular defining range of values, sum and product of two sets and determining if value lies in particular range. Nodes located higher in a tree has their bounding boxes computed basing on their child nodes. Bounding boxes of child nodes can overlap, which means that some fragments of space are described by more than one node. It makes searching not so efficient as it was in B-trees. Sample 2-dimensional R-Tree with overlapping bounding boxes is shown in Figure 1.

During inserting tuple bounding boxes of all nodes up to root must be updated, so 4th principle (locality of changes) is violated. During removing tuple hypercubes should also be updated, but this is often omitted for performance reasons. It makes removing faster, but later searching is slower, especially when searched keys values are not found, as more nodes has to be inspected. As in B-trees, inserting into full node requires splitting. But unlike B-tree, there is more than one way of dividing keys to nodes.

One of algorithms is described in [15]. There are two seed tuples chosen. One remains in old node, one is moved to new node. Each of the remaining tuples is tested to two nodes, and is places in this, in which it requires less increase of volume of bounding box. If one of nodes reached minimal number of elements, remaining

elements are put into the other one. This algorithm's main disadvantage is that when tuple is added to one node, it's part may belong to bounding box of another node. This complicates searching, as many nodes must be visited during search.

R*-trees try to solve this problem. They choose nodes and seed points in a way ensuring that nodes will share minimal volume of space. However, this requires visiting more nodes in a tree, so locality principle is violated again. It is price for speeding searching, but even in this case there can be necessity to search many nodes when tuple is not found.

Another genre of R-tree are hR^{Π}-trees, very complicated in implementation. They ensure that bounding boxes do not share common points. Inserting is done in three steps, and there can be return from inserting function after first phase. Two other, tiding steps can be made later, which can be faster if they are done for more that one inserted tuple. But even in those trees searching for element not existing in a tree is slow, as more than $O(log_B N)$ nodes must be searched to be sure that tuple is not in tree.

All R-trees store the more additional information the more dimensions space have. Because each node must store a bounding box, and each object must store its space coordinates, memory requirements are $O(N^M)$, where M is the dimension of space. Additional memory usage means that each of nodes can store less objects, so a tree has more height.

Different way of storing spatial data is presented by structure called z-order, similar to hash. It takes bits from binary representation of key attributes, and from this bits a integer number is created. This number is used as a new key to store object in ordinary B-Tree. Advantage of this solution is that the key size does not depend on number of dimensions. But there are many disadvantages. In many cases, especially when tuples are located near each other, many of bits will be the same, so many keys will be very similar. As z-ordering uses attitude similar to hashing, tuples located near in space may be located in different places on a tree, and one node can hold tuples located in different places in space. This depends on bits used for creation of key, and they cannot be changed after creating a tree. Such change would require rebuilding the entire tree and during this operation no other operations on data can be performed. Searching for ranges of values is inefficient, as is in hashes. However, authors of [15] noted that there is one algorithm allowing for searching for range, but it requires reading more disk pages that is necessary, thus violating 3rd principle. But, as z-order uses B-tree, there is only one order. So z-order is not popular and none of popular databases uses it.

2.4 Temporal structures

There are many structures created to efficiently store temporal data, described in [15]. They are created under an assumption that time is another key, but it is very special key. Some of temporal operations that require using this special key are:

– find all tuples from time T
– find a record with particular key at time T
– find records with keys in range at time T
– find all values of tuple at any time in past (searching for history).

One of temporal structures is WOBT, Write-Once B-Tree. This B-Tree contains additional temporal key, so operations can be performed both on main and temporal key. It assumes that any given disk block can be written only once, and then is accessed in read-only mode. So it can be used for reliable logging, if it is required by law. Even on ordinary magnetic drives each disk block will be written only once, so historical data will be always available. Of course rewriting unused pages would lessen disk space requirements, but ability of accessing all nodes as they were in the past has more advantages than less memory usage.

In WOBT each page stores range of keys from some time slice. Searching is done as in an ordinary B-tree but taking time value into consideration.

Inserting key (and also changing it, as changing a value is equivalent to removing the old value and inserting a new one) requires writing at least one new page. When split is needed, only one key is used to split at any given time. If splitting is required for both key and time, first key-split is performed before time-split. Splitting is done in a way requiring upgrading only one parent node, storing information from last time period. This means that only youngest parent is upgraded, and older parent nodes are not updated. They still can point to active data, so they cannot be moved to archive; all pages must be treated in the same way, because without traversing through the entire tree there is no way of knowing if each of nodes points to current data.

As result of splitting there can exist many nodes pointing to the same tuples, and each node can have many parents. So it is not a tree, but rather a directed acyclic graph (DAG).

This problem is not present in Time-Split B-tree (TSB). This tree is split by using only keys. Unlike in WOBT, time chosen for a new and an old node can be any time, not only the current one like in case of WOBT. But it must be the moment before any of currently active transactions has started. Splitting according to any time, not only current moment is unique for TSB. TSB allows for creating disk pages (and nodes) with only inactive data (data from the past, changed or deleted afterwards). It

allows for moving such pages to an archive, for example to other drive, which makes management of disk space easier.

Another structure for storing temporal data is Partially Persistent R-tree (PPR). It is an R-Tree with time as an additional key. It has the same performance as R-Tree during searching, but inserting and deleting nodes is much more complicated. It is a directed acyclic graph (DAG) storing differences between trees that have been done in time. There are many trees, where each stores data at each moment. Some of trees share nodes, so there can even be many roots. Both during inserting and deleting there can be a split, always according to the current time. The most recent root points to the tree with active ("alive") data.

3. Databases

Temporal data can be stored in relational databases (described by Codd in [4], an article introducing relational databases); one can manage them using all features that can be used when dealing with non-temporal data, like access through Structured Query Language, transactions (introduced by Jim Gray in [10]), database constraints, etc.

There are three types of temporal values, as described by Snodgrass in [17].

instant something happened at a particular moment
interval length of time
period length of time, but the anchored one, with a starting moment defined

Similarly to data structures described in Section 2., adding time to relational database requires extending primary key. Usually we add two additional columns, the one with start of period of validity and the second with end of such a period.

This means that there can exist many rows with the same values of columns of the old primary key. So now we must add time to a primary key, to be able to distinguish rows. This approach will allow for easy retrieving all historical values. Simply we ask for all rows with a given value of the primary key, without using time, and DBMS returns all values, the current and the historical ones.

3.1 Standards

As relational databases was becoming more and more populer, different communities tried to extend SQL by adding custom types and keywords to better serve them. To prevent from creating many incompatible SQL variants, International Standards Organization (ISO) created SQL Multimedia and Applications Packages (SQL/MM), standard extending SQL with types usefull in different fields. It consists of six parts:

1. Framework
2. Full-text
3. Spatial
5. Still Image
6. Data Minint
7. History

Currently different databases implement different parts of SQL/MM. Most commonly supported is full-text part, whether by native types (Microsoft SQL Server, Oracle), or by extensions (contrib/tsearch2, full-text search suite in PostgreSQL). Spatial standards are managed by OpenGIS consortium, and extensions for different DBMS are available: MsSqlSpatial for Microsoft SQL Server, Oracle Spatial for Oracle, PostGIS for PostgreSQL. Data mining solutions are available by using extensions.

Description of SQL/MM and it's purposes, and more detailed description of different parts can be found in [13]. Here only "Part 7: History" ([1]), describing temporal data, is discussed.

Part 7 describes recommended way of creating history tables. It presents types, functions, and stored procedures for working with temporal data. Its idea and presented solutions are very similar to described by Snodgrass ([17]), but more formalized and presenting more details of used functions. Snodgrass focused of ideas and theory, ISO on implementation.

Names of all objects described in standard start with letters HS and underscore (HS_). As SQL/MM is using object-oriented SQL as base, used types can contain attributes.

For each table additional table storing historical data is created. Historical table contains all columns that originaln table contains, plus additional column with type HS_Hist, consisting of two attributes, HS_HistoryBeginTime and HS_HistoryEndTime. Those attributes desribe start and end time 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 are created. However those triggers disallow correcting any mistakes, so wrong values remain in history tables forever. System similar to this, but without triggers disallowing changes in history tables, was created and desribed ([14]) by Author.

Three main parts of standards are Chapter 5, describing contcepts and presenting examples, Chapter 6 describing procedures used to implement it, and Chapter 7 describing neccessary types. Described procedures are used to create all objects, like

130

tables and triggers, neccessary to manage historical data. Functions are able to cast temporal types to other types supported by database, and provide input and output methods. Also, functions for checking temporal predicates (described by Snodgrass, e.g. checking if moment is in particular period) are described, for example HS_Intersects for checking whether two periods have common subperiod.

SQL/MM Part 7 is yet to be fully implemented by any commercially available database.

3.2 Implementation

As all operations are executed inside transactions, their implementation is crucial to assuring validity of data. The most important property from point of view of data consistency is Isolation. It requires that each transaction sees version of row that existed at its beginning and all changes made by itself even if there is many transactions accessing this row. If it is not, many problems with consistency of data (like non-repeatable and phantom reads, described in [12]) may occur.

The easiest implementation of transactions can be, that each row (or table, or any other object) has its own lock. When transaction wants to read data, it gets read-lock; when it wants to write data it gets write-lock. There can be many read-locks for any object, but only one write-lock. So many transactions can concurrently read data, but only one can write. This solution is easy, but very slow. In this solution there is only one state of each row at any given moment.

The other possible solution, called Multi Version Concurrency Control (MVCC), is present in namy different databases. PostgreSQL stores data in this form nativelly since it's beginning, Microsoft SQL Server uses MVCC as primary way of storing data since version 2005, other databases (like Berkeley DB, Firebird, ZODB) offer this technique as option. DBMS has monotonic counter of transactions. Each transaction is given different number and counter is increased. So transactions that started later has bigger number identifying them. Each row has two additional columns, one with number of transaction before which row isn't valid, and one with number of transaction after which row isn't valid. Each transaction only sees rows with appropriate values of these columns.

MVCC allows for coexistence of many versions of each row; there is as many versions as many transactions changed it. But each transaction sees only one row, one with the largest number of transaction in first column; of course this number must be less than number of this transaction. Example is shown in Figure 2.

MVCC allows for fast read access, as there is no need for locks. Also when one transaction wants to write and other ones want to read, there is no conflict. Only
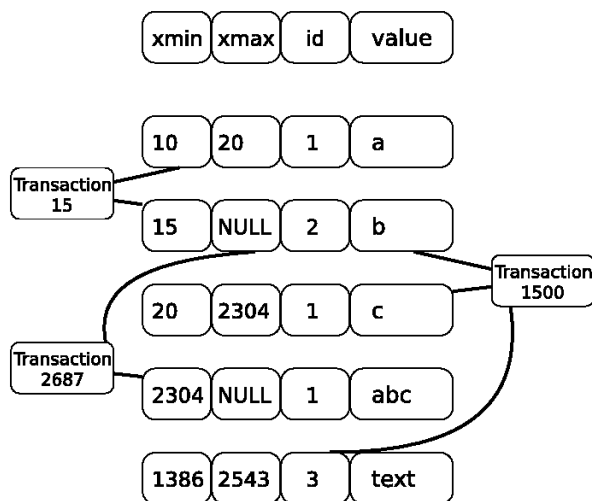
Fig. 2. Multiversion Concurrency Control table

if there are two transactions wanting to change the same row, DBMS uses locks to resolve conflict on write. One of transactions (the later one) is locked until the other finishes, either by commit or by rollback.

However there is cost associated with using MVCC. Each operation changing data creates new, active rows and leaves old, inactive and invisible for all new transactions rows. This is similar to WOBT, where number of taken disk pages also was growing during performing operations on data. But databases using MVCC allow for removing old rows by using vacuum command. This operation takes number of the oldest transaction (it is the lowest number of active transaction) and removes all rows that are not visible by this transaction. These rows can be deleted, because if they aren't visible by oldest transaction (with lowest transaction number) they aren't visible by younger ones (because new transactions have bigger transaction numbers). Vacuuming frees space taken by dead rows, but requires time, because it must check all rows on disk drive. Also it works better when there is no long-lasting transactions. If there are such ones, not every row that should be freed can be. But long-lasting transactions are also problem in lock-based DBMS, because they take many resources, especially locks.

To implement temporal database, we need to take care about managing two additional time columns. When we add new row, we must insert current time as value

of column describing start of validity period. When removing row, it is not removed, but only current time is inserted into column with end of validity period. Changing value of row is really removing old row and inserting new one.

As shown in [14], this changes can be done in existing database schema, without making changes to any of programs using database. Old programs will function without using temporal features, and new ones will be able to use them.

Snodgrass suggested ([17]) that second time column should not be empty in active rows, but should contain moment from distant future. It is consistent with suggestion of Date [7] that no column should contain NULL value. But choosing distant moment is very hard, so in most implementations rows with current data has NULL in their second time column, even though queries that must deal with this must contain additional conditions and aren't so nice-looking as those which are in systems implemented according to Snodgrass' suggestion. Choosing too early end moment means that in some time in future this "future" moment will be in the past, and entire database will be useless as source of temporal data. Also, different databases has different ranges of possible date values, so there is no one maximum date that can be used in many system similar to INT_MAX in C language. So, ISO standard regarding tempral and historical data ([1]) uses NULL as value indicating that particular row is still active.

Sample implementation of temporal database, called timetravel, is distributed together with PostgreSQL in module contrib/spi. Contrib part contains additional functions and modules that can be used together with database, but that are not as popular or has not yet been enough tested to be included into main server.

Timetravel requires adding two columns, date_on and date_off, both with type abstime, for storing period of validity. Default value for column date_on is currabstime(), and for date_off is infinity (PostgreSQL has infinite time, which is moment very far away). This module adds triggers which are fired when current (active, one with infinity in date_off) row is changed or deleted.

For managing time-enabled tables there exist functions set_timetravel and get_timetravel. It enables and disables trigger responsible for operations on data in tables.

3.3   Alternate data values

Most of the commercially available databases (IBM DB2, Oracle, Microsoft SQL Server) allow for creating hot-swap solutions by using master server as souce of events and one or more slave servers as recipients of these ovents. This way each of slaves has the same state as master, so in case of master's failure it can be promoted to

be new master. This technique is using the same mechanisms that are used to recovery after failure in case of single server ([8]). But 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 is called Point In Time Recovery (PITR).

To create backup it is required to save initial state of database by simply copying entire data directory. Then all changes in database will be saved in the backup. Instead of saving new values commands that made those changes will be saved, in order they were executed by the database. PITR is done by using provided by user command for dealing with files saved by backup subsystem. This can be any command, such as compressing, copying files, transferring them using network, as long as it is possible to check if command succeeded. Command cannot require any data from user entered using keyboard.

This approach is very interesting. During creating initial backup state database needn't be in consistent state. All changes, and transactions that are active during creating initial state will be saved later.

This implementation allows using PITR as simple replication system (system for storing data in more than one DBMS, described in [8]), even though it's main function is creating backup. But it is very simple system, and requires manual applying changes; for each changeset an operator must "recover" system as there was an error.

If there is need to restore data, all data files must be removed, old files must be copied back to data directory and then all files saved by PITR must be played again. To use those files, user must provide system with a command to recover them. This command is reverse of the command used to backup files. All saved files are played in chronological order, i.e. in order they were saved.

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 last transaction or as time.

Ability to stop recovery at any given moment is very interesting, but PostgreSQL uses it to provide user with more interesting functionality. If recovery files were not played to the end and then in the database were done changes, PostgreSQL creates a new "timeline". Each timeline has it's own identifier, so it is possible to choose a timeline during recovery. It means that during PITR activity it is possible to change data in the database, than go to earlier point in time, and make other changes. It is

134

even possible to switch from one timeline to an other. Even though it requires restart of the server, it is a very interesting possibility.

Using both PITR and temporal databases can be problematic. When we stop recovering at any point earlier that current, and then resume operations in data there will be a gap in time. Time values will look like there was the end of some period, and then from some time there is activity again. It will look like there is a "silent period". It may or may not be problem, but analyzing such data may lead to wrong assumptions or results.

4. Performance

There are two types of usage of databases mentioned in [18], called OLTP and OLAP.

OLTP OnLine Transactional Processing
OLAP OnLine Analytical Processing

OLTP is a normal relational database, where there is equilibrium between reading and writing operations. It usually contains current data, without many historic records, to be able to process data very fast. It is normalized, so there is no redundancy, and is tuned to serve as many queries per second as possible, and to be responsive even if it is serving many clients.

OLAP is a database used for analysis of data. Data warehouse contains historical records, used in data mining processes to allow for quick answers for questions asked during analysis of business trends.

OLAP database is used differently than OLTP one. It is used by one or few clients, who run long-lasting jobs to calculate report about trends in data; mostly requiring long and complicated computations.

Trend analysis requires analysing large amounts of data, usually from different periods, for different clients or regions, and comparing those disparate fragments of database. Each variable that can be used to distinguish fragment of database important for business analysis (usually relating to some real world concept, like region or client) is called dimension. So set of all those variables (dimensions) creates multidimensional space. When we imagine that all dimensions are located on perpendicular axis, and values coming from database are distributed along them, we will get multidimensional cube. Main usage of OLAP is to divide this cube into fragments and to compare one fragment to another in search for trends and differences important from business standpoint.

As we do not know what is the scale of needed fragments, database must be able to cope with the largest (entire company) and smallest (single customer or shop)

fragments. And on each of those levels we must be able to get all data from all other dimensions appropriate for this level.

But relational databases are normalized and contain minimal amount of indexes to be able to respond fast to usual OLTP needs. This means that queries required by data mining system will be executed very slowly, especially when switching between different variables and levels of detail, because it would require calculating all aggregates for another distribution of variables.

So usually OLAP databases store denormalized, precomputed values for all dimensions and all possible levels of detail. Such denormalized database contains many repeated records, columns and attributes. Usually its data comes from OLTP database (for example every Sunday data from previous week is imported) and during importing there are many changes in structure of database, records, and many values are precomputed to allow faster queries later. However it makes non-read access very hard, if not impossible, as to change anything would require changing values of variables in many places of cube. Such precomputed cube may take much more disk space than ordinary relational database, as data is repeated for each level and each variable. As data size may grow exponentially, it is possible that there will be phenomenon called database explosion. It occurs when there is many dimensions containing sparse data. In such case precomputing creates many empty cubes (but each such record takes portion if disk to write) filling disk with information that there is no information. Moreover, creating such cube will take long time, as all calculations must be done during importing data into OLAP database.

OLAP cubes store higly-dimentional data, so they can be implemented by using previously described data structures, for example R-Trees. But this would require changing algorithms operating on those structures. On each level of tree we would have stored data being aggregation of data stored on lower levels of the tree. Each level of tree would be particuler level of detail for variable. Also algorithms changing trees would not be used, as OLAP database is used only as source of data, and no operations changing data are executed. Even importing new fragment of data only adds few new nodes, without changing shape of the tree.

Temporal databases try to join OLTP and OLAP approach. They contain both current and historical data. The longer temporal database exists, the more historical rows it contains, so current rows become lesser fraction of entire table. So if a query requires accessing all rows, it becomes less and less efficient in time.

This problem can be solved by moving historical data to other place, but it must remain in the database. In current RDBMS it is possible by moving data to an other table or by partitioning existing table. Problem with moving to other table is that analysis will be harder as it will require accessing more tables. PostgreSQL solves it

by allowing for table inheritance. One table may be used as a template for an other. When both of them contain data, reading data from the original table will also read from other tables. This way data can be divided into many tables which may reside on different disk drives, and use the same queries for accessing data.

4.1 Indexing

Shasha and Bonet point in [16] that using indexes makes read operations on data in DBMS faster. Usually for indexes are used B-trees and hashes. As noted in Section 2. and in [15], hashes allow only for eqality comparisons, and B-trees can be used in operations on ranges, so B-trees are used more frequently, as they can be used to optimize more queries.

All available databases use some way of indexing. But different databases offer different levels when trying to index custom types. The most advanced solution is offered by PostgreSQL. This DBMS allows for creating custom indexing structures. User can create two types of indexes, either Generalized Search Tree (GiST) described in [3] and in Chapter 50 of PostgreSQL documentation ([11]), and Generalized Inverted Index (GIN), described in Chapter 51 of [11]. PostgreSQL's contrib module contains sample implementation of an R-tree done using GiST. Implementing GiST index requires providing library with six functions; GIN requires implementing four functions.

GiST allows for creating indexes similar in structure to B-Tree indexes. But it can be used to implement custom comparison functions or additional conditions that should be met by a row to be chosen by a query ([11]):

This extensibility should not be confused with the extensibility of the other standard search trees in terms of the data they can handle. For example, PostgreSQL supports extensible B-trees and hash indexes. That means that you can use PostgreSQL to build a B-tree or hash over any data type you want. But B-trees only support range predicates ($<$, $=$, $>$), and hash indexes only support equality queries.

So if you index, say, an image collection with a PostgreSQL B-tree, you can only issue queries such as "is imagex equal to imagey", "is imagex less than imagey" and "is imagex greater than imagey"? Depending on how you define "equals", "less than" and "greater than" in this context, this could be useful. However, by using a GiST based index, you could create ways to ask domain-specific questions, perhaps "find all images of horses" or "find all over-exposed images".

137

On the other hand, GIN stores keys together with lists of rows where the key value appears, so it allows for speeding up queries searching for rows with particular value of column. It also can allow for building statistics of data stored in a database, as it stores list of all values of column.

A code can be built together with a domain specialist. It is next step on a way proposed by Codd ([4]). He proposed to split responsibility for storing and operating on a data between a database and a program. Now we can let database experts to build storage engine, and domain experts to build functions operating on data and put those functions into database, to make queries more efficient ([11]):

> One advantage of GIN is that it allows the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert. This is much the same advantage as using GiST.

As shown in [15] and [12], relational databases can use many indexes concurrently to optimize query. These indexes can be of different types (B-trees, hashes, own types); also they can involve one or many columns. Moreover, one column can be taken into consideration in many indexes. DBMS cas then build partial results using those indexes and then use so called bitmap index to join them to receive a final result.

But having many indexes can worsen performance during changing data, as all of them must be updated. Also, they require space. But it is often good trade-off to sacrify some space for efficient data operation.

DBMS is free to choose a way of performing query (inter alia choose one of indexes) because SQL is a declarative language, To do this, it creates many equivalent queries using the relational algebra, and then checks what will be the most efficient one. Entire process of optimizing queries is described in [12].

To be able to choose the best plan, DBMS needs information about data and system. The database either computes different performance estimators, or allows for administrator to set them. For example, PostgreSQL allows for tuning a database to hardware it is located on by using different configuration variables, described in Chapter 17 of [11]. Some of them (enable_bitmapscan, enable_hashjoin, enable_nestloop, enable_seqscan) allow turning on and off usage of particular types of indexes (hash index, B-Tree index etc.). Other allow for setting cost of different operations in the system (seq_page_cost and random_page_cost allow for setting cost of reading one disk page id different modes, cpu_tuple_cost is cost of processing one row, cpu_operator_cost is cost of applying one operator, effective_cache_size is size of memory used by operating system to cache disk pages).

As it can be seen, system administrator can tune database to be as optimal as possible.

5. Summary

Temporal data must be stored in special structures, which are complicated in description and implementation. These structures, however they are enriched b-trees, present less performance during operations on temporal data, and require more disk space to store it. Data structures like trees offer only one way of performing operations on data stored in them.

On the other hand database optimization allows changes of "query plans" (ways of performing operation on data in database), based on available indexes, number of rows, even values of columns. Thanks to that DBMS can choose optimal plan, which helps with solving problems with efficient operations on temporal data.

So it is preferable to use relational databases instead of own data structures to store temporal data.

References

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

[2] Frank Adelstein. Live forensics: diagnosing your system without killing it first. *Communications of ACM*, 49(2):63–66, 2006.

[3] Oleg Bartunov and Teodor Sigaev. Introduction to gist. Technical report, Moscow, Russia. http://www.sai.msu.su/ megera/postgres/gist/doc/intro.shtml.

[4] E. F. Codd. A relational model of data for large shared data banks. *Communications of ACM*, 13(6):377–387, 1970.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Wprowadzenie do algorytmów*. Wydawnictwa Naukowo–Techniczne, Warszawa, wydanie trzecie edition, 2000.

[6] Edward Cutrell, Susan T. Dumais, and Jaime Teevan. Searching to eliminate personal information management. *Communications of ACM*, 49(1):58–64, 2006.

[7] Chris Date. *Database In Depth. Relational Theory for Practitioners*. O Reilly Media Inc., Sebastopol, CA, USA, 2005.

[8] Michael J. Franklin. Concurrency control and recovery. In Tucker [19]. Published in Cooperation with ACM, The Association for Computing Machinery.

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

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

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

[12] Yannis E. Ioannidis. Query optimization. In Tucker [19]. Published in Cooperation with ACM, The Association for Computing Machinery.

[13] Jim Melton and Andrew Eisenberg. Sql multimedia and application packages (sql/mm). *SIGMOD Rec.*, 30(4):97–102, 2001.

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

[15] Betty Salzberg and Donghui Zhang. Access methods. In Tucker [19]. Published in Cooperation with ACM, The Association for Computing Machinery.

[16] Dennis Shasha and Philippe Bonnet. Tuning database design for high performance. In Tucker [19]. Published in Cooperation with ACM, The Association for Computing Machinery.

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

[18] Alexander Thomasian. Transaction processing. In Tucker [19]. Published in Cooperation with ACM, The Association for Computing Machinery.

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

PROBLEMY ZE SKŁADOWANIEM DANYCH TEMPORALNYCH

Streszczenie: Artykuł przedstawia problemy implementacyjne związane z przechowywaniem danych temporalnych. Opisuje struktury drzewiaste, które mogą być użyte do przechowywania temporalnych danych oraz problemy z nimi związane. Przedstawia sposób użycia relacyjnych baz danych do przechowywania danych temporalnych, jakie wiążą się z tym problemy, jak można wykorzystać możliwości oferowane przez bazy danych do ułatwienia implemenacji. Opisuje problemy z wydajnością oraz sposób ich rozwiązania w relacyjnych bazach danych.

Słowa kluczowe: struktury danych, drzewa, dane temporalne, bazy danych, relacyjne bazy danych