Database Block Management using Master Index

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

***Abstract*—A database is formed by a set of data files holding the data. These files are block oriented. Each row can be located by the ROWID address pointing to the data file, data block, and its position inside the block. For processing, block granularity is used for memory loading and evaluation. However, a block is fixed in size, thus, during the Update operations, block fragmentations can be present. Moreover, once the block is associated with the table, it is not commonly deallocated, whereas it is part of the extent, not allocated individually. All these facts have strong importance and impact on the performance of the data retrieval, mostly in the case of sequential block scanning. This paper deals with the Master index extension to locate fragmentations, manage shrinking and identify empty blocks. Thanks to that, database performance can be significantly improved. The study deals with the temporal environment.**

I. INTRODUCTION

Data management in the IT industry has a strong history. In the initial phases, data were embedded into the application, limiting the migration and consecutive usability in another system. Later, the data were separated into a specific layer, delimited by the data file management, with no complex data management and retrieval supervision. Currently, databases are not just separate layers, they are excluded into separate, hardware and software optimized servers [1]. The whole activities are covered by the database systems. Relational databases are formed by the data model consisting of the entities and relationships between them, forming referential integrity. The whole consistency and overall integrity passing are supervised by the transaction support ensuring the data correctness. Although the data are still stored in the data files, the user cannot access them directly, instead, the instance operated by the background processes is used. Fig. 1 shows the data flow of the system architecture. The user is delimited by the client (user) process contacting the database listener, which ensures the mapping by creating a server process and interconnecting it directly to the client site. A server process is extended by a small memory structure dealing with the local variables, cursor states, and parameters, called Private Global Area. The database instance itself is formed by the memory structures and background processes supervising the infrastructure and database [1] [2]. Thus, there is a strong data separation and client operations. The data transfer, loading, and result set building, as well as change operations, are maintained by the background processes and transaction managers. Data are block oriented in the data files, which are consecutively treated and loaded to the memory for the deeper evaluation in the memory Buffer cache, which block size reflects the same principles as the physical data repository itself.

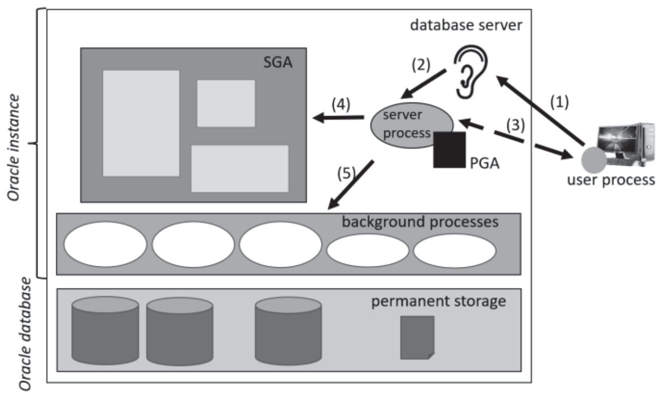

Fig. 1. Database system architecture - user and server process mapping

As evident, a database is block oriented and forms the main unit of the whole data processing. Data are stored in the blocks, a block is the smallest unit for the data transfer, and the memory Buffer cache is also formed by the block matrix. The crucial element is its size related to the tuple structure. Typically, the size of the tuple is not fixed, the size of the attributes can vary, like for character strings, as well as various data value precision. Moreover, individual values do not need to be present, modeled by the NULL value notation. All these factors form data block fragmentation [2] [3]. By using huge Updates, even empty blocks can be present [3].

This paper deals with the data block optimization, shrinking space, and relevant block identification by the proposed Master index structure, pointing to the block precision, instead of rows, which are characteristic of conventional indexes. Thanks to that, the performance of the data transfer, evaluation principles, and disc storage demands can be optimized.

The whole management is operated on the temporal database layer [3] [4], whereas such an environment is expressly associated with the precision changes and high frequency of state updates; however, the solution can be applied using any relational database platform.

The proposed paper is organized as follows. Chapter 2 deals with the temporal database principle and modeling summary pointing to the various architectures and granularity levels. Temporal models form the core environment for the proposed solution management. Chapter 3 refers to block management and related problems limiting the performance of the system. The main contribution of the paper is made in chapter 4.

II. Temporality – Granularity Models and Architectures

The concept of temporal data management is based on the tuple identifier extension by dealing with the Date and Time spectrum. At least, one temporal dimension must be present, forming a uni-temporal system, which commonly covers the validity of the object state. Bi-temporal model deals with two-time dimensions. Validity is typically enhanced by the transaction reference allowing to store data corrections for any existing state. In general, a multi-temporal system can be present referencing any time sphere. One aspect is important – sortability and timeline reflection. Any temporal meaning is used, it must be always possible to identify, which state occurred sooner, to define the borders, and to allow data corrections. Simultaneously, it must be ensured, that the object cannot be covered by more than one valid state anytime, powered by the extended temporal integrity.

There are several time spheres to be referenced, like validity, transaction reflection, data load reference, data transaction approval, synchronization timestamp, etc.

Object-level temporal architecture is based on the original object identifier (primary key) extension by the temporal sphere. Thanks to that, multiple states can be accomplished for one object, referenced by the timeline position. Considering that, any change of the object attribute forces the system to create the new state completely, regardless of the real change for the particular attribute set. Consequently, particular values can be stored multiple times, even if the value is not changed, the original value must be copied to form a new state. One big advantage of such a model is related to the consecutive state composition, which is straightforward, only time reference must be identified and the row expresses the state. A more complicated process, is, however, expressed by the real change identification. It is necessary to get the direct predecessor and compare values attribute-by-attribute. Additionally, there can be huge storage demands, if the change frequency rate is not the same for all attributes. Namely, there can be static attributes, which do not have their values over time. Object level temporal model is shown in fig. 2.

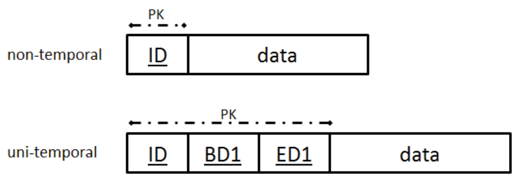

Fig. 2. Uni-temporal model [4]

A different approach is delimited by attribute granularity, which forms the basic unit of temporal processing. By using attribute oriented temporal model, each table column, which is temporally registered is extended by the Date and Time frames. Thus, the overall state of the object is formed by individual attribute value compositions. No duplicate values are present, each attribute can be enhanced by different temporal positions and spheres, e.g., some attributes can be validity modeled, the others can be covered by the validity and transaction reference allowing to store data corrections. Moreover, attribute-oriented granularity can sophistically cover static attributes or conventional attributes, by which the evolution is not monitored – only one current valid state is stored, and history and future valid data are not processed. The efficiency related to the change frequency is both a strength and a weakness of the whole approach. Namely, if an only subset of attributes is changed, only those are processed, by using attribute granularity. Thus, any attribute change reaches one Insert statement to the temporal layer. So, if multiple attributes were synchronized in the change process, regardless of set size, each attribute would be processed separately.

The architecture of the attribute-oriented temporal model is in fig. 3. Applications communicate with the current valid state layer, as well as temporal management to obtain historical or future valid perspectives. Temporal management layer can deal with any data and is considered as a supervisor of the whole process. Actual and outdated data layers cannot share the data directly, it can be done only by accessing temporal management layer.

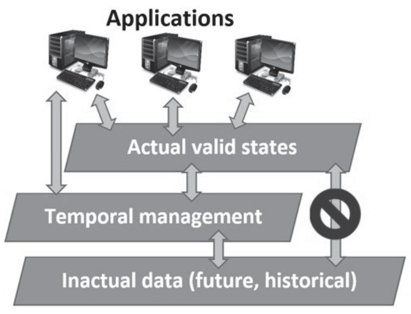

Fig. 3. Attribute oriented granularity [5]

The group-level temporal architecture was defined in 2017 by introducing a data_val object, which can be formed either by one attribute or a synchronization group consisting of multiple attributes or sub-groups. Data_val object is a core granularity used in this model and is temporally oriented. Thanks to that, the processed precision can reflect any granularity by covering any temporal model sphere. The architecture of the group-level temporal model is the same as attribute oriented, but there is no management of attributes, just the data_val object is referenced. Fig. 4 shows the data model of the composition.

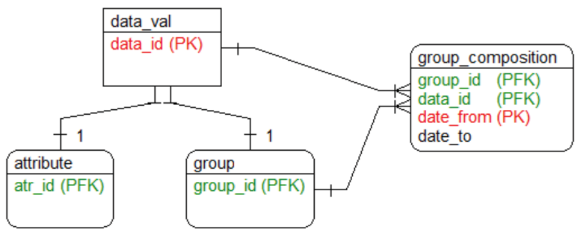

Fig. 4. Data_val composition [5]

Although the proposed solution of the Master index extension is primarily intended for dynamic and frequent data changes by focusing on temporal evolution management, the solutions can be applied generally in any system. Even very simple conventional systems can profit, whereas fragmentation is present in any solution that offers dynamic tuple size.

III. BLOCK MANAGEMENT AND RELATED PROBLEMS

Each data object – table or index – is formed by the segment. The data object segment consists of the object definition, parameters, properties, and pointer to the associated object data. These data are stored in the blocks, however, such blocks are not allocated separately, instead, a set of blocks is created at once, called data extent. Individual extents are chained together forming the list of associated object data blocks. The last associated block is referenced by the High Water Mark (HWM) pointer, forming the upper block limit. By sequential scanning, the first block is defined by the segment pointer, the last block is pointed by the HWM. The scanning process is done using block granularity. Blocks can be differentiated based on the data style, either a data block or index block can be identified [6] [7] [8].

Thus, one object is generally formed by one segment. In the case of using table partitioning, each partition is then reflected by one segment. One segment has an unlimited number of associated extents, which should correspond to the amount of data stored in them. However, that prerequisite is true only in optimal conditions. Individual database systems do not deallocate data blocks, if they are empty, for two main reasons. Firstly, data blocks cannot be allocated and deallocated separately. Thus, to do that, the whole extent should be empty, which would require additional copying and transferring data. Secondly, data storage enhancement is really demanding process, consuming many system resources. It also requires significant processing time in an additional manner. Whereas it is assumed the data blocks will be used in the recent future, they are retained in the table to limit consecutive allocation necessity. However, that´s just the performance limitation, if sequential scanning is used.

A. Data access methods

Full Table Scan (Table Access Full – TAF) is a common access method, by which all the associated table block set is scanned sequentially, block-by-block. It requires taking the block and transferring it to the memory Buffer cache for the consecutive evaluation and tuple identification, which is then treated regarding the conditions of the query. Thus, there must be a decision, whether a particular row passes all the conditions to be part of the result set in a required format, delimited by the Select clause of the query. TAF is the most demanding access method, whereas it is not prone to block fragmentation, even non-relevant or free blocks are loaded lowering the precision and overall efficiency. It is used, if no better access method using an index is available or the query evaluator background processes decide that using an index would be more demanding than TAF method usage. Although a suitable index may be present in the system, based on statistics it is clear that if more than 15-25% of the index data are selected, sequential scanning is preferred [1] [2] [9].

An index is a specific data structure, extending the table definition and data management, by allowing access and identifying rows more effectively by using the index key. Index in relational databases is commonly B-tree oriented, which is resistant to the data changes without efficiency degradation. Based on [3], it is rather molded to width than to depth. Thus, 200 million rows require to use of only 4 nodes to identify the row in the leaf layer. Root and internal nodes of the B-tree are used for the traversing, leaf nodes consist of the data pointers – ROWIDs, which are delimited by the data file, data block, and position of the row inside the block. From the logical perspective, the ROWID value is unique and requires 10 bytes.

Among the B-tree enhanced by the leaf layer data sortage forming B+tree, relational databases use bitmap indexes, mostly related to the analytical environment or hash indexes splitting the data into multiple buckets or partitions.

By using the index, various access methods can be used. If the index key is marked as unique and the Where clause is based on the equality regarding the index key, Index Unique Scan (IUS) can be used, by which no more than one row is returned. In contrast, Index Range Scan (IRS) can produce any number of rows. It takes the list of ROWIDs obtained by the index traverse, followed by particular row loading into the memory Buffer cache for the evaluation. The granularity of the loaded data is always the block itself.

Other index access methods are Index Full Scan (IFS) and Index Fast Full Scan (IFFS) methods. IFS reads the entire index and uses the fact the index items are leaf node sorted. IFFS is an analogy of the TAF, but the amount of processed data is limited, whereas the index is commonly distributed over a smaller number of blocks. Moreover, not all the data attributes are indexed, lowering the storage demands, as well [9] [10] [11].

A specific access method has been introduced in Oracle database 9i. It uses the physical orientation of the composite index key and skips the leading index to obtain the data. Index Skip Scan is initiated by probing the index for distinct values of the prefix column. Each of these distinct values is then used as a starting point for a regular index search. Afterward, the partial result sets are merged. As a result, the index is scanned from the second layer. The database optimizer selects this method if it assumes the total demands are lower than TAF. Note, that index structure is optimized by getting the same depth for any covered tuple. The disc storage reflection is also optimized, compared to the entire table.

From the performance point of view, the index can be considered an ideal solution. Data block addresses (ROWID values) are extracted, which form the most efficient way to identify and access the record in the block. So, is there any limitation? Sure, it is. ROWID should be precise, however, there are many situations, which result in ROWID "desynchronization". Firstly, migration automatically invalidates existing ROWIDs, whereas the addresses are changed. Secondly, multiple indexes can point to one tuple. If the tuple position address is changed, original ROWID pointers remain in the indexes. And it very often happens in dynamic systems, where the record after the Update operation can no longer be served by the original location and the system has to find a new block for it. A migrated row is another performance limitation related to the index. If the index is used, particular ROWID is not precise. By loading the original block to the memory Buffer cache, the system identifies just the pointer to a different block, which must be memory loaded and evaluated. It is evident, that the amount of I/O operations is rising. It is inevitable to highlight that the multiple nodes can be affected by it forcing the system to load several blocks to find the particular block. In the optimal settlement, just one would be enough.

Thus, it is evident, that data fragmentation is a critical factor influencing the performance of the whole system. Temporal databases are characterized by storing object state evolution in time frames. Sooner or later, however, historical data do not need to be covered in an original form and can be removed by deleting such rows or by moving them to another repository, typically data warehouses, data archives, or lakes, operated by the ETL (Extract-Transform-Load) process. Consequently, when moving the data from the original repositories, many fragmentations of the blocks are present, even empty blocks are part of the extents. Database systems do not mark them specifically and remain them in the system. TAF method can degrade significantly, free blocks are loaded and evaluated.

Concluding this section, three main problems are identified by proposing own solutions described in this paper:

- Relevant block identification by limiting empty blocks from the evaluation strategy (TAF).
- Limiting data migration by improving the performance of the data access to locate the data consistently (index access methods).
- Consolidating the data by using proposed block shrinking and defragmentation methods.

B. Physical definition of the block

Oracle storage management unit is a data block by multiplying Oracle data blocks, not operating system blocks. To ensure performance, the Oracle data block should be a multiple of the operating system block size, whereas physically, all the activities are always supervised by the operating system, pointing to the internal processes and operations. The size of the Oracle data block is delimited by the DB_BLOCK_SIZE initialization parameter of the database [1]. Without a total data reconstruction, it is unable to change it later. DB_BLOCK_SIZE defines the size of the element of the Buffer cache matrix. Typically, the block size is 8KB. Besides, five non-standard block sizes can be defined and referenced. However, to make it usable, additional memory structures must be allocated and formatted regarding the particular block size. The data block itself is a complex structure and can be differentiated into several parts. Fig. 5 shows the architecture of the block. It consists of the header consisting of the general block information, like a reference to the segment (data or index). Table Directory structure contains the information about the table origin covering rows by that block. Row Directory refers to the actual rows in the blocks. It contains the addresses for each row piece. Header, Table Directory, and Row Directory are commonly named Overhead, which can have variable sizes, however, it refers to 84 to 107 bytes on average [12]. Row Data is a core part of the block consisting of the values for individual attributes of the rows, referred to as the block working area. Free space of the block is used for the Insert and Update operations, in case the original size is smaller than the state after the change. Free space can be also accomplished by the transaction entry requiring a small information piece for any Insert, Update, Delete operation and cursors (for Update type).

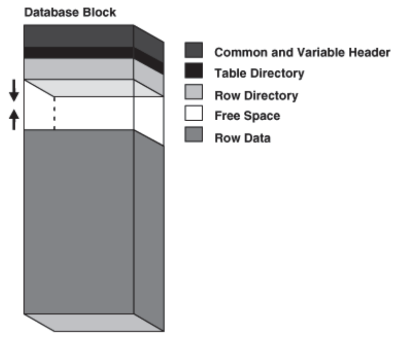

Fig. 5. Database block structure [12]

C. Free space block optimization

Delete and Update statements can increase the free space inside the block if that Update operation changes existing values to the smaller values from the storage perspective. The released space is, however, available just for the same transaction or makes it available after the transaction approval. Thus, any other concurrent transactions cannot use during the run. Generally, free space is not continuous making the fragmentation deeper. Oracle does not coalesce the free space, as long as it is not necessary. It does the compression only if it is clear, new data will form and can be covered by the free space. Note, that the compression causes local block data migration. Namely, ROWID also points to the data block position, however, during the compression, particular rows are shifted, although they are still part of the original block. Therefore, by using the index, the relevant block is identified and loaded optimally, but the row pointer is not precise forcing the system to rescan the whole block [12] [13] [14].

D. Migrating and chaining rows

The row can be too large to fit into one block, even during the Insert, but also the Update operation. It typically reports the column of LONG or LONG RAW data type. In that case, the row must be effectively reconstructable by accessing all blocks converting that row. Oracle uses a chain of blocks reserved for the segment. Row chaining is a standard activity and is unavoidable. Combining various data block sizes is not suitable, whereas it would be necessary to spread information across multiple memory structures, operated by multiple background processes. Moreover, it would be necessary to be spread across various data files with specific data block size representation making the solution too complicated. And finally, it would require a sophisticated solution in the area of block transformation, the transfer and optimization of access not only from the block perspective, but also related to the index structures and methods. Block chaining cannot be directly optimized, if it occurs frequently, there is just a recommendation to reconsider the block size apparent for the table.

Data migration occurs if the row length increases after the change and there is no free space in the block to serve the extension. In that case, Oracle migrates the whole row to another block by the assumption, the newly provided block can fully cover the row. The original row piece is retained, extended by the pointer to another block. The ROWID inside the index does not change and points to the original position.

Row chaining, as well as migrated row presence, decreases I/O performance, whereas more than one row must be loaded to retrieve the information value of the row [1] [15] [16].

IV. The proposed solution – Master index extension

This section deals with the proposed solution focusing on the individual challenges summarized previously. Subpart A deals with the Master index definition, architecture, and usage. Subpart B deals with the space management enhancements, C delimits the migration management using the Master index, D defines the searching priority based on the data block usage and E refers to the block space shrinking.

A. Master index definition

The master index is based on the assumption, that each table has at least one index, at least defined by the unique row identifier – primary key. In principle, for the Master index definition, any index can be selected, there is just one prerequisite – it must cover all the table tuples - rows, which hold NULL values for the whole index key, are not indexed, whereas it would be impossible to provide traverse operations through the index to locate index leaf. NULL values cannot be mathematically compared.

The Master index selection can be done either manually by using ALTER table command, or the selection can be done automatically by the database optimizer. It prefers precisely those indexes that are small in size since their browsing is shorter than in the case of other indexes or those that are the most efficient for processing. Efficiency is in this case expressed by the cost of loading the index into memory for evaluation. Thus, a relatively complex index may be preferred, because it is already loaded in memory Buffer cache entirety, or its proportional part is present there. And hence, the overall processing can be less demanding and faster.

The manual decision for the table can be selected as follows:

```
ALTER TABLE <table_name>
   SET MID = <index_name>;
```

Automated selection and management are more preferred, whereas it can be dynamically changed based on the current data memory perspective:

```
ALTER TABLE <table_name> SET MID = AUTO;
```

To disable Master index selection, NULL is set.

```
ALTER TABLE <table_name> SET MID = NULL;
```

The master index is used as a pointer locator to the block set, which holds real data. Thus, sequential scanning is transformed into the Master Index Scan (MIS). It is analogous to TAF, but the scanning of free blocks is automatically refused, whereas there is no pointer to them from the Master index. To optimize the performance of the MIS, for the particularly marked index, an additional layer operated by the Block extractor background process is created. There is also a B+tree index structure, but the leaf layer contains only pointers to the block themselves, not the rows. It uses logical pointers to the additional layer, which refers to the list of blocks, which hold at least one data tuple. Fig. 6 shows the data flow of the Master index usage. It is clear, that there can be significant benefits related to I/O loading operation.

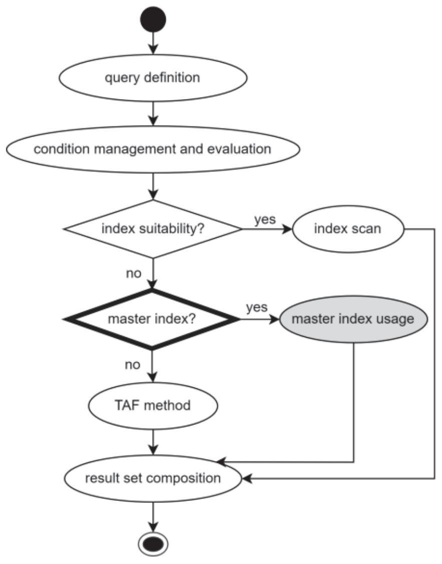

Fig. 6. Data flow - Index access selection [3]

There is still sequential block scanning using MIS methods, however, it does not refer to all associated blocks pointed by the segment and HWM, instead, only relevant blocks are scanned. The proposed MIS method, if enabled, completely replaces the TAF method.

B. Free space management

The primary purpose of the Master index definition is to identify relevant blocks to be used for the evaluation and loading during the data retrieval process. It replaces the sequential scanning necessity of the whole block set associated with the table segment through the extents.

However, how to identify free blocks available for holding new data tuples? In principle, the result is composed of the difference between two sets – all associated blocks, reference by the extent of content and its interconnection with others, and the set of used blocks, which can be obtained by the Master index. The original solution used the mathematical operation difference, but the performance was not optimal. Namely, during the Insert operation, it is necessary to identify one block, which can cover the requirement, not to reflect all the blocks. Therefore, the Master index extension has been proposed, by introducing a Block reference module. It consists of a list of all blocks, that are associated with the table, divided by the extent and data file affiliation, forming the index key. The master index then does not reference the physical address of the block, just the Block reference module is reflected. The physical address of the block is then part of such a structure. Thanks to that, blocks, which are not referenced, are free. Moreover, by grouping individual blocks together based on the extent membership, it is possible to easily identify completely empty extents, which can be optionally deallocated from the table. Fig. 7 shows the architecture of the Master index enhancement by the Block reference module.

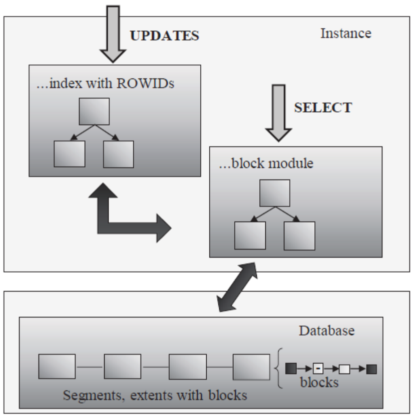

Fig. 7. Master index – Block reference module

Note, that the interconnection between the Master index and Block reference module is done by a two-side linked list, thus, from both structures, unused blocks can be referenced. The master index is preferred for the data retrieval process (Select statement), whereas the Block reference module is optimized for the Insert statements.

C. Migration management using Master index

The core granularity of the Master index is the block itself, but there is still an ordinary index origin, which can be used, or incorporated, respectively. Before proposing this solution, migrated rows were handled by rebuilding indexes [2] or by using logical data pointers to the physical infrastructure [3]. The optimization technique used in the proposed solution is based on indirect pointers from the Block reference module to the index set. It is done on the block granularity, thus there is a piece of evidence, that a particular row contains at least one migrated row. Referenced index set covering such a block is then re-evaluated by removing migration. Consequently, there is no need to use logical data pointers – ROWlogs, which can form a performance bottleneck, whereas multiple indexes can be scanned in parallel, however, layer mapping logical pointers to physical infrastructure is only one. Moreover, only one transaction could operate a particular block by the Update, if the migration was present, the rest retrieval operations hung till the transaction ended.

By using the Master index extension, data migration reflection is done out of the main transaction and does not impact any running operations or transactions.

D. Optimization - number of tuples covered

The leaf layer of the Master index is formed by the BLOCKIDs – identifiers of the relevant data blocks. By loading a particular block into the memory Buffer cache, it is ensured (by removing migrated row reference), that at least one data row is present there for the evaluation. By using optimization criteria producing some data portion almost immediately followed by the full result set loading, it is inevitable to highlight the block content and probability, a particular row passing the criteria is present there. The finest assumption is based on the tuple number covered by the block. Thus, the block filled with 3 tuples should be preferred compared to the block showing 1 row only. From the Master index, however, block occupancy is not evident. Moreover, the utilization of the block itself is not a relevant composition, due to the various size of the row.

It may even happen that a single record requires more disc space than two analogous ones referring to the same table. Thus, from the table point of view, a block with more content should be preferred (higher loaded blocks), however, from the sequential access processing and record identification point of view, it is just the opposite, and blocks that contain more records should be preferred.

From the core B+tree index, by taking the leaf layer and extracting the block references from the ROWIDs, it is possible to get the number of rows part of each block. That prerequisite is done by marking block usage in the Master index for any existing table and each new consecutive data change is then reflected to the value tick, associated with any data block. Another solution is done by counting the number of pointers pointing to the Block reference module. Such a value can be then used as a part of the Master index key enhancement by preferring a higher block filling ratio.

E. Shrinking space

The limitation of the dynamic system full of undefined values and various attribute value sizes is reflected by no correlation between the tuple count inside the block and its fullness. As stated, several rows covered by each block can be obtained from the existing structures (mostly indexes) and there is no necessity to load that block using I/O and parse it. However, the fullness of the block cannot be obtained without ticking that block. Therefore, the crucial task is, how to obtain those values and how to deal with the missing calculations.

Shrinking space operation is not performed randomly, but must be preceded by a specific event, delimited commonly by the Delete operation or possibly by the Update operation lowering the storage demands. However, when executing a row removal operation, the particular row must be loaded to the memory and supervised by the transaction. And that´s just the point. If the block is present in the memory Buffer cache, there is no problem to calculate the block usage ratio. It is operated by the introduced background process – Block analyzer, which is the master process, extended by the Block analyzer slave(n).

Thus, in the proposed solution, some data blocks are excluded from the potential shrinking, but they are not properly described. On the other hand, suitable blocks are initially those, which have been freed by the Delete operation. And all of them are described by the usage in the Master index. In the second phase, each ticked block is automatically evaluated for utilization, if any change operation (Insert or Update) is performed. For the data retrieval, the calculation is enhanced by the introduced parameter Select_block_utilization set for the table. It defaults to False meaning, that Select statements are not extended by the block utilization calculation. Vice versa, if set to True, if the block has not been processed, yet, the particular block is analyzed and utilization is stored in the Master index for consecutive reference.

Later on, by the analysis, we introduced one extra parameter value called Exclude. In that case, if the block is not described, its reference is added to the buffer for the evaluation and utilization processing, however, it is done outside the main transaction. Consequently, the original Select statement is not influenced by the utilization calculation, which could strongly impact the performance, if the table is the huge and total amount of blocks to be recalculated is high.

```
ALTER TABLE <table_name>
   SET SELECT_BLOCK_UTILIZATION
                = {TRUE | FALSE | EXCLUDE};
```

V. Conclusions

This paper aims to propose multiple methods for optimizing the performance of the database system on the physical data layer. The overall performance of individual data manipulation operations depends on index usage, as well as physical infrastructure. Each table is delimited by the segment forming the structure and set of extents linked together. Logical database space is an extent, formed by a specific number of contiguous data blocks allocated for storing table or index data. Block itself is fixed-size and influenced by the data fragmentation, whereas the data rows are not the same size.

This paper proposes methods for data fragmentation and shrinking space by focusing on the empty blocks, which are not commonly deallocated, whereas it is too demanding and assumed, the data blocks will be used in the recent future. If sequential data block scanning is used during the data retrieval, even free blocks are memory loaded by increasing costs and processing time.

Another problem is related to index definition and migration. It occurs if the original block cannot fit the updated row. In that case, the particular original block stores only the pointer to the next block, where the row resides. During the index data processing, multiple blocks must be loaded, instead of loading just one block.

The proposed solutions are based on the Master index, which primarily reflects block orientation in a B+tree format. It is extended by the utilization and number of tuples covered, which contribute to block layer optimization.

During future research, provided techniques will be physically applied to the database definition, pointing to the limitations in a data-distributed manner. It is assumed, that the proposed solutions can be significantly beneficial in cloud environments, where the Buffer cache memory perspective can occupy a larger space. By using data partitioning, the Master index must be applied separately for each partition or fragment, however, data shrinking can be generally applied across the partitions. The intended solution is related to the pointer stitching.

Acknowledgment

This publication was realized with support of Operational Program Integrated Infrastructure 2014 - 2020 of the project: Intelligent operating and processing systems for UAVs, code ITMS 313011V422, co-financed by the European Regional Development Fund.

EUROPEAN UNION
European Regional Development Fund
OP Integrated Infrastructure 2014 – 2020

MINISTRY
OF TRANSPORT
AND CONSTRUCTION
OF THE SLOVAK REPUBLIC

It was partially supported by the Erasmus+ projects:

- Project number: 2022-1-SK01-KA220-HED-000089149, Project title: Including EVERyone in GREEN Data Analysis.
- Project number: 2020-1-HR01-KA226-HE-094713, Project title: Cloud cOmputing for Digital Education Innovation.
- Project number: 2021-1-SI01-KA220-HED-000032218, Project title: Better Employability for Everyone with APEX.

References

[1] D. Kuhn and T. Kyte, *Expert Oracle Database Architecture: Techniques and Solutions for High Performance and Productivity*. Apress, 2021.

[2] R. Greenwald, R. Stackowiak, and J. Stern, *Oracle Essentials: Oracle Database 12c*, O'Reilly Media, 2013.

[3] M. Kvet, J. Papán, "The Complexity of the Data Retrieval Process Using the Proposed Index Extension", IEEE Access, vol. 10, 2022.

[4] M. Kvet and K. Matiaško, "Analysis of current trends in relational database indexing", 2020 International Conference on Smart Systems and Technologies (SST), Croatia, 2020.

[5] M. Kvet, "Autonomous Temporal Transaction Database", 30th Conference of Open Innovations Association FRUCT, 2021.

[6] M. Malcher and D. Kuhn, *Pro Oracle Database 18c Administration: Manage and Safeguard Your Organization's Data*, Apress, 2019.

[7] J. Lewis, *Cost-Based Oracle Fundamentals*, Apress, 2005.

[8] S.Y.W. Su, S.J. Hyun and H.M. Chen, "Temporal association algebra: a mathematical foundation for processing object-oriented temporal databases", IEEE Transactions on Knowledge and Data Engineering, vol. 4, issue 3, 1998.

[9] D. Kuhn and T. Kyte, *Oracle Database Transactions and Locking Revealed: Building High Performance Through Concurrency*, Apress, 2020.

[10] T. Cunningham, "Sharing and Generating Privacy-Preserving Spatio-Temporal Data Using Real-World Knowledge", 23rd IEEE International Conference on Mobile Data Management, Cyprus, 2022.

[11] X.Yao, J. Li, Y. Tao and S. Ji, "Relational Database Query Optimization Strategy Based on Industrial Internet Situation Awareness System", 7th International Conference on Computer and Communication Systems (ICCCS), China, 2022.

[12] Oracle database documentation, Data Blocks, Extents, and Segments, Web: https://docs.oracle.com/cd/B19306_01/server.102/b14220/logical.htm

[13] Z. Liu, Z. Zheng, Y. Hou and B. Ji, "Towards Optimal Tradeoff Between Data Freshness and Update Cost in Information-update Systems", 2022 International Conference on Computer Communications and Networks (ICCCN), USA, 2022.

[14] W. Wang, Y. Jin, B. Cao, "An Efficient and Privacy-Preserving Range Query over Encrypted Cloud Data", 2022 19th Annual International Conference on Privacy, Security & Trust (PST), Canada, 2022

[15] S. Pendse, et al., "Oracle Database In-Memory on Active Data Guard: Real-time Analytics on a Standby Database", 2020 IEEE 36th International Conference on Data Engineering (ICDE), USA, 2020.

[16] J. Janáček and M. Kvet, "Shrinking fence search strategy for p-location problems", 2020 IEEE 20th International Symposium on Computational Intelligence and Informatics (CINTI), Hungary, 2020