# Temporal Database Architecture Enhancements

Michal Kvet
University of Zilina
Zilina, Slovakia
Michal.Kvet@fri.uniza.sk

Emil Kršák, Karol Matiaško
University of Zilina
Zilina, Slovakia
Karol.Matiasko@fri.uniza.sk

*Abstract*—**Database systems used in the past were characterized by storing only current valid states, which is not, however, optimal for intelligent systems and applications. Temporal paradigm allows us to delimit the object state with the time validity regarding the modeled granularity. Temporal systems are based on the extension of conventional approaches, which do not provide powerful solutions. Database server architecture is described with emphasis on the optimization options to improve and shorten the data retrieval process. In this paper, we attach significance to the migrated row, which forces the system to load multiple data blocks from the database into the memory. The solution, based on the evolution steps, is implemented using the mapping module inside the memory of the database instance.**

## I. Introduction

A core part of almost any current application and information system comprising efficiency and intelligence is just data. A number of data stored in the systems are still rising over the decades. Data are commonly stored in the database, which provides effective techniques for data handling, manipulation, and processing. However, the most significant data access performance parameter is just data retrieval process. In the past, disc storage and complex hardware were really expensive, therefore the number of data to be stored was strictly limited. It meant, that only current valid data were stored. Thus, any change in the data caused executing direct *Update* statement and historical data were replaced with newer ones. Another aspect was just impossibility for handling future valid data, there was no space for future states recording. Later and even now, the hardware possibilities are wide, prices are relatively low and data management possibilities are easily manageable, as well. Thus, the data amount is extremely rising creating significant press for hardware and software techniques. Nowadays, conventional database approach is being continuously replaced with the sensorial data management and each state is time delimited. Therefore, data tuple is bordered with the time definition on an object or even attribute granularity. In this paper, we propose extension module for the data retrieval proposing more effective and reliable solution with reference to actual solutions. Thanks to that, retrieval process can be shortened. The ideal solution is to save all the data in the memory with effective and fast access, however, such solution is still mostly in a theoretical way because it would require huge memory modules and continuous necessity to extend them, whereas data amount is rising. Another aspect is just reliability, accessibility, but mostly security. It must be covered with the metadata forming characteristics on memory data to provide the possibility to reconstruct data after any failure. Cloud can only partially solve the problem.

Current relational database systems are characterized by the instance forming memory structures and database. Data are permanently stored in the database and manipulated in the memory. If there is any change (*Insert, Update* or *Delete*), particular metadata (original and new image of the tuple) is stored in the log file of the physical storage. Thanks to that, after any failure, it is possible to reconstruct states and therefore the entire database. Database memory structures are determined to provide fast access. Naturally, memory capacity is commonly smaller than the whole database, thus it is necessary to load data into memory, as well as to write changed blocks back into the database. Whereas data are not always accessible in the memory, it is necessary to provide a fast and reliable approach to locating data in the database – on the physical file system. The easiest way is to scan the whole database with an attempt to find particular data. Such approach is, however, very naive, because system formed on this assumption would be inappropriate, too time-consuming with poor performance, effectivity, and reliability.

Therefore, index access layer has been proposed. This paper deals with the extension of the index structure to remove the impact of the fragmentation and data block relocation. The aim is to lower a number of I/O operations, to ensure, that required data are actual, in the data block, we assume.

## II. Conventional and Temporal Database

Most data stored in the database during the last years were characterized by storing only current valid states for the application domain. Tuples, properties, and characteristics, however, evolve over the time and if there was a change, non-actual data were replaced. However, everything has its evolution in time, its history and future, as well. Therefore, changes monitoring can be progressive allowing data evolution monitoring, creating prognoses, better decision making, etc. The significant aspect is just the security. When all data spectrum is stored, there is no problem to reconstruct any data image during defined timepoint or interval. The aim of the temporal definition is to provide complex information about the object state during the defined period with emphasis on changes. Thus, it was necessary to extend conventional paradigm, respectively to create a new one. With such definition, the data amount significantly rises. On the other hand, process optimization, evolution management and possibilities to predict problems, management and reactions based on real historical data are really powerful [10], [12]. Thanks to that, it reduces application domain costs and increases decision efficiency. The second aspect is, however, performance. The fact, that we have data over the whole time

spectrum does not automatically mean, that they are usable on a daily basis, that they are quickly and reliably accessible. Temporal paradigm is characterized by the extension of the object identifier. Primary key, as the unique identifier of the tuple (row), does not contain only object definition, but the time validity, as well. Fig. 1 shows the principles of the temporal model extension. Uni-temporal table extends primary key of the conventional approach with one pair of the time attributes expressing validity interval, which can be modeled with various time characteristics – closed-closed or closed-open time interval. A special solution is based on only one-time attribute – in that case, each newer state delimits the validity of previous one. The bi-temporal approach uses two-time spectra – validity and database time reflection. In general, the multi-temporal solution can be used with a various number of time spectra – validity, time locality, database time, reliable time, etc. The proposed solution is robust, can manage states anytime – historical, current and future valid data, as well. Defined model is object-oriented – validity defines the whole image of the data tuple.

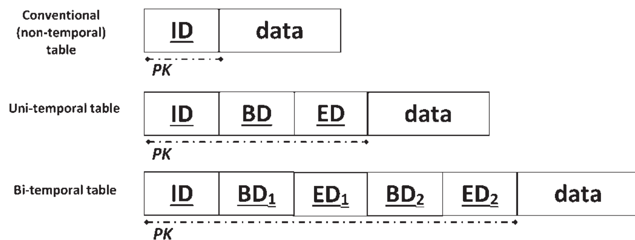


Fig. 1. Conventional and object level temporal model

In 2016, more precise granularity model has been proposed (Fig. 2). In that case, validity defines the attribute value, not the whole object. The main advantage is significant size demands reduction with no duplicate values. Systems and applications communicate either with a layer with actual states or with a temporal layer providing an image at the time. It is not possible to access in actual values directly because of the security reasons.

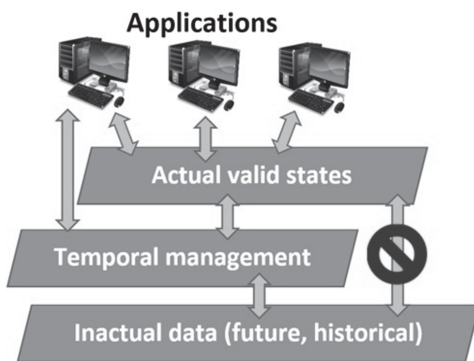


Fig. 2. Column-level temporal architecture [15]

The core part is the temporal manager described in [14].

Any temporal solution, however, provides a significant extension of the data to be processed and stored. Therefore, it is inevitable to store and retrieve them effectively.

## III. Physical database architecture – technical background

Database server consists of two entities – the *instance* and the *database*. The instance is formed by the memory structures and processes, the database is defined by the physical files on the disc storage. Although they are physically separated, they must be interconnected to be usable, otherwise, they are unworthy of the data access from the other systems. In principle, the instance is created sooner, afterward, the connection is created using the mounting process during startup. If the process of the creating connection fails, the database cannot be opened and data are not available. There are several types of architectures, like single instance, Real Application Clusters (RAC), Streams, Data Guard or Cloud in a distributed environment - approach principles are the same [2], [16]. Client-side defines user process, which requires server listener to create a specific process on the server side for communication. Server-side can process requests from the client side and apply them to the database using background processes. Fig. 3 shows the components for the single instance database.

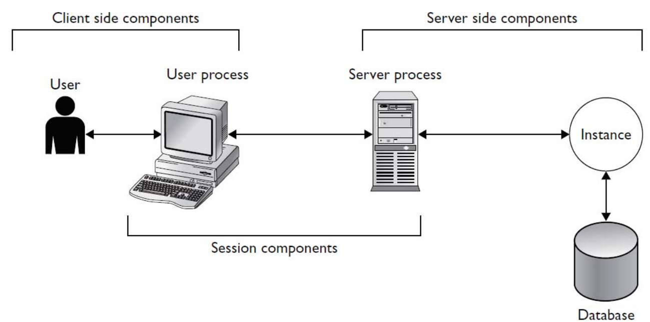


Fig. 3. Communication principles in single instance architecture [2]

The performance aspect of the system covers also instance memory structures. Advanced techniques in the newer versions of the database systems can provide self-administering solutions of the memory structures by relocating memory on demands. Each database instance must consist of three database structures – *Database buffer cache*, *Log buffer* (small, a short-term staging area for change vectors before they are written to the redo log on the disc. It consists of change vectors – modification applied to the data. Redo log ensures, that data will never be lost. Whenever any data portion is changed, particular change vector is created and written to the log. Thus, by applying change vectors to any backup, it is possible to reconstruct database) and *Shared pool*. For the temporal data retrieval, it is an important to box, whereas data must be always loaded into memory before the processing. Optional structures are *Large pool*, *Java pool* and *Streams pool*, which are not, however, part of our optimization.

*Database buffer cache* is core part of the processing – work area for the SQL statement execution. When dealing with data, connected sessions do not access and change direct data on the discs. Particular data blocks are first copied into this memory structure. Moreover, after the processing, changed data blocks are not immediately copied back into the

database files, but remain in the memory, if possible. Thus, the size of such cache is significant. From the point of view of the performance, the ideal solution would be, if all data could be accessible directly from the memory. However, as described sooner, it would require really significant demands on the hardware. If the data amount is rising, memory demands would be sooner or later unsustainable. Moreover, the process of loading during the startup can last too much time. Sequential writing before a shutdown would be a problem, as well. And we are not talking about the recovery process in case of a system crash. Data in the files, as well as memory structures, are formatted into fixed-sized blocks. Usually, size of the block is 8kB or its multiplies (performance impacts and dependencies on the block size definition can be found in [14], [15]).Thus, ideally, but the realistic option for the size of the buffer cache is determined by storing often accessed data [16], [18]. As we will highlight in this paper, core part is formed by the indexes and their improved access and performance solutions. Background processes load and unload data blocks into and from the memory based on demands. It is necessary to distinguish between *clean* and *dirty* blocks. Clean buffer block can be rewritten immediately, if necessary (such block has been used only for data retrieval). In comparison with a *dirty* block, it must be copied into the database before the replacing with another block. Thus, if there is a requirement to obtain new block for data copy, current database management systems look for the clean block sooner. In our opinion, such solution is not optimal. Therefore, in this paper, we extend optimizer manager by newer approach technique.

## IV. OWN SOLUTION – BUFFER CACHE BLOCK MANAGEMENT

Our solution is based on separating block types – indexes are always on the memory. State management is a bit more complicated. When dealing with temporal management during a big time frame with many updates, it is not possible, or even used to store and evaluate all the data on the daily basis. The validity of the state and time definition delimits the priority of the tuple to be placed in the memory. The volatility of the temporal object means, that historical images may lose meaning and value over the time, therefore they are consecutively removed from the evaluation. Such characteristics and properties are also temporal and can evolve over the time.

### *A. Shared pool – existing principles*

The shared pool is a complex structure, which consists of these components:

- *library cache,*
- *data dictionary cache,*
- *PL/SQL area,*
- *result cache.*

In most database systems (including Oracle database system used in the evaluation), size of the structure is dynamic, limited by initial parameters. *Library cache* is a memory for storing recently executed code in its parse form, which is directly executable. It contains the technique to obtain requires data – access parameters, indexes, steps to be done, the order of table joins, etc. [1], [3], [4].

### *B. Shared pool structure – our own approach*

Principles of the *Library cache* management is too strict. Although two approaches exist now – *exact* and *similar* for comparing statements to form them into the structure server can evaluate them with existing parsed code. Such approach, is, however, not enough for the temporal definition. It does not cover time execution spectrum with emphasis on the frequency of changes, a number of changes, reliability, stability, and precision. Our own solution removes the time sphere from the SQL statement and stores parsed version with the universal time delimitation. It means, that it is time length independent with the assumption, that index structures, and optimized access is used. Thanks to that, if different time spectrum is used, but referencing the same object or object group, already stored parsed version can be used. Principles are shown in the Fig. 4. The first part is based on parsed version definition – time spectrum is removed followed by the parsing process. The result is stored in the library cache. When another SQL statement is defined, first of all, time spectrum is evaluated based on the frequency of changes, number of data in the result set. If there is a recommendation for optimized data access (index structure, etc.), database manager looks for prepared parsed version. If it exists, it will be used, otherwise, the first part is used with the aim to create a new parsed version.

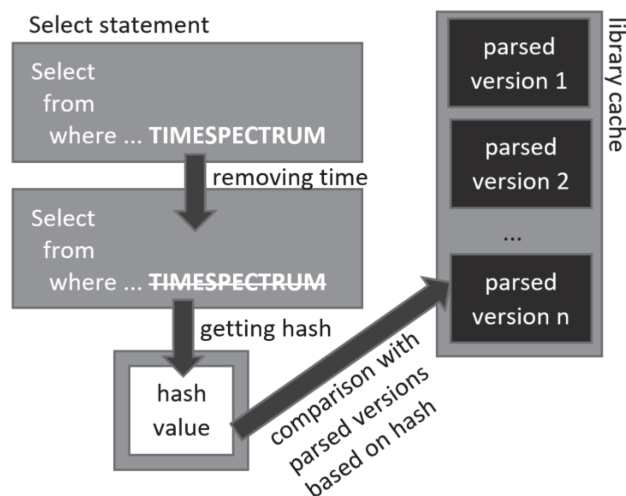


Fig. 4. Statement evaluation and hash value comparison

As already mentioned, size of the *library cache* is not unlimited, thus it is necessary to evaluate parsed versions, which will be removed and replaced with new parsed images. Database systems use *Least Recently Used (LRU)* approach. For each parsed code, time of the last usage is stored, thus it is easy to find the victim code to be removed. Such solution is, however, also inappropriate, for temporality modeling. First of all, the quality criterion should be highlighted – the number of times, the particular statement has been evaluated to be covered by an existing parsed version with reflection to the complete number. The second aspect is just the assumption for soon usage and time, the parsed version is in the memory. We also focus on the complexity of the evaluation process – time and size demand to obtain parsed versions. In the temporal

environment, most *Select* statements are really complex finding dependencies and correlations between object states over the time. Therefore, versions to be removed from the *library cache* are pre-calculated based on mentioned aspects. Reliability of the parsed version is compared to time spectrum definition, in our solution. Before the parsed object removal itself, it is worth to evaluate a number of times, the parsed code has been used, complexity and frequency of such usage. Thanks to that, it can be reloaded based on assumptions or defined rules (e.g. some statistical and report evaluations are always launched at the end of the week or month, thus it is useful to have parsed version already in the memory). When the parsed code is removed from the memory, it can be optionally stored in the disc space. In our solution, we add specific database tablespace storing compressed versions of the parsed code. When unloading into memory, it is de-compressed to original form. However, we must mention also the timeliness and worthiness of the parsed version. Whereas the data specification, number, and frequency of changes is continuously changing and data amount is rising, it is necessary to evaluate, whether the parsed version is still actual and provides an effective tool for data accessing and consecutive retrieval. Therefore, we extended our solution and provide several parameters, which can be set. It can be either defined for the whole database, object or even attribute granularity can be used. It is delimited by the number of newer states, elapsed time from the point of parsed version definition or a number of times, particular parsed version has been transferred to the disc and back (Fig. 5).

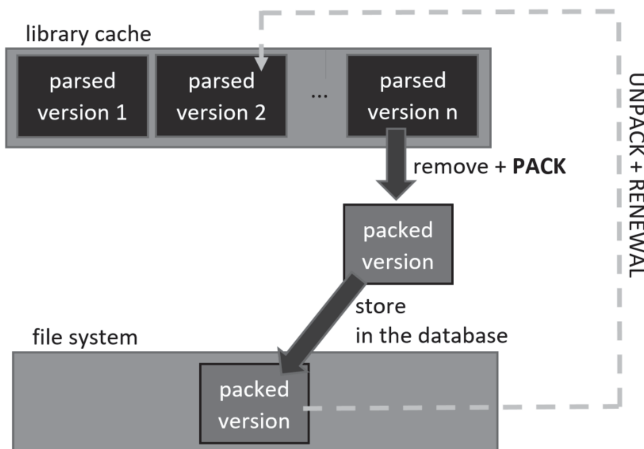


Fig. 5. Packing, unpacking, and renewing

In this approach, it is strictly essential to keep object statistics actual.

*C. Data dictionary cache – existing and own solution*

*Data dictionary cache* is often defined as a row cache. It deals with object definitions, table descriptions, indexes and many other metadata. Keeping these definitions in the memory provides direct and immediate access when evaluation and parsing should be done.

Our implementation extends such structure with *recent statistical information about object definition and characteristics – a number of changes, the frequency of changes*. It is part of our extended statistic tool and recalculated automatically or on demand. These statistics are with regards to the time sphere evaluated during the SQL statement.

## V. BACKGROUND PROCESSES

Database instance consists of memory structures described in the previous sections, but it is also formed by the processes accessing, managing database, as well as processing and managing communication with the users, to be responsive to their requests. Each of the processes has its own significance, meaning, and history. For purpose of this paper, we will mostly highlight performance impacts of such processes and our own methods for improvement.

*System Monitor (SMON)* is responsible for the mounting and opening database. It manages the process of the startup and interconnects the instance and database during the mounting.

*Process Monitor (PMON)* is a process, which manages user and server process by solving transaction problems if the connection is lost. In the standard conventional environment, *PMON* is not proactive. Simply, it does not detect, whether the connection is still active or not. In temporal definition, such paradigm is still valid. However, based on the experiments and performance analysis, we came to the conclusion, that temporal environment requires more strict solution [5], [6], [7], [15]. Temporal data can be characterized by strong data input stream and must deal with the conflicts, which lies in the data lost, incorrect data or communication failure. In that case, it is necessary to locate problems and try to recreate data communication channel. It is done by dropping communication between the existing server and a user process. Fig. 6 shows the difference between existing conventional approach and our designed temporal definition. In that case, server cyclically sends the signal through which the connection is controlled. Signal cycle is time dynamic and depends on the frequency of the data flow, based on historical data evaluation. Compare the solution with the Fig. 3 showing original conventional definition.

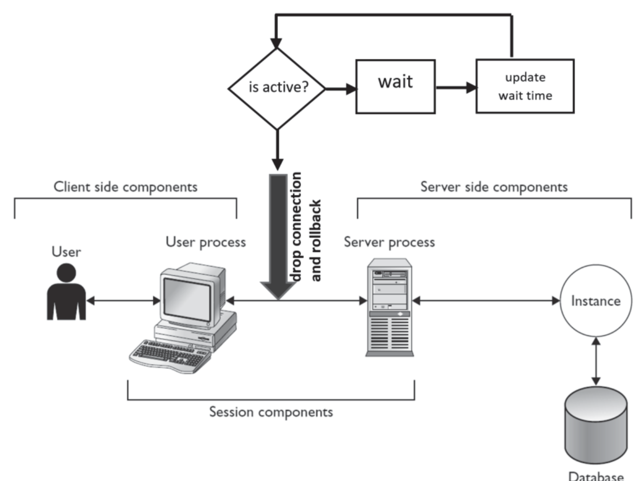


Fig. 6. Temporal transaction session management extension

*Database Writer (DBWn)* is a process, by which data are transferred from the memory into the physical database files. Notice, that sessions cannot directly access the physical database, nor change some data portions inside it. *Database Writer* writes data from the memory to database in four circumstances – no free space in the buffer cache, too many dirty buffers, three-second timeout, and when there is a checkpoint. Database systems have their own approach for selecting buffers to be written and freed.

Our defined approach uses the autonomous decision-making process as a support tool and extends the rules ensuring that index structures are always available in memory.

*Log Writer (LGWR)* is a process, which transfers the content of the log buffer into disc log files. Thus, it ensures no data can be lost. In the temporal environment, we have encountered a problem of speed and efficiency of the existing solution. If the input data stream is strong, processing bottleneck is just the log buffer and its background process manager. Therefore, our approach detaches transaction and log data by multiplying the a number of log segments in the memory-forming groups. There can be several *Log Writer* processes, each of them has its own log buffer data. A new transaction is assigned to one *Log process* using *Log manager*, thereby minimizing the amount of data to be transferred at the end of the transaction. To get desirable performance improvements, it is necessary to assign separate disc space for each *Log group* handled by the separate controller, otherwise, the problem would just be in another layer – moving data from logs to the file system (Fig. 7).

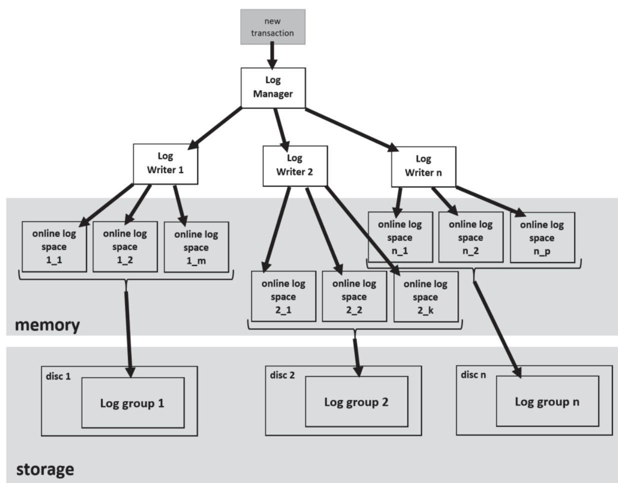


Fig. 7. Data logging management

*Checkpoint process (CKPT)* is a process that moves all data from the *Buffer cache* to the database. Its aim is to free all blocks to create consistent database image in case of a shutdown. In the past, it has been done regularly to minimize time to reconstruct memory image after failure.

*Manageability Monitor (MMON)* is a core performance process of the temporality. Its aim is to manage statistics about the objects, to ensure, that they are correct and represent actual states. It is strongly important to protect them and to guarantee automatic refresh. Conventional and many temporal databases use maintenance windows, usually planned at night, during which the workload is lower. In our case, we had to think about an extended solution. First of all, temporal databases usually have a uniform workload, respectively there is no significantly weaker input data stream, during which statistics can be globally refreshed.

Therefore, we propose memory extension consisting of the linear linked list, in which objects are sorted based on priority, which is modeled by the number of changes in comparison with actual states. For each object table, a number of rows are stored in the statistics. The difference between stored and current value defines the priority of the record in the linked list. An object with the higher priority is recalculated (refresh statistics) as the first. It is done periodically in the cycle, however processing of the input data stream monitored and resources are dynamically relocated to ensure, that input data will be processed correctly with no significant delay.

*Memory manager (MMAN)* is an autonomous process that reallocates memory between individual structures. Database system Oracle goes even further and allows complete automatic management (from 11g version). Although results can be stored in the result cache memory structure, it is not satisfactory, therefore particular data should be part of the buffer cache. Fig. 8 shows the structure of our proposed solution. The buffer cache is divided into three parts. The first part stores index structure data, the second deals with the temporal management tables. The third is just the original part of the buffer cache consisting of clean and dirty buffers. Only data from the third part can be relocated to the database and replaced with the other data images. Thus, the status of the buffer block can be divided in our solution into three parts:

- *clean* (data block is not changed in comparison with database image and can be removed from the memory immediately),
- *dirty* (data block is changed, therefore before removal, it must be copied into the database),
- *protected* (data block is locked and cannot be moved away from the memory).

The protected block is always present in the memory from the instance startup until the shutdown of the system. Regardless the data are changed in the protected block, it cannot be moved away from the database, although it convenient to hold such blocks clean, therefore specific *Database Writer (DWprot)* has been introduced by us, which handles only protected blocks. Principles for data transferring are the same as the standard *Database Writer*, however, it only copies data, but particular objects must be also present in the memory after the copying process.

## VI. INDEX STRUCTURES

A database index is an optional structure, that can rapidly improve the performance of the data retrieval by accessing particular data tuples defined by the key path. It reduces total processing costs, as well as improves the speed of the data retrieval. In comparison with standard database approach, where data are sequentially scanned, index access is based on locating data using pointers to the physical database. Thus, the

main advantage is a quick data location possibility without the necessity to search every database row, which can be even located in multiple data files in many discs. The index is created for the table based on attributes, which are covered by the index. Theoretically, it is possible to cover all table attributes in the index, however, it depends on the memory structure size associated with the buffer cache. Thus, it is recommended to cover only temporal management table, which is performance significant, whereas data should be time-sorted.

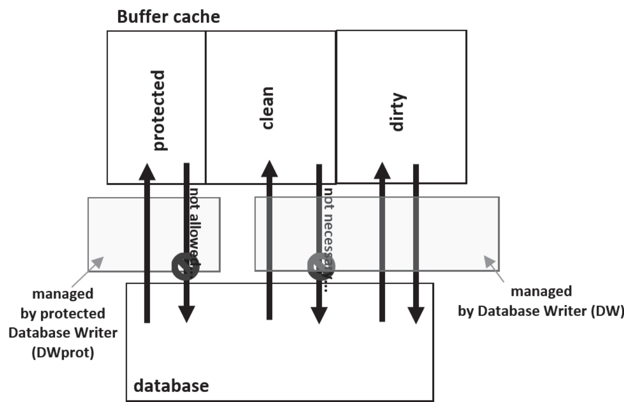


Fig. 8. Buffer cache management

As already mentioned, the index consists of direct attribute values and pointers to the data file locating the whole row. Moreover, there is a possibility not to index attribute values directly, but they can be preprocessed using functions, which must be, naturally, deterministic.

Current database systems covering also temporal environment use mostly B+tree schema index (Fig. 9), which maintains the efficiency despite frequent changes of records. Other solutions are *hash indexes*, *partitioning indexes,* and *bitmap indexes*. The special category covers *domain indexes* and *functional indexes.* Performance comparison and characteristics can be found in [8], [9], [19].

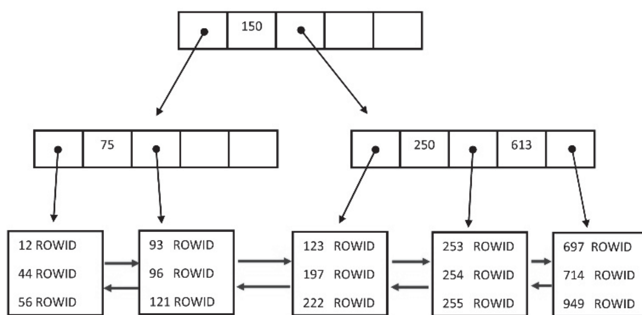


Fig. 9. B+tree index

B+tree index consists of a balanced tree in which each path from the root to the leaf has the same length. In this structure, we distinguish three types of nodes - *root*, *internal node* and *leaf node*. B+tree extends the concept of B-tree by chaining nodes at the leaf level, which allows faster data sorting. DBS Oracle (used for experiments) defines the accessibility methods of the two-way linked list, which makes it possible to sort ascending and descending, too [9].

The leaf level of the B+tree index structure consists of the *ROWID*, which characterizes the pointer to the database, where the row physically resides.

### *A. ROWID*

ROWID is a pseudo column returning address of the row. In the past, it was formed as 8 bytes, nowadays; it has been extended and consists of the 10 bytes [16]. It uniquely identifies a row in the database and deals with these categories:

- *data object number* (1-32 bits),
- *the data file*, in which the row resides (33-44 bits),
- *the data block* in the file, in which the row is located (45-64 bits) and
- *the position of the row* in the data block (65-80bits).

It is the fasted way to access a single data row. On the other hand, there is still space to optimize locating process and to speed up performance. The temporal environment is characterized by a various data stream, which is, however, usually, powerful and consists of wide data range evolving over the time. Data reliability and precision is also significant, which means, that also attribute data types can be changed based on data structure and precision. Thus, the row size is not strictly defined mostly regarding the size of the whole row. It is commonly extended over the time. If the row after the change cannot be placed into the same position inside the data file, another free data block must be found and such row is placed into newer position. And that´s the problem. Index structure locates data in the row, however, such data are not placed there. Inside the particular block, which must be loaded into the memory, is only a pointer to another block, where data can be located, or it can even consist of only another pointer, as well. Thus, performance is degraded, because index structure does not point to the direct data row. Problem is limited by the fact, that when data are updated, there is no pointer from the database to the memory, thus *ROWID* cannot be updated to reflect newer precise data location and position. What does it mean? The answer is easy – it is necessary to rebuild index periodically [11], [17]. Thus, if the performance is degraded, it is convenient to recreate or rebuild the index. In new database releases, it can be done online – the original index is used till the new updated index is created. Afterwards, the original one is dropped. On the first side, it can be considered as the adequate solution, however, that´s not true for the temporal environment. Data are changed rapidly and to ensure the performance of the index, it would be necessary to rebuild them continuously. It can be even said, that when a newer version of the index is created, it is not even actual at that moment because, during the creation (which can last much time), data characteristics and amount are changed. The solution must be, therefore, significantly more complex and robust.

## VII. OWN INDEX ACCESS APPROACH

The interconnection between index and database is just *ROWID*, which does not guarantee the direct access to the data. The migrated row in the temporal environment, where the block holds the only pointer to another block, is

performance limitation and can strongly degrade the whole performance. Our proposed solution is based on *ROWID* updates and techniques to guarantee the right and direct access without migrated row management limitation. To get the complex image, it is useful to mention also the evolution of our solution, to create a complex image. Problem is shown in the Fig. 10. Index *ROWID* points to the row located in the block *B1*, however, it is too full to cover the image of the row after the update. Therefore, migrated row is created and a particular object located in the block *B2*. To get the data, block *B1* is loaded into the memory at first, followed by loading block *B2*. Thus, we needed two disc operations to get data. However, in general, the number of disc operations can be drastically higher.

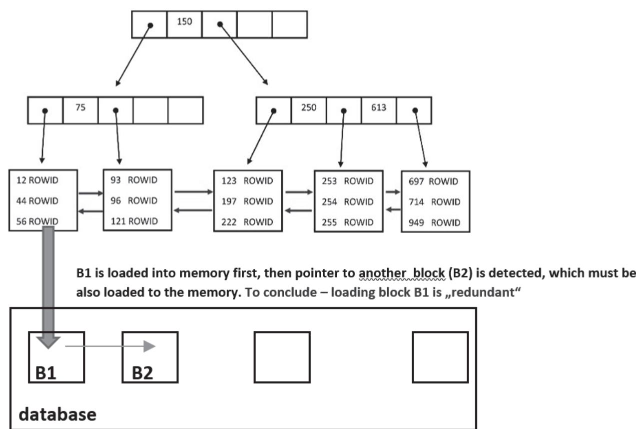


Fig. 10. Migrated row

The first introduced solution was based on minimizing the number of migrated row hops – for each operation, no more than 2 operations would be required. It is ensured by storing the address of the block, where the *ROWID* points. During the evaluation and loading particular blocks, the final block is interconnected to the original block, which is also shown in the Fig. 11 – only block *B1* and *B4* must be loaded after the processing (in comparison with original solution forcing the system to load blocks *B1, B2, B3,* and *B4*. This is done only during the data retrieval process, whereas the linking is only one-directional. Highlight the situation, to get only one row, it can occur many times, that it would even be necessary to load tens of the block into the memory. Moreover, it requires to find clean blocks in the buffer cache or even to make the decision for selecting that one, which will be replaced with a block to be loaded. The specific situation can occur, if block, which actually holds particular object image, would be replaced. The solution is performance better, however, does not eliminate migrated row completely.

Later, our aim was to shift the solution to the upper level, to ensure, that only one block will be loaded, to ensure, that required data are in the block, where *ROWID* locates. And that is a really complicated process and can be divided into two separate stages.

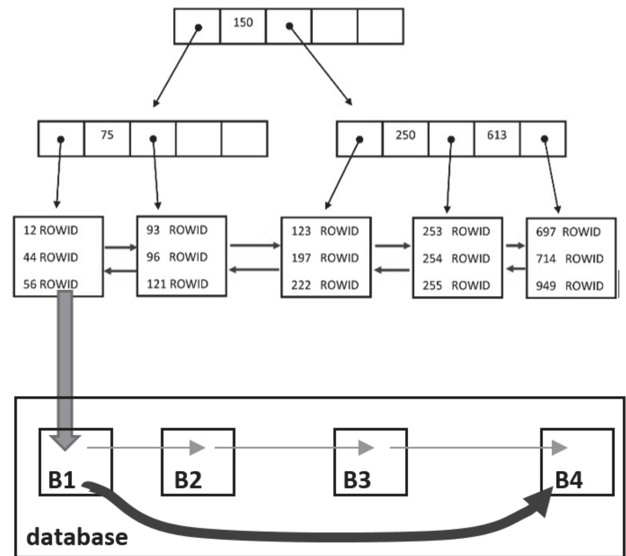


Fig. 11. Own solution - interconnection

The first one is easier, however, less efficient. It deals only with one index structure. During the data manipulation (*Insert*, *Update* and *Select* statement), if the row is located using the index, a new temporary pointer to the leaf node holding particular *ROWID* is created. If the row is transferred to another block, using such pointer, new *ROWID* is calculated and stored. Fig. 12 shows the solution, which is not, however, robust. In this case, several pointers to the same row can locate different data blocks. Some of them are direct, some of them use migrated row technique, whereas *ROWID* pointer is changed only for one index. That is the consequence of the fact, that system uses only one directional linked list. As we can see in Fig. 12, index *IND2 ROWID* locator is updated, thus it points to the block *B4* – there is no migrated row. However, it is not updated to the other indexes, they locate the original position and migrated row exists for them.

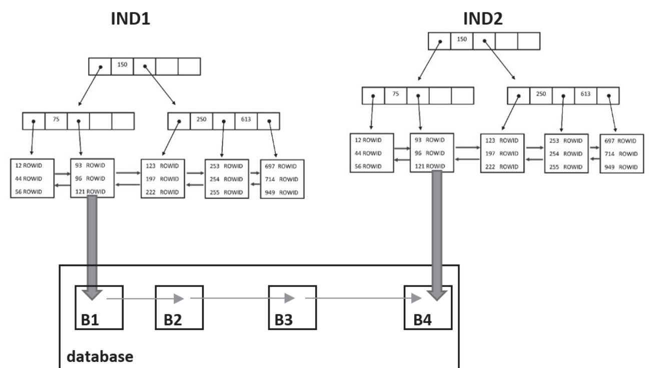


Fig. 12. Own solution – interconnection and multiple indexes

We, therefore, propose other two solutions, which are performance compared. The first is based on two directional linked list, thus the database block additional structure points to the memory based on particular ROWIDs (Fig. 13). Naturally, it is created dynamically, if the data are accessed. Whereas index is always loaded in the memory, if the database

server is active, pointers from the database are static. After the system reloading, database pointers are automatically deactivated during the mounting and opening process of the database and instance. Fig. 13 shows the solution. Limitation of the solution is just the process of the deactivation and building process of the bi-directional linked list. It is necessary to mention, that there can be several indexes, which can point to the same database block, thus the reflection from the database would require dynamic approach using a dynamic array or linear linked list. Therefore, the second proposed solution is based on an extension of the memory structure using pointer layer. There is still deactivation process during the instance startup, however, there is only one pointer from the physical database. Individual pointers to the indexes are located in the memory. Fig. 14 shows the implementation principles using mapper *module*., which interconnects all ROWID values from the indexes with the physical data location provided from the database. There is no migrated row problem, at all.

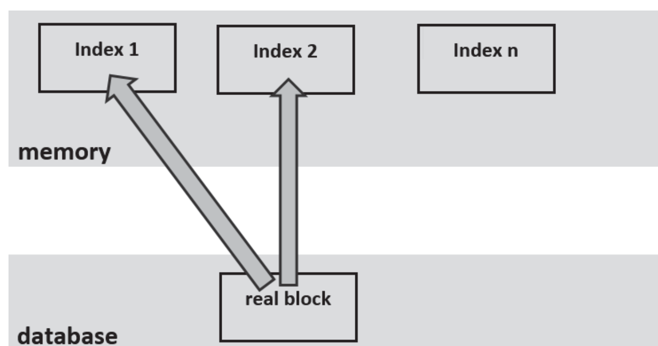


Fig. 13. Mapping database block into memory – database pointer position

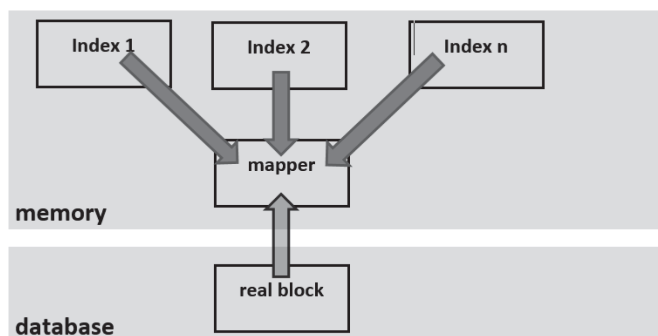


Fig. 14. Mapping database block into memory – memory pointer position - mapper

## VIII. Performance

Experiment results were provided using Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production; PL/SQL Release 11.2.0.1.0 – Production. Parameters of the used computer are:

- Processor: Intel Xeon E5620; 2,4GHz (8 cores),
- Operation memory: 16GB (8 modules, DDR 1333MHz)
- HDD: 500GB.

Experiment characteristics are based on real environment consisting of *1000* sensors producing data *ten times for one second*. If the difference between consecutive values is lower than *1%*, such values are not stored in the database and original value is considered as unchanged. Thus, based on our environment, the average amount of new values is approximately *1000* per second. Amount of data after one hour is *3 600 000*.

The performance experiments are based on the attribute-oriented temporal solution. Comparison with other temporal architectures (object-oriented, group, synchronization) can be found in [13], [14], [15].

Five temporal models proposed in this paper have been compared. The reference model is an attribute-oriented temporal architecture (*AOTA*). The second approach (*M2*) is based on Fig. 11 – migrated row is partially eliminated to ensure, that original block is directly connected to the last one, where the data tuple actually resides. There is an assumption in the model *M2*, that only one index is defined and it is always suitable. Such definition is, however, not possible to be implemented in the real environment, therefore it is extended in model *M3* using several indexes (Fig. 12). Whereas there is no pointer to the other indexes eliminating migrated row, performance is worse.

Model *M4* principles are shown in Fig. 13. There is a mapping pointer structure, located in the database. After the system reloading or startup, the structure is deactivated and consecutively built again. Performance is evaluated after the process of database pointer definition is completely done. One block, in that solution, points to the whole set of the indexes, which reference such row. The last model (M5) is a final step of our optimization is used *mapper* structure located in the memory. Thanks to that, access time is really low, whereas mapper itself is located in the memory. Moreover, there is no necessity to extend the database block rapidly, whereas it consists of only one locator to the *mapper* structure.

Performance of the whole system is on the Table I highlighting *costs, CPU* and *processing time* of the data retrieval.

TABLE I. Experiment results

| | AOTA | M2 | M3 | M4 | M5 |
|---|---|---|---|---|---|
| **Costs** | 3447 | 3012 | 3127 | 2940 | 2781 |
| **CPU [%]** | 11 | 10 | 10 | 9 | 8 |
| **Processing time [s]** | 52,3 | 46,5 | 48,9 | 44,1 | 42,6 |

As we can see, the best performance provides model *M5*. Referencing the standard attribute oriented model (*AOTA*), *M2* provides improvements 12,62%, model *M3* lowers costs using 9,28%, the model *M4* benefit is 14,70% and the last model *M5* reflects improvements 19,32% for costs. Fig. 15 shows the results for the costs in the form of a graph.

Fig. 14. Performance Costs

When dealing with the CPU consumption, model *M2* and *M3* requires one percent of the system resources less, model *M4* improvement is two percent and model *M5* requires 8 percent of the system resources in total.

The significant performance aspect is just the processing time expressed in seconds (tab. 1). AOTA requires more than 50 seconds for the processing, which is eliminated by the migrated row problem separation in model M5 up to 42,6s, which represents the improvement on 18,54% (reference model is AOTA). Model M2 improves solution using 11,09%, model M3 reflects 6,50% and model M4 reached the improvement on 15,68%. The graphical representation is in Fig. 16.

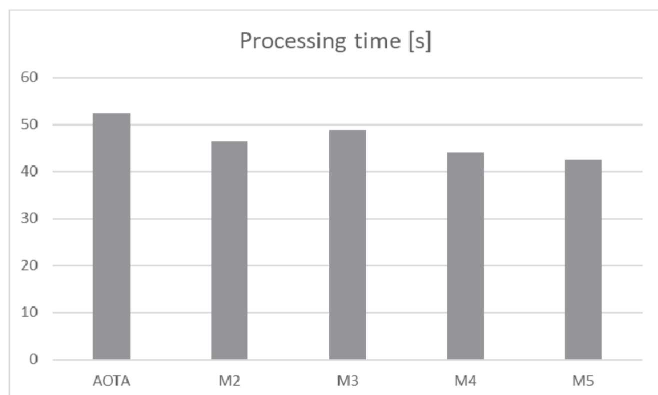


Fig. 15. Processing time results

## IX. Conclusions

Current database systems are characterized by the great data size to be managed. Many data objects are extended by the validity timeframe forming temporal system. Nowadays, many temporal models are available, highlighting the granularity, which is based on, from the point of the whole object, up to attribute granularity or group definition. Effective data retrieval in the temporal environment forces to create a new paradigm and extend current structures and approaches to adapt new trends. This paper can be divided into two parts with emphasis on the server architecture – instance and the physical database. It describes the existing solutions, management, and approaches and proposes new techniques to improve performance on different levels. First, part is based on the time spectrum identification after Select statement definition to form the statement into pre-prepared parsed versions, which require also statistics module extension to ensure complex and reliable data evaluation. Transaction management has been also extended to secure the system and remove transaction connection identification, which can never be completed successfully. Thanks to that, data locks are freed sooner, system resources consumption is lowered, as well. The main contribution of this paper is just the impact of the index structure, by which the data block in the database can be accessed fast. To get the data from the block inside the database, it must be loaded into the memory area – Buffer cache. Performance limitation is just the migrated row, which forces the system to load data blocks, in which, however, data are not located. In that case, pointers to other blocks are used, particular blocks are consecutively loaded. In this paper, we propose several solutions, which are performance described and compared to the existing system. It describes the principles and limitation of the migrated row supported by our proposed techniques to improve solution and remove the impact of migrated row. The first solution is based on the simple interconnection of the migrated blocks, by which the maximal level of the migration is two blocks. It is, however, necessary to cover all the indexes in the systems locating the same row, respectively block, which can be covered by the definition of the mapper located in the database or memory. The best solution is just provided by the memory mapper, by which the costs, CPU and processing time can be reduced significantly (based on experiments, costs are reduced by more than 19%, processing time reflects the improvement on more than 18%).

In the future, we would like to extend the solution by index grouping and mapping. We assume, that it would not be necessary to define own mapping structure, but it would be a direct part of the index reference. We will also attempt to eliminate the need for deactivation and rebuild the mapping structure after next database system startup.

## Acknowledgment

This publication is the result of the project implementation: *Centre of excellence for systems and services of intelligent transport*, ITMS 26220120028 supported by the Research & Development Operational Programme funded by the ERDF and *Centre of excellence for systems and services of intelligent transport II.,* ITMS 26220120050 supported by the Research & Development Operational Programme funded by the ERDF.

This paper is also supported by the following project: "*Creating a new diagnostic algorithm for selected cancers*," ITMS project code: 26220220022 co-financed by the EU and the European Regional Development Fund.

*"Podporujeme výskumné aktivity na Slovensku*
*Projekt je spolufinancovaný zo zdrojov EÚ*

## REFERENCES


[1] K. Ahsan, P. Vijay. "Temporal Databases: Information Systems", Booktango, 2014.

[2] L. Ashdown. T. Kyte "Oracle database concepts", Oracle Press, 2015.

[3] G. Avilés et all. "Spatio-temporal modeling of financial maps from a joint multidimensional scaling-geostatistical perspective", 2016. *In Expert Systems with Applications*. Vol. 60, pp. 280-293.

[4] R. Behling et all., "Derivation of long-term spatiotemporal lanslide activity – a multisensor time species approach", 2016. *In Remote Sensing of Environment*, Vol. 136, pp. 88-104.

[5] C. J. Date, N. Lorentzos, H. Darwen. "Time and Relational Theory : Temporal Databases in the Relational Model and SQL", Morgan Kaufmann, 2015.

[6] M. Erlandsson et all., "Spatial and temporal variations of base cation release from chemical weathering a hisscope scale". 2016. *In Chemical Geology*, Vol. 441, pp. 1-13

[7] J. Janáček and M. Kvet, "Public service system design by radial formulation with dividing points". In Procedia computer science [elektronický zdroj], ISSN 1877-0509, Vol. 51 (2015), pp. 2277-2286

[8] T. Johnston. "Bi-temporal data – Theory and Practice", Morgan Kaufmann, 2014.

[9] T. Johnston and R. Weis, "Managing Time in Relational Databases", Morgan Kaufmann, 2010.

[10] A. Kadir and N. Adnan, "Temporal geospatial analysis of secondary school students´ examination performance", 2016. *In IOP Conference Series: Earth and Environmental Science*, Vol 37, No. 1.

[11] M. Kvassay, E. Zaitseva, J. Kostolny, and V. Levashenko, "Importance analysis of multi-state systems based on integrated direct partial logic derivatives", *In 2015 International Conference on Information and Digital Technologies*, 2015, pp. 183–195.

[12] M. Kvet and J. Janáček, "Relevant network distances for approximate approach to the p-median problem. In Operations Research Proceedings 2012: Selected Papers of the International Conference of the German operations research society (GOR)", Leibniz Univesität Hannover, Germany, Springer 2014, ISSN 0721-5924, ISBN 978-3-319-00794-6, pp. 123-128.

[13] M. Kvet, K. Matiaško, "Transaction Management in Temporal System", 2014. *IEEE conference CISTI 2014*, 18.6. – 21.6.2014, pp. 868-873

[14] M. Kvet, K. Matiaško, „Temporal data Group Management", 2017. *IEEE conference IDT 2017*, 5.7. – 7.7.2017, pp. 218-226

[15] M. Kvet and K. Matiaško, "Uni-temporal modelling extension at the object vs. attribute level", *IEEE conference UKSim*, 20.11 – 22. 11.2014, , pp. 6-11, 2013.

[16] D. Kuhn, S. Alapati, B. Padfield, "Expert Oracle Indexing Access Paths", Apress, 2016.

[17] S. Li, Z. Qin, H. Song. "A Temporal-Spatial Method for Group Detection, Locating and Tracking", *In IEEE Access*, volume 4, 2016.

[18] Y. Li et all., "Spatial and temporal distribution of novel species in China", 2016. In Chinese Journal of Ecology, Vol. 35, No. 7, pp. 1684-1690.

[19] A. Tuzhilin. "Using Temporal Logic and Datalog to Query Databases Evolving in Time", *Forgotten Books*, 2016.