Master Index Access as a Data Tuple and Block Locator

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

Veronika Šalgová, Marek Kvet, Karol Matiaško
University of Zilina
Zilina, Slovakia
{Veronika. Salgova, Marek.Kvet}@fri.uniza.sk

***Abstract*—Relational database systems cover the main part of the current data management of information technology. Data are formed into the relations connected using relationships. Each tuple is physically stored in the database operated by the background processes of the instance. The user query is transferred to the server, analyzed and processed. Data are sent back to the user as the result set. The main part of the processing and query evaluation is just access to the data themselves. If there are a suitable index and conditions, access can be optimized. Vice versa, if there is no relevant index, the whole table must be scanned sequentially, which can bring high demands and processing steps resulting in poor performance. This paper deals with index access methods, the process of the data access, its optimization, architecture and whole apparatus. The main contribution is in our own approach, which aims to remove the need to search the entire table physically by using Table Access Full access path. The structural index denoted as the master is used to access and locate a record with an emphasis on fragmentation options.**

I. INTRODUCTION

Relational databases are still one of the most often used techniques to store data of the information systems. They were firstly defined in the 60ties of 20th century, but they are still so powerful to cover the current environment and technology demands. Approaches and techniques are based on the relational paradigm and mathematical apparatus – algebra to reach the performance [1] [2]. Data are formed in the shape of the *relations* – tables and connections between them – *relationships*. Performance from the logical point of view is ensured by the data structure and optimization. Data themselves must be formed correctly with emphasis on normalization process. It ensures, no data duplicates except connection possibilities are defined. Moreover, no data anomalies can be present [3].

The main property of the relational paradigm is just the transaction. Any change on the data (*insert*, *update* or *delete* statement) is component of the transaction, which can be consequently accepted or refused. Thus, before data being visible publically (to other users, respectively sessions), they must be approved – committed. Transactions ensure the complex correctness and reliability of the data. They must pass all the requirements characterized by the data domains, integrity, user rules up to identification possibilities of the data. The transaction is defined by these four rules [15]:

- *Atomicity* – the whole transaction is either accepted or refused completely. If it is not approved, all changes made inside it are rollbacked – removed and original data versions stay valid.
- *Consistency* – transaction after its approving ensures, all the data requirements formed by the integrity rules are passed.
- *Isolation* – non-approved changes are not visible to other users or sessions. It is ensured by the transaction log management and snapshot techniques, which are described in a complex manner in our previous publication [9] [10].
- *Durability* – data after the end of the transaction must be durable and restorable, even after the system crash.

In the physical database, approved data changes are commonly present. Thus, if you want to get the data from the database, the physical repository is usually contacted to get the relevant data. Ideally, the data image can be present in the memory, but in principle, it should be identical to the image in the physical database, respectively reconstructable with the support of the log files [8].

The connection between *user* and *server* is created via the *listener*, which contacts the particular *Process monitor* background process. It creates the *server process* on the server-side and interconnects it directly with the *user process*. Afterward, the communication is done in a straight manner, without the necessity of any other system cooperation. Fig. 1 shows the architecture of the server and client connection. Naturally, data cannot be operated from the client directly, due to many reasons. The most significant aspect is related to the security, reliability, and integrity of the whole system. Data management, individual access methods, and transfer are protected by the background processes located in the instance of the database server. Instance itself consists of two main structures – *memory structures* and *background processes.*

As the physical database is separated from user-side access, it is essential to develop sophisticated and, in particular, efficient access to data. The aim of this paper is to present own proprietary method by which it is possible to robustly eliminate the need for a sequential search of all blocks associated with a table to locate data. Whereas data are dynamically changing very frequently at present, fragmentation is located at the physical level, which is, naturally, a significant limitation of performance when using the *Table Access Full* method. Own solution is described in section 3. It is based *on the Master index* definition.

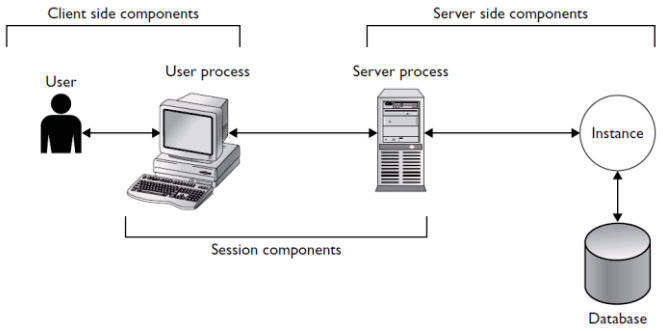

Fig. 1. Communication principles in single instance architecture [12]

II. Technical Background, State of the Art and the Database Context

In the following section, technology and terminology of *DBS Oracle* will be used, whereas the proposed solution has been tested using that system environment. Therefore, memory structures, processes and principles are described for *DBS Oracle*, as well. The main advantage is the efficiency of data management and retrieval, as well as direct interconnection between database approach and procedural language *PL/SQL* and *Java* (covered by the *Java pool* memory structure). On the other hand, our defined solution is universal and can be adapted to any system.

A. Data management memory structures

The architecture of the database server is formed by two structures – *instance* and *database*. *The database* cannot exist without *instance*, respectively it does not have any sense. Vice versa, the *instance* can exist without *database* (e.g. in *nomount* stage of the database system, but without the *database*, no data are available to be managed. The main artifact of the *instance* is *System Global Area (SGA)* allocated at instance *startup* and released on *shutdown*. It consists of several structures, like *Buffer cache*, *Log buffer, Shared pool, Large pool, Java pool* or *Streams pool* [4] [5]. For the query processing and optimization, database *Buffer cache* structure is the most important as the data repository. Queried data before the evaluation must be placed in such a structure. It is shaped as a matrix of the blocks with the same size as the block inside the database itself. In principle, each data change is applied in the *Buffer cache* and noted in the *Log*. Thus, data in the database does not need to be up-to-date, however, based on the present log files, it must always be possible to construct the current image [7] [12]. Block of the *Buffer cache* can be either *dirty*, *clean* or *empty*. *Dirty* block just reflects the fact that the change has been currently applied only in the operating memory and has not been physically entered into the database. *Clean* or *empty* blocks are available to receive new data from the database or by constructing a new one completely. Among this, performance important structures include *Shared pool* formed by *Library cache*, *Data dictionary cache*, *PL/SQL area* and *Result cache* as the repository for the executed code in the parsed form, definition of the structures and objects, stored PL/SQL scripts and metadata [6] [12].

B. Instance processes

Among the memory structures, several processes of the instance must be present to monitor, cooperate and manage the memory structures and the system as a whole, as well. They are launched when the *instance* is started, or explicitly. Some of them are part of the database management core and must be present (if any of them is corrupted, the whole instance crashes immediately), like *System Monitor (SMON), Process Monitor (PMON), Database Writer (DBWn), Checkpoint Process (CKPT)*. Vice versa, many of them are optional associated with the specific job to be done – e.g. *Memory Manager (MMAN*) for the memory structure size optimization reflecting the workload, *Archiver (ARCn)* for non-current log file management or *Recoverer (RECO)*.

Process Monitor manages and supervises all server processes, launches and terminates them. It interconnects the *user* and *server* based on the *listener* requirement by creating the server process in the instance.

Database Writer writes dirty blocks from the *Buffer cache* into the database and frees up space for processing other blocks. The principle of the security is based on the fact of dividing instance and database itself – users cannot access the database directly, it can be exclusively operated just by instance processes. It ensures passing all transaction rules [9].

For the purposes of index management and query processing, the *Log Writer* background process is far important, as well. It writes the contents of the *Log buffer* to the online redo log files on disk to ensure no data loss. Content of the log file covers information about the executed activity, connection to the system via *System Change Number (SCN), UNDO* and *REDO* images to ensure no data loss. By these structures, it is possible to construct the new and existing image, as well, based on the status of the particular database block.

C. Transactions

Transactions as the main database units are important not only for the data changes but the *Select* queries, as well. From the physical point of view, data are always accessed from the *Buffer cache* memory structure. In an optimistic case, the required data may be directly available, whereas they are present in the memory. Thus, the result set is constructed and sent to the user. If the data are not present there, or some portion of them is missing, they must be loaded from the physical database storage to the memory, where the next processing steps are executed. And this part is the bottleneck and performance limitation of the whole system. The point is, how to locate and select relevant data based on the user-defined query? How to access them optimally? What about their shapes?

Physical database storage is delimited by the data files belonging to the tablespaces. Internally, each data file is made up of individual blocks with the same size, usually, 8kB delimited by the parameter of the database, when creating, respectively derived from the internal parameters with emphasis on mapping structure to the memory outside of the *Buffer cache* [7], [9]. Thus, during the processing, the whole block is transferred into the memory, where the relevant data are searched and located. Selection of the block with relevant data can be done using two techniques – sequential scanning or by using index. Sequential data block scanning means, that all blocks belonging to the particular table are loaded to the memory *Buffer cache* step by step, where the evaluation is done. The core of the processing is just the loading (transfer)

process from the physical storage to the memory using I/O operations, which are usually time, technically and resource intensively demanding. Thus, the aim is to avoid such situations and access the relevant blocks more effectively. The first solution was proposed shortly with the development of larger systems where processing took too long, which was unacceptable.

D. Index

One of the main property of the query processing optimization is just an index structure reflection. The index itself is used for direct access to the row inside the database by using *ROWID* on the bottom leaf nodes. *ROWID* is the locator for the data and consists of these layers: identification of the data file, in which the row resists, the pointer to the block and position inside it. Moreover, it uses the specific object identifier, as well. Thus, based on the definition, *ROWID* value is unique for the standalone database [12].

An index is an object of the database with the associated tablespace, which aim is to monitor data and reflect changes inside to be prepared for the query to access the row using *ROWID* directly. In the database systems, various index structures and approaches can be used. The most often used is just the *B-tree*, respectively *B+tree* [6], whereas it maintains the complex efficiency despite frequent changes of records. It does not degrade over time and remains balanced. The structure consists of the tree in which each path from the root to the leaf has the same length [12]. Three node types are present: root, internal node, and leaf node. Root and internal node contains pointers S_i and values K_i, the pointer S_i refers to nodes with lower values the corresponding value (K_i), pointer S_{i+1} references higher (or equal) values. Leaf nodes are directly connected to the file data (using pointers).

Model of the B-tree index structure is in Fig. 2. Leaf layer contains locators of the rows in the physical database – *ROWID* values.

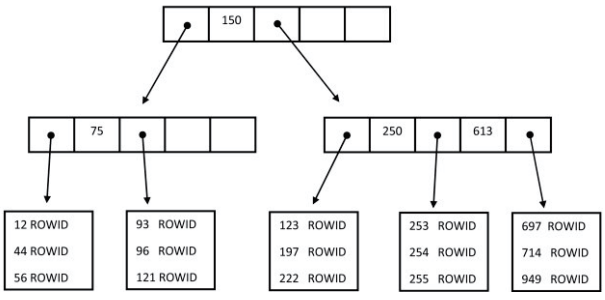

Fig. 2. B-tree

B+tree index approach is an extension on the leaf layer, where individual nodes are chained together forming a linked list. Thus, such layer holds sorted data based on the attributes, which are there indexed [12].

Other index techniques mostly arise from this category and improve either modeled data group or data shapes to be processed, like *inverted B+tree key*, its *unique version*, *table* or *cluster* index. Different structural approaches are based on *bitmap* or *hash* indexes, which cannot be, however, universally used due to specific requirements and limitations [12].

E. Index methods

Query processing consists of several steps, which are consecutively executed. The output of the individual step is transferred as the input of the subsequent one. Fig. 3 shows the query processing steps. *Parser* performs *syntactic* (command grammar) and *semantic* (object existence and access rights) analyzing of the query and rewriting the original plan to a set of relational algebra operations. *Optimizer* suggests the most effective way to get query results, based on the optimization methods, developed indexes and collected statistics. Thus, it selects the best (suboptimal) query execution plan, which is used in the next *Row source generator* step. It creates the *execution plan* for the given SQL query in the form of a tree, whose nodes are made up of individual row sources. Afterward, the SQL query is executed with emphasis on the provided execution plan. The result set is constructed and sent to the client [4] [5] [12].

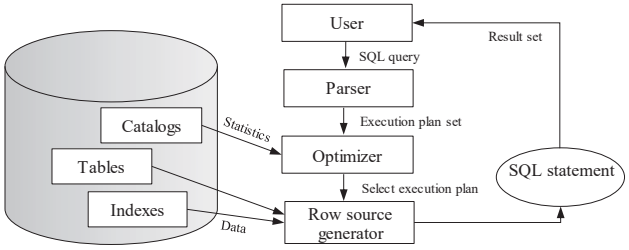

Fig. 3. SQL statement evaluation

The most important step in terms of the processing efficiency, optimization and access rules is just the *execution plan*, which determines usage of indexes. Access methods can be, in principle, divided into two categories – direct access to the data files (*Table Access Full – TAF* method) or access by using an index (*Index Scan*) or just by their combination. *TAF* traverses the entire table, all data blocks associated with the table, which can be physically widespread into multiple data files. The word entire table is significant. For each data segment, *High Water Mark* (*HWM*) symbol is defined determining the last block for the particular structure, which can be generally empty, whereas new blocks are not associated separately, but as the group forming extents. That means, that also blocks with no valid data must be moved into the memory for the evaluation. Even more significant limitation is formed by the fragmentation properties. Block does not need to be completely full, in the real environment, there is significant fragmentation on the block granularity caused by the variability of individual row size, as well as processes of data changes, where the updated row does not fit the originally allocated space [12] [13]. To ensure efficiency and robust performance, it is necessary to limit the usage of *TAF* methods.

The aim of the index definition is to remove such impact. Index for the primary key and unique constraints are defined automatically, others are user-provided. The index can significantly improve the performance of the query, but there is slowdown during data modification operations, whereas the change must be applied inside the index, as well. Therefore, it is not effective, even possible to define all suitable indexes [12] [14]. As a consequence, whereas no suitable index is proposed,

TAF methods are repeatedly used resulting in poor performance and user complaints.

Index scan method category searches the data based on the index. The output of this method can be either whole data set if all of the required attributes are present in the index or set of *ROWIDs*, which are consequently processed and particular blocks are loaded into the memory by using *ROWID scan* method. Following *Index scan* types can be distinguished:

- *Index Unique Scan* – based on the condition, which always produces unique data, thus either one or no rows are selected.
- *Index Range Scan* – standard method, in which the index columns are in the appropriate order, but there is no guarantee that the result will be no more than one record.
- *Full Index Scan* – the whole index is searched in a sorted manner and particular *ROWID*s on the leaf layer are selected. The condition on the leaf layer can be directly evaluated.
- *Index Skip Scan* – method, in which the leading index attribute is not suitable, but the rest ones are appropriate. In that case, it works like the index in the index, thus the first index attribute is skipped.

Point of the processing is, therefore, the suitability of the index. If the order of attributes is not suitable, the index is not used. Let have a table *T* consisting of four attributes: *A, B, C, D*. Let have an index *I* formed by the pair of the attributes *A, B*. If the query requests values of the attribute *C* based on the attribute *D*, it is clear, that the particular index cannot be used. Thus, the *TAF* method is used. The only solution to cover the problem is to create a new index, which has, however, negative aspects in the term of the change management performance. If the system is dynamic with various query construction types over time, the problem is much deeper. The whole table must be scanned sequentially with emphasis on the data fragmentation. Therefore, performance is getting worse and worse. Deleting old records does not solve the problem, too, whereas the number of blocks allocated for the object is never decremented (*HWM* cannot be shifted to the left part of the linked list). The point is therefore clear – propose the solution to cover the problem by removing the impact of full table scan necessity. Next section deals with our proposed solution.

III. Own contribution – index

Limitation of using *Full index scan* method category is based on the fact, that the context reflected by the *Where* clause conditions can be evaluated directly in the leaf layer of the index. Thus, although the order of the attributes inside the index does not fit the query, relevant attributes are present, however in non-suitable order. As a consequence, if the *ROWID* on the leaf layer is selected by passing the *Where* condition of the query, it is certain, that the record will contain the data needed to create the result set. Thus, no irrelevant data block is loaded into the memory *Buffer cache,* except for the migrated row problem.

Our proposed solution uses a different principle. If accepted, the defined index will be used in any case. If the index based on the attributes is not suitable for the query, it will be used only as of the access path to the data blocks with the real data. The importance of our solution definition is described in the following example. Let have *4* data rows for the table. For simplicity, let assume, that each data row is located in the separate data block at the beginning. Then, insert two new rows, which will be located in the same data block. Notice, that the blocks are associated in the object in the form of individual extents, not the blocks directly. So, let assume, that the extent contains two blocks. Thus, after the execution, *6* blocks will be used, the last one will be empty. Now, in the third step, remove the data of the third tuple. What about the results? The third block will be associated with the table but will be totally empty. It is clear, that only four data blocks are relevant for the evaluation, just only they contain the same data portions. By index, they are accessible via *ROWID* values of the index. However, in this case, if the index does not contain the attributes characterizing the query condition, *TAF* approach method is used. Unfortunately, *TAF* method does not have any information about the empty blocks associated with the table, no data defragmentation or migration is done due to performance impacts – such table would be inaccessible during such process, which is not acceptable. Moreover, nowadays, the number of update statements is high and is still rising, thus, data consolidation would require to be executed too often to ensure the benefit, but it is too resource demanding. Overall, the improvement would be minimal, even if any. Thus, by using *TAF* in the described situation, six blocks would be loaded into the memory, but two of them do not provide any data. The global efficiency would be *4/6* – just a bit higher than *66%.* Sure, it is just a simple demonstration of the problem, in the real environment, performance significantly below *50%* would be reached, so more than half of the system work would be unnecessary for the evaluation and the processing. This is, of course, a huge problem in terms of the performance and growth of the data requirements and complexity. Individual processing steps are shown in Fig. 4. The black color of the block represents its occupation, white blocks are empty.

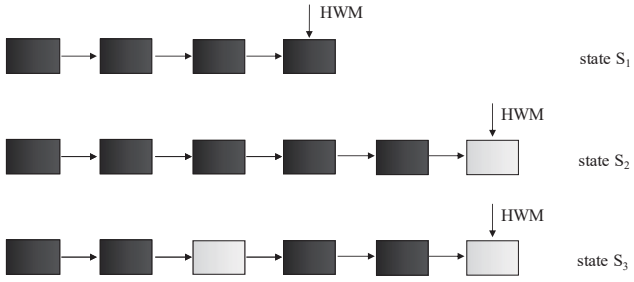

Fig. 4. Data block modeling

In this paper, we propose own solution based on the *Master index*. It uses the fact, that each data row in the relational database can be uniquely identified using the primary key, thus each table usually contains a primary key definition, which has the property of the *uniqueness* and the *minimalism*. Whereas the value of the primary key must be present from the definition (cannot hold undefined *NULL* value) and primary key automatically creates the index, in the system, at least one index exists with *ROWID* pointers to each data are present inside. From this point of view, if the index would be used, just the relevant blocks would be selected. The solution is shown in the data flow diagram in Fig. 5. When the query is obtained to be evaluated, first of all, existing index suitability is evaluated.

If there is a suitable index, naturally, it is used. In this case, therefore, there is no change compared to existing approaches. One of the *Index scan* methods is used, either *Index unique scan, Index range scan* or *Index full scan* with its variants. If there is, however, no suitable index based on the data characteristics and conditions, own proposed solution as the data optimizer extension is used. The system evaluates, whether there is the Master index definition for the particular table. If not, the *TAF* method must be used with all its limitations. However, if one of the indexes is so marked, it will be used, not for the data evaluation, just as data access.

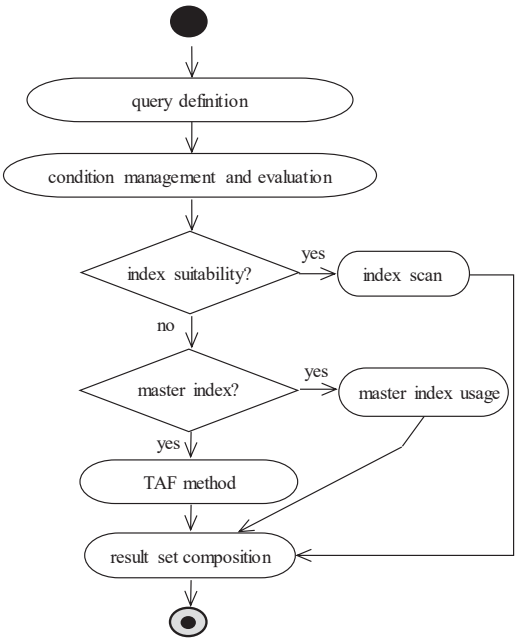

Fig. 5. Data flow – Index access selection

A. Master index definition

There is only one strict requirement for the *Master index definition (MID)* – all data rows must be accessed via it. Thus, it must cover all the data. As mentioned, most often relational database index structure is *B-tree*, respectively *B+tree*. It has one limitation – undefined (*NULL*) values are not indexed. So, if at least one of the indexed column has the property of potentially holding *NULL* value, there is no certitude, that all the data are present by using index. However, whereas in principle, each table is delimited by the primary key definition, such suitable index should always be present. The point is just, whether it is the best suitable or not.

Master index definition is defined for each table and can be selected either automatically or manually based on the user decision. The decision for the table can be selected this way:

Alter table <*table_name*> set MID=<*index_name*>;

In the previous case, the user defines the *Master index* manually. Thus, if the index is dropped or denoted as corrupted (needs to be rebuilt), *MID* parameter is automatically set to *NULL* and proposed technology will not be used later. Thus, if the setting for the table would be *NULL*, the proposed extension would not be applicable for the table resulting in using original *TAF* method.

Alter table <*table_name*> set MID=NULL;

Selection of the *Master index* can be done automatically by the system, as well. The decision is done by the optimizer based on the current statistics of the index and the whole system. Suitability of the index to be declared by its size on the leaf layer. Generally, the fewer amount of nodes indicates better performance. Another aspect of the selection is just the availability of the index in the *Buffer cache* memory structure. Similar to the table, the index must be loaded into the memory to be processed, as well. If some index is already available there, either partially, the process of the loading using I/O operations are removed, respectively shortened. Therefore, it is gainful to use automatic system management and decision making. The option is done on the table granularity by using the following command:

Alter table <*table_name*> set MID=*AUTO*;

The advantage of this approach is reflected by efficiency. If some index is dropped, the system automatically evaluates, whether it is marked as *master* or not. If so, a new suitable index for the processing is selected, if possible.

B. Index master method

In the previous paragraphs, the principle of *Master index* definition selection is described. For now, it is necessary to explain the principle of data access. *TAF* method principle is characterized by the sequential scanning of all data blocks associated with the table. It uses the fact, that the individual blocks are formed in the extent shape, which is linked together [11], [12]. As described, for the processing and evaluation, the block is always loaded into the memory, even if it does not include relevant data for the query, respectively, it is empty. Thus, in the first phase of the development of own approach, the aim was to remove such blocks from the evaluation. Our firstly defined solution was based on two sides linked list. Each block then consisted of the information about the fullness of the direct following block (***model 1***). Thanks to that, the empty block is skipped from the evaluation. The disadvantage is the necessity to store the two way linked list and modification of the whole path after the change on the block level, as well. Our second proposed solution (***model 2***) improved the original approach by storing the pointer to the next used data block. The principle is shown in fig. 6. Black block is occupied, white is free. Let assume, that the third and last associated blocks are empty. Bold arrows indicate added pointers to the other blocks. Whereas the last block is free, the fifth can point either to the same (fifth block) or *NULL* pointer can be used regarding the consecutive way of evaluation. *NULL* is better from the size point of view but worse for the management.

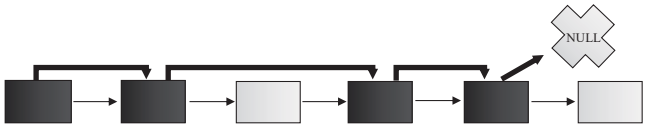

Fig. 6. Model 2

After the complex experiments, we came to the conclusion, that the proposed solution is robust, but not optimal. Inside each block, specific space had to be allocated for the pointers and fullness management. As a consequence, each block itself had to be shortened based on the size. Therefore, we define also another solution (***model 3***), which is just based on the *Master index*. It is used as the source of pointers to the data blocks. Is it

possible to determine existing and at least partially occupied blocks? Sure, it is, by using *ROWID* pointers at the leaf layer. To ensure the efficiency, we added new parameter associated to the *MID*. It holds the pointer to the first node at the leaf layer of the index. Thanks to that, it is not necessary to traverse the whole index from the root. Name of the parameter is *MID_pointer_locator* and is maintained automatically, thus if the structure of the index is changed as the result of rebalancing, such parameter is automatically notified to ensure correctness.

MID_pointer_locator gets the first index node for the processing, respectively the first *ROWID* pointing the block inside the database. *B+tree* has linked list on the leaf layer, therefore individual data segments can be directly located from that level. Logically, there is a list of *ROWIDs*, which are used to access the physical data. Thus, non-relevant data blocks are not processed at all, whereas no *ROWID* points to them.

Our proposed approach uses the *Private Global Area (PGA)* of the server associated for each session separately. In this structure, local variables are stored. In our case, we use it for the list of individual blocks. Multiple rows can be located in the same data block. Therefore, before the evaluation, the address of the block is extracted from the *ROWID* value, which is consecutively checked, whether such block has already been processed or not. Notice, that the whole block is evaluated, not only the row itself. The reason is based on the efficiency of I/O operations. It could happen that block with multiple records is read into the memory. If only one record was evaluated, such a block would have to be processed later for further records. In the meantime, however, the block could be removed from the *Buffer cache* as it is a *clean* block type - no changes were made to it. Thus, the number of I/O operations would increase beyond the number of blocks actually used. Diagram expressing the processing steps is shown in Fig. 7.

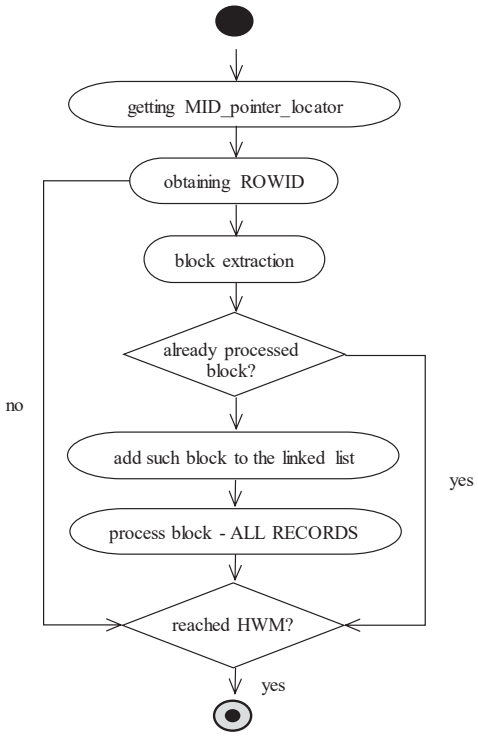

Fig. 7. Data flow – *MID_pointer_locator* and consecutive data management

C. ROWID vs. BLOCKID

In the previous definition, the principle of using *ROWID* values to identify blocks was used. As described, management was extended to check, whether such block had already been processed or not. The aim of the solution was clear, to minimize the amount of I/O operations, which is part of the most expensive operations of the systems themselves. As a result, it would be grateful, if the solution can use identifiers of the block on the leaf layer instead of the *ROWID*s. It is, however not possible directly, whereas there is no possibility to modify existing index approaches in the core of the database system. The solution is, therefore, based on two interconnected index structures. One of them resides the original and consists of the *ROWID* values in the leaf layer. The difference is, that they do not point to the data blocks in the physical database, but are routed to the second index. Pointers are always paired – from the index to the block module and vice versa as well. Thanks to that, any change on the block management can be easily identified and the whole supervising layer can be notified. Block module form is similar to the index; it uses the *B+tree* structure too. On the leaf layer, pointers to the physical database are on the block granularity. If any block is freed, respectively associated without particular data, such blocks are not part of the *block module* and automatically skipped. *Master Index* method uses only the block module and scans the blocks in a parallel manner. If there is any change on the data, the original index is used, which, however, automatically reflects the change in the block module, if any change in the segment or extend block positions are done. *Select* statements use direct access to the block module. The architecture of the solution (***model 4***) is in Fig. 8.

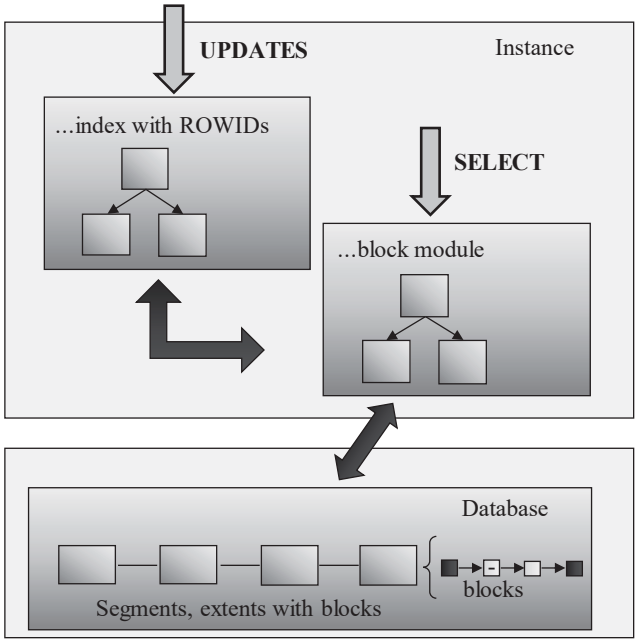

Fig. 8. The architecture of the solution

IV. RESULTS

Performance characteristics have been obtained by using Oracle 11g database system based on the relational platform. For the evaluation, a table containing 10 attributes were used, delimited by the composite primary key consisting of two attributes. No specific indexes were developed, therefore the primary key was denoted as Master index.

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.

Results and performance solution management were applied to the models described in the previous sections of this paper. Four models were evaluated. *Model 1* extends the block definition by the information about the fullness of the direct following block. The negative aspect was identified, if multiple blocks in the linked chain were free, located together in the group. *Model 2* removes such constraint and contains the pointer to the following used block with relevant data. As evident from the results, the proposed solution brought relevant improvement in terms of processing time and size of the whole structure, as well.

Model 3 uses different architecture. It does not modify the physical structures inside the database block but uses our proposed *Master index* approach. Thus, if the index definition is not suitable, the marked Master index is used to locate the data. This approach is very convenient if the data tuples are modified very often with various size demands for the attribute values. Solution plays a significant role also in cases of data fragmentation in the database structure. Improvement of the solution is carried by the last *model 4*. Described in section C of chapter 3.

Obtained results are shown in Fig. 9, which reflects the performance expressed by the processing time in the second precision. Values in the graph express the improvement or slowdowns in the processing time in percentage. The referential model uses the original method for data access – *Table Access Full.* As evident, all of them offer significant improvement. *10%* of the data were part of the result set. In optimal conditions, perfectly defined index for the query would require *10%* of the processing time in comparison with the *TAF* method. In our case, *model 1* obtained *23%* load, *model 2* required *21%*. Significant improvement was reached when using *model 3 – 17%.* Architectural model 4 was the best and required only 13%. Thus, *3%* of the processing was associated with *Master index* management with an emphasis on the data location in the leaf layer. Notice, that optimal solution would require no data fragmentation, which is, unfortunately, very difficult to ensure in the real system environment, where the structure and size of attribute values can vary significantly. As a consequence, real deployment would degrade to use fixed-size variables, mostly strings, which, of course, is not entirely appropriate for disk space requirements.

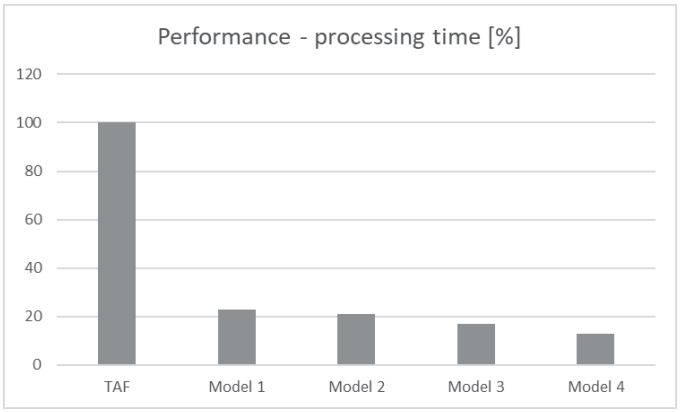

Fig. 9. Processing time results

When dealing with the size demands for the whole structure, the following results were obtained. Original solution with no specific structure and management added was used as referential. Values are in percentage expressing the additional demands. Results are in Fig. 10. *Model 1* reduces the size of the block itself to extend the header to store information about the next block and load it. It required additional demands of *5%. Model 2* uses only pointers, which does not need to indicate direct following block if it is empty. It removes the impact of the free block grouping, as well. It required just *3%* of additional size demands. Slight differences using only *0,1%* can be identified in *model 3*. It does not, namely, use any additional structure, just one of the indexes meeting the requirements is marked as *Master*. Model 4 is the most complicated, whereas additional index on block granularity is used. In this case, the size requires an additional 12%.

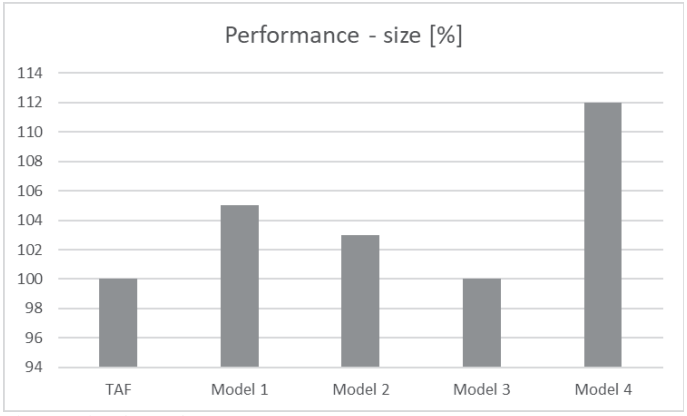

Fig. 10. Size demands

The last experiment in this section is based on evaluating the rate between block occupation and free blocks caused by data restructuralization, shifting, and fragmentation. Results are shown in Fig. 11 expressing the rate of actually used blocks (block, which consists of at least one relevant tuple inside). Again, we use our four designed models, that are paired to the original *TAF* access method. Based on the results, the suitability of the *model 1* is limited by the value *63*, *model 2* limitation is *65*. *Model 3* can work effectively up to *81* and *model 4 – 84*. These values express the rate between free and used blocks. Thus, the best performance was obtained by the *model 4*, which can work effectively up to *16%* (100 - 84) of free blocks. If the rate is less (number of free blocks is lower

than 16%), original *TAF* is more effective, although non-relevant blocks must be scanned. If there is no free block in the system – all of them have at least one consistent data tuple, reached results are following (expressing the slowdown of the system in terms of the processing time). Values are expressed in percentage:

- Model 1 58%,
- Model 2 51%,
- Model 3 23%,
- Model 4 23%.

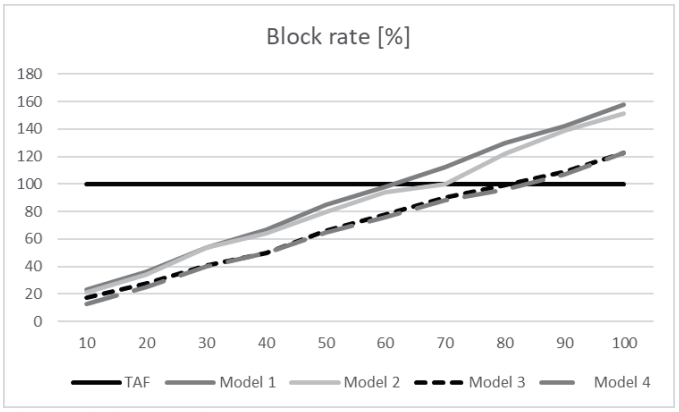

Fig. 11. Block rate results

Proposed solution has been tested in the *DBS Orac*le environment, but can be adopted to any relational system, whereas the principles are the same. We assume that *DBS Oracle* is the most comprehensive and powerful system and technology at the same time [2], [12]. In the near future, naturally, another paper will be published to compare individual database systems and proposed solution performance applied to it.

V. Conclusions

Effectivity of data processing is one of the most significant tasks to ensure the performance of the whole system. Nowadays, the number of data to be handled is high and is still rising. The structure of the data can evolve, as well. Moreover, such data dynamically change their values and properties over time. As a consequence, table complexity is still rising and more and more new data fragments are identified. Therefore, it is clear, that these factors must be taken care of when the database system management is defined. Many systems can be connected to the same database producing various data analysis. Thus, data queries can vary significantly resulting in poor performance, whereas it is not possible to develop all indexes for the defined properties. Sequential scanning of all blocks associated with the table is the last step before the total collapse of the system and user as well, whereas they are not willing to wait for the result sets. The aim of our proposed solution is to limit the necessity to use sequential data block scanning performed by the *Table Access Full (TAF)* method. Our technology uses the *Master index*, which is not used for the evaluation itself, whereas it does not fit the conditions of the query. The core is that it contains all the pointers to the data on the leaf layer. Therefore, the index itself is used as the data locator. There are two proposed models highlighting the *Master index* definition. The first one is based on data row granularity, the second one is shifted to the block identification and uses two indexes. Based on the reached results, the best solution reflects block granularity. In this case, in comparison with the original *TAF* method, performance in processing time was lowered to 13%, if one-tenth of data should be provided in the result set. The principle is based on removing the evaluation of free blocks, which must be transferred into the memory in a standard manner. Vice versa, there is approximately *12%* of the increase in size demands. It is caused by the necessity to develop a new index on the block granularity for the queries. In the near future, we would like to lower the size demands to sharpen to performance of the whole solution in such an aspect, as well. We will also deal with the solution reflection in the environment of the temporal systems, where several non-defined values and the whole states can be present.

Acknowledgment

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

The work is also supported by the project VEGA *1/0089/19 Data analysis methods and decisions support tools for service systems supporting electric vehicles* and *Grant system* of the University of Zilina.

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

[4] 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

[5] 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

[6] T. Johnston. “Bi-temporal data – Theory and Practice”, Morgan Kaufmann, 2014.

[7] T. Johnston and R. Weis, “Managing Time in Relational Databases”, Morgan Kaufmann, 2010.

[8] 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.

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

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

[11] 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.

[12] D. Kuhn, S. Alapati, B. Padfield, “Expert Oracle Indexing Access Paths”, Apress, 2016.

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

[14] 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.

[15] A. Tuzhilin. “Using Temporal Logic and Datalog to Query Databases Evolving in Time”, Forgotten Books, 2016.