Relational Data Index Consolidation

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

***Abstract*—A database index is an important object associated with the table to provide an additional layer for data access. By using the index, it is not necessary to scan the table block by block to locate relevant data rows. On the contrary, reflecting the usage of the tree structure it is possible to search for a record with logarithmic complexity. The limitation of the index structure is identified by the covering of the whole data set. If such a requirement cannot be ensured, the index method cannot be used, whereas some data portion does not need to be present in the result set, although all query conditions are passed. A typical example of such a problem is the NULL definition associated with the object or attribute itself. This paper deals with the existing solutions based on various transformation modules and proposes its architecture extending the index by node pointing to the undefined values.**

I. INTRODUCTION

Information technologies and decision making are based on provided data as the inputs. Quality, reliability, and availability form the core part of the processing. Without relevant data, no correct decision can be done, no robust system can be developed. To ensure the complexity of the system, it is inevitable to manage, process, and store data in a performance suitable storage. Database systems are now the most often used tool for data manipulation [4]. They provide the interface between the data and information system, respectively application itself. Data entering the system can originate from various systems, delimited by the quality. Similarly, sensor data can be routed by the non-fidelity network provided by the wireless technology, often operated by the ad-hoc networks. As a consequence, there can be several data streams with various relevance factors. Data can be delayed, non-trusted, or even do not fit the required criterion or range. Such activity must be recorded and evaluated, as well [8].

Relational database systems were created and firstly introduced in the 60ties of the 20th century. Their core elements are entities and relationships between them as the data diagram model. Data normalization ensures, that just the controlled redundancy can be present, either as the result of the entity relationships or created by the reports and outputs to ensure easier, faster, and more proper management [3].

The performance of the relational database system is enhanced by the data indexing, by which the relevant row can be identified and located easily and effectively [5], [9]. A database index is an optional structure, which can improve the performance of the database query [4]. However, if there are undefined values marked by the NULL symbol, a particular reference does not need to be in the index resulting in the necessity to scan the whole table, block by block sequentially [11], [16], [18]. This paper deals with the existing data structures with an emphasis on the system architecture in MySQL database system and Oracle. The main difference between these platforms is based on the way, they manage, process, and identify NULL value in the data tuple. As you can see reflecting the performance, Oracle does not manage NULLs inside the index, at all. Vice versa, particular undefined values are present in the MySQL system in some specific architecture.

The main contribution of this paper is our own solution managing undefined values in the index by using additional node. In comparison with other transformation and locator modules, which change the undefined values to the logical or physical perspective. Thanks to that, relevant data can be obtained regardless of the number of NULLs, with no additional costs caused by transformations. Direct pointer to the data brings extra power during the data retrieval and block parsing process. Moreover, the proposed models provide the benefit, if the database statistics are not up-to-date, as well.

Section 2 deals with the database architecture and loading process definition from the physical storage to the instance memory. Section 3 deals with the index and selectivity supervised by the database optimizer. The impact of clustering is specified in section 4. The author solution is proposed in section 5, supported by the evaluation perspective managed in section 6.

II. DATABASE ARCHITECTURE

The database server consists of two entities – the instance and the database. The instance covers the memory structures and background processes supervising the second entity – database, which is reflected by the physical storage – files on the disk. The interconnection between the instance and database is defined by the system tables, tablespace, and controlfile. Thanks to that, complete logical data abstraction can be ensured [3], [12], [14].

The database instance is transient, holding data in the RAM of the server, supervised and operated by the CPUs. Thus, by shutting down the instance, particular data are removed. Vice versa, database stores data indefinitely, therefore, background processes must ensure proper management with no data loss.

ISSN 2305-7254

Such a requirement is operated by the transaction logs, by which the database can be recovered after the failure. Users cannot directly access and manipulate the database itself, individual requests are managed by the background processes transferring relevant data blocks from the database into the instance memory for the consecutive processing and result set building.

The processes making the instance are present during the whole lifecycle of the instance. They are mostly self-administering. Memory can be divided into two categories – System Global Area (SGA) and Program Global Area (PGA). SGA is shared among the whole instance covering the data changes, parse activity, etc. In contrast, PGA is private to the session holding the current environment, parameters, or local data [1], [7].

From the physical perspective, data files, online redo log files, and controlfile defining the structure can be highlighted.

Data to be operated are physically located in the data files covered by the tablespaces characterizing their properties, approaches, etc. If any data tuple is to be processed by any operation, it must be loaded into the SGA memory structure – Buffer cache. The granularity of the transfer between the instance and database is the block itself, which can contain multiple data rows [6].

Fig. 1 shows the single-instance database architecture. As you can see in the graphical form, the user is represented by the user process connected to the server process on the server-side supervised by the instance. Background processes operate instance and database and manage data transfer. Therefore, each data point must be transferred into the instance to become available to the server process. Thanks to that, optimization pointing to the data manipulation on the physical storage and instance is a crucial part influencing the whole system performance. The ideal solution is to locate all data in the memory for direct access. Such requirement is, however, mostly infeasible, due to the need for huge memory space, but determined by the system architecture, as well [3], [12]. The requirement of the relational transaction is the durability, thus committed data can never be lost. The data stored in the memory are only temporary and they would be lost after the failure. Therefore, it would be necessary to design a recovery system pattern, that is already partially in place, defined by the transaction logs, but this mechanism is too time and technically demanding, whereas it is necessary to build the memory to the state before the failure [11], [14].

It is defined either by the data locating, as well as data transfer. The aim is straightforward – to lower the size demands - the amount of the transferred data encapsulated by the optimization of the data structure. A useful robust data tuple locator is the index structure.

III. INDEX

An index is an optionally created database object associated with the table used primarily to increase query performance. The main purpose of the insed to simplify and accelerate the process of data location and retrieval. Besides, it can be used to ensure integrity rules. It can be created either implicitly (by the definition of the primary key, respectively unique constraint) or as the result of the explicit user definition to ensure scalability and performance, to create an environment for better optimizer decisions in the performance manner.

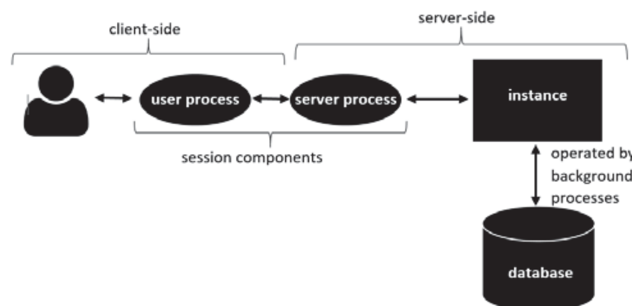

Fig. 1. Server-client, session

In individual database systems, various index structures and access principles can be identified, based on the B-tree [5], [12], Bitmap [12], [19] or Hash architecture [19], with various enhancements – unique key, reverse key, compression, function-based, virtuality, invisibility, etc [8].

B+tree is the default and significantly widespread relational database index structure. It is created automatically for the primary key or unique set of attributes. It consists of the balanced tree delimited by the root node, internal nodes, and leaf nodes with the pointers to the data in the database themselves. ROWID locator is a 10-byte pseudo column returning the address of the row – data object number (1-32 bits), data file, in which the row resides (33-44 bits), data block (45-64 bits), and position of the row inside the block (65-80). Such value provides the fastest and easiest way to locate data directly by the starting position of the tuple. For the non-clustered environment, ROWID specification is always unique [8], [9].

The limitation of the ROWID value definition is precision and correctness. There is just the one-direction pointer from the index to the database, meaning, that if the data are later relocated (resulting in changing ROWID values), references would locate incorrect data blocks, where the required data are not present. To ensure the reliability, if the whole object is moved, all ROWID values are denoted as non-relevant and the index is marked as invalid. In such a situation, the index cannot be used at all, forcing the user to rebuild it manually.

Data tuple can, however, change its position in different granularity, as well as creating a migrated row. Namely, if the data row is shifted from one data block to another, index structures are not notified, but from the original data block, a new pointer to block holding particular data is added. The complexity of the reallocation problem and migrated row definition can be found in [9]. Notice, that the rebalancing methods can significantly influence performance, whereas non-relevant data blocks are loaded into the memory and parsed [9].

Database optimizer is an autonomous process, which is responsible for detecting and selecting the best suitable method for data access and loading. It evaluates the available index set, as well as statistics collected for each table [3], [4]. Thanks to that, the relevant data access method can be selected, based on the heuristical approaches with very correct result outputs.

One of the main aspects influencing the methods to access the data is just the selectivity [4]. Selectivity represents the degree of uniqueness of the data values contained within an index. Selectivity (S) of the index (I) is the number of distinct values (d) contained in the data set, divided by the total number of records (n):

$$S(I) = d / n$$

Selectivity value ranges from 0 to 1. With the rise of the selectivity value, index usage is more and more preferred.

The limitation of the selectivity calculation and index usage is just the NULL values, which cannot be compared to each other, sorted or positioned with emphasis on real existing values. Portability of the NULL into the index definition is therefore strictly limited resulting in complete non-availability in the Oracle database. Vice versa, in the MySQL database, undefined values can be part of the index, if suitable model architecture is used.

IV. CLUSTERING

Physical data modeling and representation play an important role in dealing with index and global access. MySQL database system provides multiple formats of the data organization. Reflecting on the history and evolution, MyISAM structure manages data in a non-sorted manner by providing a pointer to the data in the index. Nowadays, this architecture is mostly used for reading operation preference and provides table-level locking. On the opposite side, the InnoDB solution can be identified as offering ACID transaction categories, multi-versioning, and complex relational integrity support. In comparison with MyISAM, InnoDB is characterized by the clustered index. Thus, each table must contain a unique non-nullable primary key, records are stored in data pages according to the order of this identifier (clustering key). If the clustering key is not specified explicitly, InnoDB creates its own internal. All other indexes are categorized as secondary [10], [15].

Although the searching on the clustering (primary) key is remarkably fast and performance effective, whereas the data row itself is directly accessible without data lookup operation necessity, there is a strict requirement about the data nullity. No undefined value can be present by the definition. Therefore, the user must remove undefined values, either by removing the data row completely or by transferring NULL into another valid data value. As the consequence, data retrieval must be aware of such a definition to provide user consistent and correct data result set. Unfortunately, most of the transformation processes must be maintained by the developer explicitly.

3.1 Execution plan dealing with NULLS

- **ORACLE**

The importance of the undefined value management is claimed in the following description. In DBS Oracle, NULL values are not present in the index, at all. Reflecting the execution, if the condition is based on the specified NOT NULL value, the optimizer can select access using the index. In the following case, there is the following condition:

where ID=12

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	2
TABLE ACCESS	EXEC_TAB	BY INDEX ...	1	2
INDEX	EXEC_IND	RANGE SCAN	1	1
Access Predicates				
ID=12				

Fig. 2. Execution plan – valid, NOT NULL condition

However, if the condition is based on the NULL, execution is routed into the sequential block scanning – Table Access Full:

where ID IS NULL

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			4	104
TABLE ACCESS	EXEC_TAB	FULL	4	104
Filter Predicates				
ID IS NULL				

Fig. 3. Execution plan – the NULL condition

- **MySQL – MyISAM**

MyISAM database architecture of the MySQL database system can hold undefined NULL values directly inside the index (fig. 4), however, there is just table-level locking shifting the solution just for reading lookups.

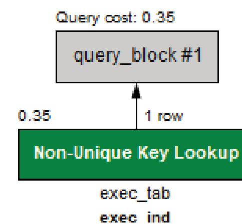

Fig. 4. Execution plan in MyISAM environment

- **MySQL – InnoDB**

InnoDB architecture cannot deal with undefined values, at all, just as the result of the function baed clustering index.

3.2 Automatic data transformation and management

Existing solutions dealing with undefined values inside the index are based on the transformation, by mapping NULL value into the specific value. It can be done either by the trigger definition of the dependency model [10], mapper functions [9], or function-based indexes [4]. The limitation of the function-based index is only one-direction – from the database to the output, thus it is necessary to adjust the query

to reference a particular transformation function, otherwise, the whole table is scanned. Moreover, after the processing, the transformed value has to be replaced by the original representation to provide the same results. In [17], NULL value estimation methods are proposed to deal with the granularity for incomplete systems. Dependencies and interconnection factors are discussed in [10]. Partially undefined conditions caused by nullity management are discussed in [13]. Velocity factors and scalability aspects are delimited in [15].

The following script shows the principle of the function-based index usage. Input data are transformed for the index usage, however, on the physical level, NULL pointer with no size demands is present. Attribute set (att_set), which can hold undefined value is transformed using the function null_transform for the index processing. On the database level, nullity is present.

```
Create index ind_name
    on table_name(null_transform(att_set));
```

The next script shows a similar solution based on the trigger. In that case, it is evident, that there are additional size demands, whereas physical transformation is present. Moreover, the user always needs to reference the transformed value, original NULL is not present at all. Thus, it can cause problems with incorrect evaluation, if the user, respectively developer is not aware of such transformation, which is done automatically, but the reverse representation - from the database to the user must be handled manually. Architecture is in fig. 5.

```
Create or replace trigger trig_name
  Before insert or update on table_name
    For each row
 Begin
    if att_set IS NULL
       then att_set:=null_transform(att_set);
    end if;
End;
/
```

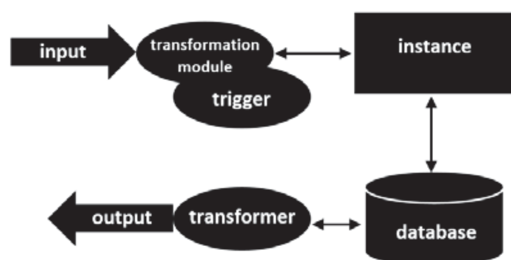

Fig. 5. Transformation modules

Mapper functions introduced in [9] offers a bi-directional solution. Fig. 6 shows the architecture. In comparison with the architecture introduced in fig. 5, the mapper is located between the database and instance. Physically, the NULL value with no size demands is present. During the loading of data into the memory, transformation is done, reflecting the indexing, in which the transformed function is present. On the server-side, before producing results set into the client site, the transformed value is replaced by the original NULL notation. Thanks to that, a function-based index reflecting the transformation can be used, however, from the client perspective, no data change is present. The limitation of the mapper module is just the transformation value representation. Users must select an appropriate value to ensure, that particular value cannot occur in real data, otherwise the data interpretation would be corrupted. From the performance perspective, there are the additional costs caused by the bi-directional mapping. Moreover, the size of the block in the database and mapped into the memory is not the same. As the consequence, the transformed block does not need to fit the Buffer cache block grid resulting in the necessity to allocate additional block in the memory, which must be, however, similarly, merged during the database write process [9]. Principles of the mapping with emphasis on the implementation particularity can be found in [9].

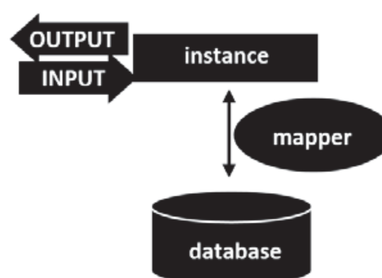

Fig. 6. Mapper

V. OWN SOLUTION

Fig. 6 shows the architectural perspective of the mapper, which is present either during the data loading from the physical database into the memory buffer cache, as well as during the result set composition. As already described, additional size demands are necessary to be present in the memory buffer cache, whereas NULL value is transformed to the physical value, which is memory located. In this paper, we have implemented the existing mapper solution and propose the following enhancements:

MODEL 1 significantly limits the necessity for block splitting. Each data block consists of the header and data themselves in the standard environment. In this perspective, the data part is divided into two groups, one small space is used for the mapping. Thus, the total available size of the block is lowered, on the other hand, there is no necessity to use the block rebalancing, if the particular transformation does not fit the original block. Based on the experiments, the block capacity is lowered to 90% of the original size [9]. Model 1 represents the existing solution; the rest models are part of this paper contribution.

MODEL 2 improves the technique dynamically by analyzing the table structure, NULL references, and data dictionary. Based on the current table statistics, the system

estimates the ratio for holding NULL value transformation during the loading process to remove, respectively significantly limit the necessity for block splitting by the mapping.

MODEL 3 extends the database statistics by forcing Manageability Monitor (MMON) background process to ensure an up-to-date image of the number of nullity presence for each attribute. Thus, if any data change is present, such activity fires the MMON process to reevaluate the NULL pointer reference counter. Thanks to that, the original B+tree index based on the attribute set can be immediately used, if the counter is set to the zero value for the particular indexed set. By the management, it is ensured, that no other data tuple can be present in the system, but not indexed due to the NULL representation. On the other hand, reflecting section 6, it is evident, that such activity has additional demands on the Data Manipulation Language (DML) operations – Insert, Update, and Delete.

Proposed **MODEL 4** reflects different optimization techniques. Transformation of the NULL values is not based on the physical change, just the logical perspective is used. Thanks to that, it is not necessary to calculate and evaluate the capacity of the block, whereas, during the transformation, the size is not updated at all. Instead of direct data value transformation, the only new pointer is used. The NULL pointer object reference is stored in the instance memory during the whole life cycle of the system (after the startup, particular reference memory object is mounted automatically by the Data Definition (DDL) trigger and released during the instance closing). Thus, the size of the block remains the same, during the transformation NULL value is pointed to the memory object. Such reference is part of the index for the consecutive processing and evaluation. The core proposed solution is covered by **MODEL 5**.

In comparison with already described perspectives, this implementation does not use the mapping at all. The NULL definition is present either in the physical database, as well as after the loading, which is straightforward, operated by the direct loading without any additional formatting. Thanks to that, the impact of the mapping module on the performance is removed. However, how to ensure the reached performance? The answer is based on extending the index approach by using NULL elements, which are present there. Thanks to that, statistics can remain using the original form defining just the overview of the data structures and values resolution and estimation. The database table index in model 5 extends the B+tree definition by using an external pattern physically implemented by the nested structure holding NULL values. A new node dealing with NULL values is present there. It can be located either in the leftmost or rightmost part of the index. The physical construction depends on the selection of the parameter NULL_position, which can hold NULLS_FIRST or NULLS_LAST values or even NONE. If NONE property is selected, undefined values are not present in the index at all, forcing the optimizer to select Table Access Full scanning. The default value for the parameter NULL_position depends on on the ascending, or descending order of the values, respectively:

```
Create index ind_name
    on table_name(att_set)
        [ NULL_position= { NULLS_FIRST,
                           NULLS_LAST, NONE } ];
```

If a new undefined (NULL) value is to be indexed, a particular node characterizing pointers to undefined values is located in the first phase. As mentioned, it is implemented by the nested table, thus in the second step, it is extended, the new value is added to the last position:

```
Null_pointers.extend();
Null_pointers(null_pointers.last):=new_object;
```

As evident, NULL values are not sorted in any specific order, just placed in the common nested table. Logically, the newer data portion is placed on the right part of the nested table, therefore values are time delimited expressing transaction flow. Thus, data are ordered in the date manner.

MODEL 6 extends Model 5 by using a position reflected in the physical storage. Thanks to that, if the query consists of the standard data tuple, as well as partially undefined values, the database optimizer can locate blocks with multiple data tuples present inside. Thus, if the optimization mode is to provide any data portion as soon as possible, provided model can benefit. Model 5 is optimized to get all data as soon as possible. In contrast, the aim of the Model 6 is to provide at least one data row almost immediately.

Models 5 and 6 are bi-directional by storing and evaluating data as they were provided in the input with no transformation, nor mapping necessity.

VI. RESULTS

Performance characteristics have been obtained by using the Oracle 19c database system based on the relational platform. For the evaluation, a table containing 10 attributes originated from the sensors were used, delimited by the composite primary key consisting of two attributes. The table contained 1 million rows, 10% of them contained undefined values. Two index structures were defined, one implicit covered by the primary key, the second index extends the primary key by covering attribute, which can hold NULL value. The select statement was evaluated covering all attribute values. The condition limited the amount of data to 10% of the whole data set. Data set management and structure was introduced in [9].

The reference model for the evaluation and comparison was **MODEL 0** defined by the two *B+tree* indexes (primary key and user-defined) with no undefined values special support. Thus, in the results, this model reaches 100% of the processing costs and size demands.

To evaluate the performance benefits, size demands, and processing time of the Select statement is analyzed in this paper. When dealing with the size of the whole structure, Model 3 and Model 4 does not require any additional storage capacity. In contrast to Model 1, where the smaller capacity of

the block is present due to the NULL value mapping into the memory. The size of the data part of the block is lowered to 90%, the header remains the same. Size demands are 111,3% in comparison with Model 0. Model 2 requires only 7.9% of data storage extension, whereas the block capacity is calculated dynamically. Proposed Models 5 and 6 use the different strategies and store the NULL value pointers directly in the index nodes. As shown, additional demands are 2,9% for Model 5 and 2,8% for Model 6, respectively. Notice, that by the nested table inside the additional NULL pointer node optimization, the amount of 10% of NULL values requires just 2,9% of the storage extension. Fig. 7 shows the results of the size demands analysis graphically.

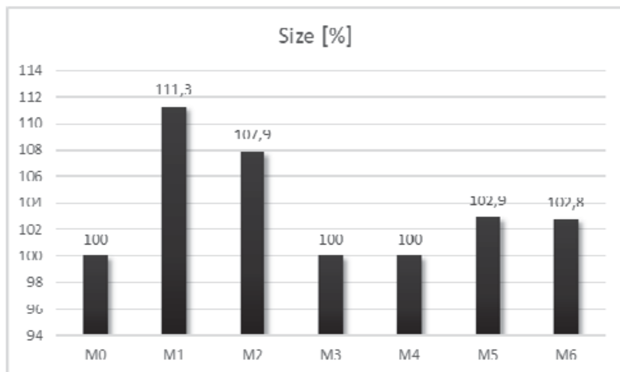

Fig. 7. Results - Size demands (%)

The second keystone reflecting the performance strategy is delimited by the Select statement processing time. Based on the defined environment and data set, fig. 8 shows the reached results. Similarly, the reference is Model 0 with no support for dealing with the NULL values, either physically, as well as in the index structure. Thus, the index access method cannot be used, at all, instead, TAF scanning is performed. In that case, therefore, 100% of data blocks associated with the table must be loaded into the memory for the consecutive parsing and evaluation. Model 1 lowers the processing time to 34,8%, by using mapping during the loading. Dynamic data block size in Model 2 brings additional benefit, whereas the total amount of the blocks is lowered. In comparison with Model 1, the proposed Model 2 lowers the costs by 10,6%. Model 3 forces the system to store up-to-date statistics reflecting the current situation of the NULL attribute value amount. Thanks to that, the processing time is lowered to 27,5% with regards to Model 0. Comparing the results with Model 2, it brings a 10,6% benefit. Similarly to Model 3, defined Model 4 does not need the physical storage extension, at all. Just the pointer to the specific memory object initialized during the instance startup is present. Thanks to that, undefined values can be located through this element. Although it is not part of the index and stored physically, the processing time of the Model 4 requires only 25,4% in comparison with Model 0. Vice versa, dealing with the Model 3, memory pointer mapping lowers the processing time by 7,6%.

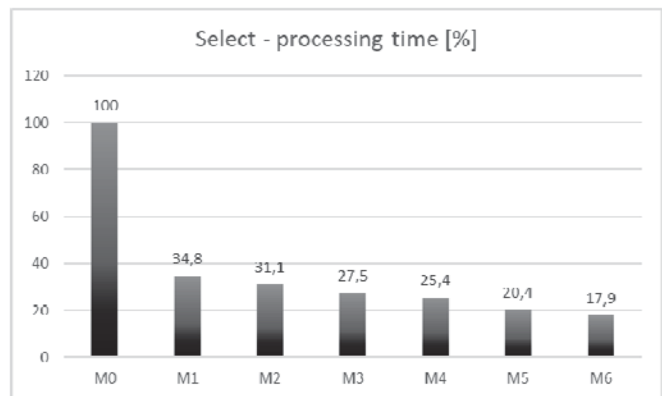

Fig. 8. Results – Select statement processing time (%)

Models 5 and 6 locate NULL value positions directly in the index, thus there is no transformation necessity, however, the size of the index has to be increased by adding a new node, either in the leftmost or rightmost part of the index, based on the definition. Thanks to that, if the clustered key for the NULL value specification is time element expressed by the transaction (Model 5), the processing time is lowered to the 20,4%. ROWID value as the key used in Model 6 expresses just 17,9% of the processing time. The main benefit, in comparison with Model 5 is based on the ability to directly access a record on physical storage. If there are multiple records in a particular block, the benefit can be even greater, because the block will be processed and parsed completely, but only once, thus the amount of the data transported by the block granularity from the database to the instance memory via the I / O interface is significantly lowered.

VII. CONCLUSIONS

Nowadays, it is evident, that the number of data is significantly rising over time. With the growth, another aspect dealing with the performance of the system should be highlighted. Data are produced from various sources with various quality, reliability, and availability. Many times, undefined values caused by the delay, connection outage, or improper communication channel throughputs are present. These values and states cannot be ignored and should be stored in a specific manner.

Index structures are mostly based on the B+tree and provide a relevant solution for data access. In many database systems, however, NULL values are not present in the index. A typical example is formed by the most complex system Oracle. To ensure performance and robustness, particular non-reliable or undefined values marked by the NULL symbol must be either transformed to another value available for the indexing or the principle of the index must be changed to provide space for the NULL inside it.

In this paper, several architectures and models are introduced and performance is analyzed. They can be divided into two categories based on the implementation. The first

category deals with the physical transformation. A NULL value can be notified either during the loading process, in that case, it is triggered or the physical transformation can be done during the loading from the database into the memory buffer cache or vice versa. Such solutions are robust and offer significant performance benefits, however, there is still an option to improve the performance. Therefore, the second category was introduced, which is based on the index storage extension by the node holding NULL values in the nested table. Based on the experiments and reached results, in comparison with the first category, the performance of the data retrieval can be improved up to 50%, just with only minimal additional costs at the database storage level (up to 3%).

Research on the undefined value management forms an important role to ensure the performance of the database system. In the future environment, the nested table module dealing with pointers to the non-valid data portion would be experimented, based on either clustering key structure and significance. Moreover, we would like to revalidate NULL with the temporal evolution monitoring changes, to predict the consistent value to estimate undefinition.

ACKNOWLEDGMENT

This publication was realized with the support of the Operational Programme Integrated Infrastructure in frame of the project: Intelligent systems for UAV real-time operation and data processing, code ITMS2014+: 313011V422 and co-financed by the European Regional Development Found.

REFERENCES

[1] Abdalla, H. I.: A synchronized design technique for efficient data distribution, Computers in Human Behavior, Volume 30, 2014, pp. 427-435

[2] Behounek, L., Novák, V.: Towards Fuzzy Patrial Logic. In 2015 IEEE Internal Symposium on Multiple-Valued Logic, 2015.

[3] Bryla, B.: Oracle Database 12c The Complete Reference, Oracle Press, 2013, ISBN – 978-0071801751

[4] Burleson, D. K.: Oracle High-Performance SQL Tuning, Oracle Press, 2001, ISBN - 9780072190588

[5] Delplanque, J., Etien, A., Anquetil, N., Auverlot, O.: Relational database schema evolution: An industrial case study, IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Spain, 2018, pp. 635-644

[6] Eisa, I., Salem, R., Abdelkader, H.: A fragmentation algorithm for storage management in cloud database environment, Proceedings of ICCES 2017 12th International Conference on Computer Engineering and Systems, Egypt, 2018

[7] Ivanova, E., Sokolinsky, L. B.: Join decomposition based on fragmented column indices, Lobachevskii Journal of Mathematics, Volume 37, Issue 3, 2016

[8] Kvet, M., Matiaško, K.: Concept of dynamic index management in temporal approach using intelligent transport systems. In: Recent advances in information systems and technologies: Volume I. - Cham: Springer, 2017. - ISBN 978-3-319-56534-7. - S. 549-560.- (Advances in intelligent systems and computing, Vol. 569. - ISSN 2194-5357).

[9] Kvet, M.: Managing, locating and evaluating undefined values in relational databases. 2020

[10] Lien, Y.: Multivalued Dependencies With Null Values In Relational Data Bases. In Fifth International Conference on Very Large Data Base, 1979.

[11] Mirza, G.: Null Value Conflict: Formal Definition and Resolution, 13th International Conference on Frontiers of Information Technology (FIT), 2015.

[12] Moreira, J., Duarte, J., Dias, P.: Modeling and representing real-world spatio-temporal data in databases, Leibniz International Proceedings in Informatics, LIPIcs, Volume 142, 2019

[13] Shiryaev, V., Klepach, D., Romanova, A.: Implementation of the Algorithm for Estimating the State Vector of a Dynamic System in Undefined Conditions. In 27th Saint Peterburg International Conference on Integrated Navigation Systems, 2020.

[14] Smolinski, M.: Impact of storage space configuration on transaction processing performance for relational database in PostgreSQL, 14th International Conference on Beyond Databases, Architectures and Structures, BDAS, 2018

[15] Ochs, A. R., et al.: Databases to Efficiently Manage Medium Sized, Low Velocity, Multidimensional Data in Tissue Engineering, Journal of visualized experiments JoVE, Issue 153, 2019

[16] Steingartner, W., Eged, J., Radakovic, D., Novitzka, V.: Some innovations of teaching the course on Data structures and algorithms. In 15th International Scientific Conference on Informatics, 2019.

[17] Xia, Z., Hanyan, Y, Mingzhu, X.: Null Value Estimation Method Based on Information Granularity for Incomplete Information System. In Third International Symposium on Intelligent Information Technology Application, 2009.

[18] Qi, C.: Research on null-value estimation algorithm based on predicted value. In IEEE 5th International Conference on Software Engineering and Service Science, 2014.

[19] Vinayakumar, R. Soman, K., Menon, P.: DB-Learn: Studying Relational Algebra Concepts by Snapping Blocks, International Conference on Computing, Communication and Networking Technologies, ICCCNT 2018, India, 2018