Relational Pre-indexing Layer Supervised by the DB_index_consolidator Background Process

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

Abstract—**The performance of the database system significantly influences the whole ecosystem reflected by the applications, information systems, and analytical tools. Data retrieval and access form crucial operations. Data indexing can provide additional power for the tuple location, identification, and proper management. By using such a technique, Select statement can benefit. Data integrity and consistency can be ensured, as well. On the other hand, adding new indexes brings additional demands on the system and change operations, whereas activity performed on the data must be mirrored into the indexes to ensure correctness and recency. This paper deals with the index definition from the data manipulation operation perspective. New modules monitoring changes are added. As a result, the amount of the indexes can be significantly increased, but with minimal impact on the Insert, Update, and Delete statements.**

I. Introduction

Data amount to be processed and managed has changed significantly over the decades. Users, developers, and database administrators can feel the significant demands and must react properly to ensure the system usability, either by the hardware, as well as software perspectives. Relational databases evolve rapidly over time. In the first phases, whereas the disk storage capacity was extremely expensive, just a small amount of data was able to be managed. The produced data amount from the other systems was really low, as well. With the advent of cheapness and availability of the disks delimited by the huge capacity, there is no problem to maintain data evolving over time, to store large objects, and monitoring data directly in the database. Hardware brings currently minimal limitations supervised by the technological overhead. On the other perspective, we can feel the strong demand for performance. If the amount of data is rising, it is inevitable to change the access rules, principles, and plans to improve the performance of the data retrieval, to locate data effectively. Such activity is mostly done by the index database objects, which cover the pointers to the relevant data based on the main data attributes or functions respectively.

The aim of this paper is to provide overview of the relational system indexing principles with emphasis on the proposed solution based on the pre-indexing layer limiting the necessity to balance the index in the main transaction. Background process dealing with index consolidation is proposed to ensure performance. This paper is structured as follows: section 2 deals with the Database architecture with regards to the block granularity, which is important during the data loading from the database into the memory instance for the evalution. Section 3 deals with the index definition, extended in section 4 managing performance. Section 5 proposes own solution. Performance evaluation is in section 6.

II. Relational database and processes

Looking at the physical architecture of the database system, two independent, but interconnected parts can be identified. A database instance is a logical unit represented by the memory structures and processes managing either instance, as well as data themselves. Background processes are responsible for memory management, allocation, and space reallocation, but they provide the whole management regarding user connection and database supervision. During the database instance startup, System Monitor, and Memory Manager background processes are started in the first phases, even during the no-mount process to prepare memory structures. In the mounting step of the startup process, background processes locate the pointers to the physical storage files and map them to be accessible from the instance. These pointers are stored in the physical storage, outside the database, in the controlfile. The last step is represented by the instance and database (as the second part of the database system architecture) availability. Thus, these parts are independent, after the server shutting down, only database data are available, however, they cannot be parsed and processed at all, without the interoperability of the database instance.

Data are physically stored in the files, in which data are formed in the block manner. The size of the block for the default management is fixed (8kB), defined during the database definition, and cannot be changed later. Available block size definitions are 2kB up to 32kB in some architectures [3]. The impact of the block definition on the performance characteristics can be found in [5], [17]. The block structure is significant in the whole architecture and management, whereas a similar block structure must be present in the memory to serve the block loading process. The memory Buffer cache is a structure operated by the Database Writer, Process Monitor, Memory Manager background processes [4]. It aims to provide space for the block loading from the physical database for the consecutive processing (e.g. parsing), as well as new or updated data are stored there,

before the physical transformation into the database. Therefore, the loading process is one of the most demanding aspects of influencing global performance. The aim is therefore straightforward, to limit the amount of the blocks to be transferred. In [15], a complex analysis of the data block structure with emphasis on the data migration can be identified. In that case, data are not in the block, where expected. It is mostly caused by the increase of the data row size after the update operation. The newly updated data tuple does not fit the original data block and must be shifted into a new one. Another problem is caused by data fragmentation [7] [8]. In that case, individual blocks have free space and the total number of blocks is too high in comparison with real usage. Fragmentation management can be supported either by the robustness of the index [9] or by the online defragmentation methods [7] influencing the data blocks themselves, as well as the indexes [18] supervised by statistics [10]. Data block compression is discussed in [19]. Partitioning is another technique to limit the data amount in one place. By using this technique, data access can be parallelized across the partitions, which can be managed either locally [11] or in the global perspective [11] [20]. The disadvantage and limitation of such a technique is just the bottleneck expressed by the loading process into the memory if partitions are placed physically in one common disk operated by just one interface. Data distribution can be relevant to limit such negatives by splitting the processing into multiple nodes operated by the separate memory structures [1]. In this context, there can be a different problem identified, just as a result of the node connectivity loss. In such a case, individual data tuples should be stored in a replicate style to avoid data loss and processing availability [1], [2], [6] secured by the global reliability of the retrieved data [21].

From the above perspective, data are operated in a split manner, either by partitioning in the local system or distributed environment. On each node, however, an index must be present to ensure the efficiency of the processing supported by the index access techniques. In principle, if the suitable index is not present for the evaluation, the database optimizer is forced to use sequential scanning of the data blocks (Table Access Full (TAF) method). When using an index access path, several techniques can be used reflecting the environment, properties, and structure of the query. The main factor influencing the performance is expressed by the Where condition of the Select statement. If the query results in getting no more than one row by the specification of the unique condition (e.g. condition based on the primary key), the Index Unique scan technique can be used. By traversing the index, if the particular node is searched successfully, evaluation can end immediately, followed by the optional Rowid Scan method, by which the required attribute values, which are not present in the index, can be obtained. In that case, the whole block containing the row is loaded into the memory Buffer cache. Vice versa, if the requirement of the unique definition cannot be ensured, Index Range method is used. In that case, the first suitable passing Where conditions are located by the index. Afterward, a linked list on the leaf layer is used to locate the next data row based on the index. The processor checks, whether the conditions are passed, as well. If so, a particular Rowid value is extracted and processing continues with the next row definition, until the first row, which does not reply to the Where clause is present. As the result, a list of Rowid values is created. These values are sequentially used as the input for the Rowid scan method to provide additional values, which are not directly stored in the index [4], [6].

The above methods are the best if the Select statement conditions are in a suitable order reflecting the attributes inside the index. In the ideal case, all required data (delimited by the Select clause) are present, thus the result set can be directly composed. If not, additional attribute values or function methods output is calculated and obtained. It is not possible, nor efficient to define all suitable indexes reflecting the possible activities. Therefore, another category of index methods has been created. By using such techniques, a list of the attributes in the index does not fit the Select statement, however, database optimizer assumes, that the estimation of the time and performance would benefit if the whole index is scanned by locating data. Generally, the database manager assumes, that the index scanning is easier and faster in comparison with sequential scanning of the whole table [4] [14]. The principle is mostly based on the number of blocks and the structure of the index in comparison with the whole table. The index is almost always smaller than the table itself. However, there is a strict requirement, that the index must hold a reference to all data rows. Otherwise, such an index will be refused from the definition. A typical example of such an index, which holds not the whole data table set, is delimited by the NULL value definition. B+tree, as the default index structure used in relational database systems, does not manage undefined (NULL) values, at all. In Oracle, it is always true, as the result of the impossibility of NULL value comparison in mathematical theory.

In other database systems, NULL management depends on data storage architecture. In MyISAM of the MySQL, data are not sorted, the index provides a pointer to the data. A non-clustered version ensures, that the record on the disk is in an unsorted manner. It can hold NULL values directly in the index definition. In comparison with InnoDB, which uses the clustered index. Each table in that architecture must contain a unique non-nullable primary key, by which the physical location is defined. Hash storage engine is another architecture, which, in principle, cannot manage undefined data values, whereas it uses mathematical operations to calculate the bucket, in which the data row resides. By using a function-based index based on the hashing as the core for the extended Hash storage engine, before the processing, the undefined value is replaced by the real present value, thus it can be located in the index, as well [12]. The problem of undefined value management is complexly specified and solved in [11], [12], [13]. Undefined values shifted into the conditions are handled in [16].

Two full scanning methods can be identified – Index full scan and index fast full scan. An index fast full scan reads the entire index in an unsorted manner as it exists in the database disk storage. In principle, this method uses index instead of the table by using multi-block IO reading all leaf blocks. It ignores the branch and root blocks and just processes the data on the leaf layer. Index full scan uses the fact, that the data are sorted in the index. It starts in the root block navigating to the left-hand side of the index reaching the leaf block. It reads across the entire bottom of the index – a block at a time in a sorted principle [4].

III. INDEX STRUCTURE DEFINITION

As already defined, the B+tree index structure is the most often used data structure used in the relational theory. It benefits the fact, that the data are sorted in the leaf layer, as well as it does not degrade the performance with the increase of the data. It consists of the collection of one or more data pages, called nodes. By searching in that structure, the DB server starts at the root, which contains a set of *n* key values in sorted order. Each key contains not only values of the key itself, but it also has a pointer to the node, that contains the keyless or equal to its key-value, not no greater than the key value of the preceding key. The keys point to the data page on which records containing the value can be found. The pages, on which key values – index records – can be found, are called leaf nodes. Similarly, index data pages containing these index nodes that do not contain index records, but only pointers to where the index records are located, are called non-leaf. The significant advantage of the B+tree index, in comparison with the original B-tree index, is based on the fact, that the leaf nodes are sorted and interconnected allowing the system to perform In, Between, leftmost Like, or comparison operations (<, >, <=, >=, etc.) directly.

Other index types include bitmap index (used mostly in the data warehouse environment), R-tree used in spatial systems, Fulltext index, etc.

In the past, the often preferred index type was based on the hash function mapping key into a value pair. In that case, management is supervised by the hash function as the method, by which a supplied search key(k) can be mapped into a distinct set of buckets (n), where the values paired with the hash key are stored:

$$h(k) = \{1, n\}$$

The Hash index is not available in the database system Oracle and can be used in MySQL for InnoDB and Memory architecture.

IV. PERFORMANCE REFLECTING INDEXES

Current trends in the relational theory supervised by the cheap hardware, cloud environment, and complex systems influence the performance, as well. Preference to locate data directly in the instance memory can be felt. Direct and really fast data access can be reached, unfortunately, mostly just in a theoretical manner. Although cloud technology is currently widespread, it is not possible to allocate unlimited instance memory. Apart from the price of the solution itself, with the increasing amount of data, the demand for memory, hardware technologies, and the whole ecosystem would also increase proportionally. Whereas the size of the database is now in the gigabytes, terra bytes, and even petta bytes, it is impossible to allocate such memory structure. Moreover, memory is the only RAM of the instance, with no reflection on physical storage. Thus, after any problem, data will be destroyed with no image in the physical structure. Imaging memory into the database would not provide sufficient power [2], [6].

Mean time to recovery is a significant parameter influencing data management in memory and physical storage. After the failure, it is necessary to restore all data as they existed before the collapse. It requires for all transactions, which are not applied physically in the database, to reexecute the processing tasks based on the Redo logs one more time. Vice versa, transactions, which were not approved, must be rollbacked and any change in the database block must be removed by applying UNDO change vectors. Mean time to recovery limits the time to recreate the database system, as it existed before the corruption. As a consequence, a huge memory amount does not bring additional benefit, whereas each change must be physically stored and represented in the database, otherwise, it would last too much time to consolidate the system after the failure.

When dealing with the Log files - Undo and Redo structures, it must be emphasized, that query is always associated with the time of its execution – begin time. Therefore, the database, nor the instance needs to hold data reflecting the execution time, just the currently valid data are present. Thus, current data images are obtained, to which individual log files to provide the required object image must be applied. If such data cannot be provided, Snapshot too old exception is raised. Solutions are based on extending the processing either into the archive log mode system [1], [18] or transforming the database into the temporal association [11], [14].

Log files are important in dealing with the indexes, as well. Transaction control ensures the reliability provided by the consistency and changes durability. Any change is logged, before the transaction approval, particular change vectors must be stored in the file system storage to ensure the possibility to recreate and execute the transaction after occurring any error. For the data changes, it is inevitable to store log files, it is impossible to turn off or limit such activity, due to the relational theory and transaction support. However, for index management, it is possible to use the nologging clause meaning, that changes on the index are not transaction logged. Whereas all data changes are stored in the database, it is correct, no data can be lost. On the other hand, after the failure, a particular index, which is marked by the nologging clause, does not reflect the current situation. As the consequence, such an index is switched into the unusable state meaning, that it cannot be used anymore. To make it usable, the related index must be recreated by using the rebuild clause of the index altering:

Create index *index_name* … nologging;

Alter index *index_name* rebuild;

If the index is unusable, it means, that it cannot be used at all for the evaluation, although it is still present in the system. Index recovery management is therefore vital functionality ensuring performance.

Index selection is an independent operation related to the database optimizer decision. It depends on various factors, index usability is just one of them. The decision-making core element is formed by the statistics for the table and index environment, which must be current to ensure performance. Another aspect is the size of the table and index regarding the fragmentation.

Decision making reflecting the individual attribute depends on the selectivity of the attribute, which represents the degree of the 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):

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

d: select count(distinct A) from T;
n: select count(*) from T;

The attribute is defined by the symbol A, the table is characterized by the T expression.

V. OWN CONTRIBUTION

The index requires ensuring the correctness of the data representation. Thus, any change must be applied to the index. Insert and Delete statements must be always implemented into the index structure. Update statement and index reflection depend on the index data set, whether it is changed or not.

Vice versa, Select statement does not influence the structure, it benefits from it, whereas relevant data can be directly located. By the transaction definition, it is ensured, that the index is always up to date, otherwise, it is automatically reflected by the invalid option (unusable).

When dealing with the data in the transactions, the management is delimited by the two-phase process – each data change must be recorded in the log file in the first phase, afterwards, the change is applied in the data and index themselves. If all these operations are finished successfully, operation, respectively the whole transaction can be approved. Fig. 1 shows the process of the two-phase commit protocol with an emphasis on index management. As evident, if the index is not present in the system, processing of the transaction changing data is executed faster. Individual operations and their processing limit the execution process and processing itself. The select statement can benefit, if the index is present in the system it must be suitable for the defined query. If such an index is not present, the TAF method performed by sequential scanning of the whole data block set associated with the table must be searched, loaded, and evaluated. Therefore, from the data retrieval perspective, the aim is to add a new index, if it is not present. Therefore, the perspective of the Select statement is to maximize the amount of the index in the system. Vice versa, another operation aim is to limit the amount, whereas they negatively influence such operations. Our contribution aims to limit the negative factor of the processing in destructive DML statements (Insert, Update, Delete), by shifting the index processing outside the transaction, however, by ensuring, that the index is always up-to-date and all data rows are there indexed.

As already described, the most often used index type is B+tree, which requires the tree balancing – the size of the route from the root to the leaf is always the same. Thus, many changes require adding a new node to the index, rebalancing, etc., which is a time and resource-demanding process. Our proposed solution aims to record the change in the extended added structure and apply it to the index as soon as possible. By using this technique, index rebalancing and node consolidation do not need to be part of the transaction itself.

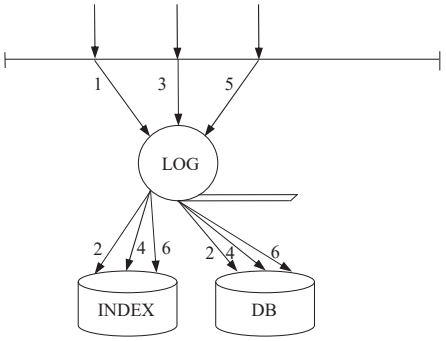

Fig. 1. Two-phase protocol

The solution process stages are in fig. 2. If a new data portion is to be loaded into the system, in the first phase, syntax and semantics are checked during the parsing stage. Afterward, the operation is logged, followed by storing the change in the pre-indexing structure – flat new index, respectively linked list. Finally, the data row is operated in the memory Buffer cache or physical database respectively approved by reaching transaction approval – commit. Thus, it is not necessary to balance the index during the transaction, just a new data portion is added, or changed. To ensure the consistency and performance reflected by the index, a new background process – DB_index_consolidator (DBIC) has to be introduced, which is responsible for the index weighting, optimizing, and balancing. It is, however, done outside the transaction. In the main system, an index is registered and a pre-indexing structure is created, to which any change is recorded in a FIFO manner. The individual records are gradually extracted and incorporated directly into the main indexes to ensure their reliability and speed of access. By default, the pre-processing layer consists only of a linked list from which the changes are applied to the index in the tree format. DBIC is responsible for the index tree balancing and removing pre-processsing layer node after its processing. Whereas FIFO approach is used, it is easy to ensure the low size demands and space deallocation.

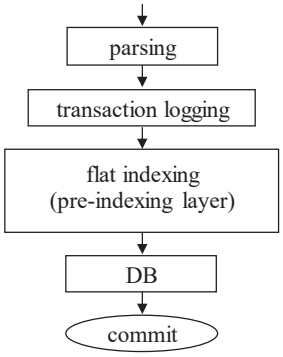

Fig. 2. Solution process stages

Fig. 3 shows the index management layer operated by DBIC. By using this proposed architecture, it is ensured, that each data portion is indexed, either in the main index or in the pre-indexing layer, from which it is consecutively loaded into the main index ensuring its balancing. Insert, Update, and Delete deal with flat index, data retrieval operates both structures.

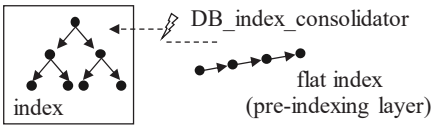

Fig. 3. Pre-indexing layer

By using the above-proposed architecture, even undefined values can be part of the indexing layer. In that case, for NULL values, a separate pre-indexing layer is created, called null_index_layer, however, such module is not operated by the DBIC (to balance data), whereas it is not possible to incorporate such data directly into the index, whereas NULL values cannot be mathematically compared using <, >, = operations, at all. The extended solution with the null_index_layer is in Fig. 4.

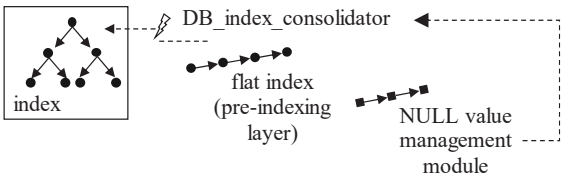

Fig. 4. NULL value management module extension

The proposed solution is based on the extended two-phase protocol. Index consolidation is done out of the transaction itself. As a consequence, such a transaction can be approved sooner, but by ensuring that the changed data are accessible using the extended index structure. We prefer adding an index_preprocessing layer in comparison with storing such values just into the log, mainly due to the log parsing necessity.

Select statement performance

Based on the previously described principles, it is ensured, that each data portion is part of the indexing layer, however, the row pointer can be present either directly in the index or pre-indexing layer. Thus, to evaluate the data, if the database optimizer selects an index to be used to access data, the database manager contacts the Index_provider background process, by which we extend the instance processing, as well. This background process is responsible for obtaining data pointers from the index layer by overcoming the original task of the process manager. In the first phase, data are located directly in the index. If the unique scan is used and data are obtained, processing ends resulting in getting the Rowid value to the environment. Vice versa, if the range scan method is used, processing cannot be stopped immediately due to the pre-indexing layer. Therefore, two workers are started in parallel, one of them is responsible for the index, the second worker scans the flat index in the pre-index sequentially. At last, the results provided by both workers are merged and shifted into the environment to load additional data attributes for the particular rows, if necessary.

VI. PERFORMANCE

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 10 million rows. No specific user-defined indexes were developed, for the evaluation, the primary key definition was used.

Experiment results were provided using Oracle Database 19c Enterprise Edition Release 19.3.0.0.0 - 64bit Production. Parameters of the used computer are:

- Processor: Intel Xeon E5620; 2,4 GHz (8 cores),
- Operation memory: 48 GB DDR 1333MHz
- Disk storage capacity: 1000 GB (SSD).

The first evaluation criterion reflects the costs of the processing expressed by the processing time for each destructive DML statement (Insert, Update, Delete) separately. Three system index approaches were used for the evaluation. **Model 1** does not use the index, at all, nor the primary key is defined. The current approach of the index management, by which each change on the index is encapsulated by the transaction is defined in **Model 2**. In that case, the transaction cannot be approved before applying the change to the index followed by the index balancing. **Model 3** uses its own proposed solution extending the index layer by the pre-indexing layer. The solution is based on extracting the balancing operation outside the transaction. Fig. 6 shows the performance results for the Insert operation. The processing time costs of Model 1 are 178.153 seconds, Model 2 requires 189.753 seconds and our proposed solution (Model 3) characteristics are 179.524 seconds. Thus, the index management requires additional 11.6 seconds, which expresses an additional demand rate of 6.51% (comparing Model 1 and 2). However, when dealing with our proposed solution, where the balancing is not present directly in the main transaction, demands are just 0.77% (comparing Model 1 and 3). Model 2 and Model 3 use the index for the data management, but Model 3 benefits and saves approximately 10 seconds of the processing in our defined environment. The reason is just based on flattening new index data into the linear

linked list. Transformation is done in the second phase by the launched background process. It is ensured, that all data are accessible by the index layer, original data are in the index itself, new values are in the pre-indexing module, and consecutively applied into the main index.

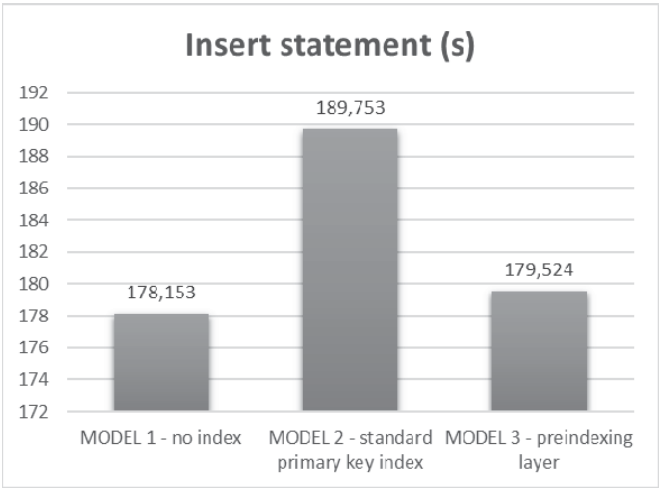

Fig. 6. Insert statement performance (in seconds)

In our performance evaluation environment, ten percent of the data is updated. Similarly, defined 3 models are used. Model 1 is not delimited by the applying changed into the index, on the other hand, data location and access must be done by the sequential block scanning requiring additional demands. In total, processing lasted 56.793 seconds. If the index is present, searching via it can be used. In our case, the Update statement condition was based on the primary key, thus the index definition is perfectly suitable. In Model 2, all data are directly in the index and the change is applied there followed by the balancing. Vice versa, when using Model 3, indexed data can be present either in the index and in the pre-indexing flat layer, as well. Therefore, two separate worker operations are launched. The Update operation itself is not balanced across the index set, just the pre-indexing layer is notified. The required processing time of Model 2 is 23.578 seconds and 24.012 seconds for Model 3, which expresses minimal slowdown (1.84%). Fig. 7 shows the results in the graphical form.

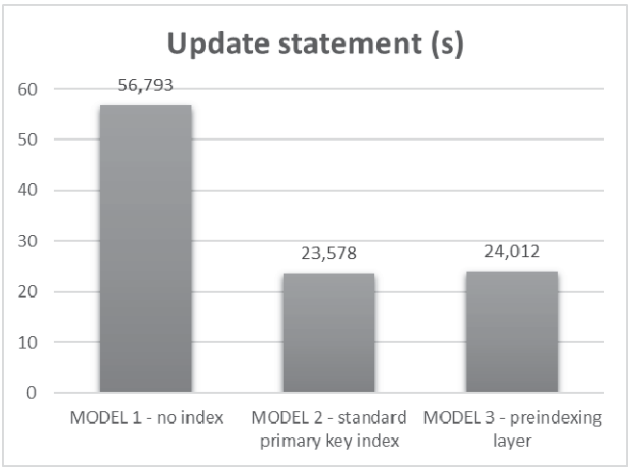

Fig. 7. Update statement performance (in seconds)

Finally, Delete statement performance was evaluated. To ensure the performance and data consistency in the index, the Delete statement is operated directly on the index layer. For Model 3, the pre-indexing layer must be checked, as well, whereas the reference to the particular object can be located there. Similarly to the previous part of the evaluation, the same three models are used, ten percent of the data is to be deleted. If the index is present, it can be used for the data access and consecutively operated by the removal operation. Note, that the index itself is not rebalanced in the Delete operation to remove free nodes [9], [10], [20].

Model 1 requires 26.577 seconds, Model 2 demands are 21.429 seconds, and Model 3 processing time is 22.603 seconds. Fig. 8 shows the results. The reached results are almost uniform with no specific peaks. Referencing Model 1 (100%), Model 2 saves 19.37% and Model 3 saves 14,95%. Although there are additional processing demands in Model 3 requiring management of the pre-indexing layer, it is operated automatically by the worker processes of the instance done in parallel.

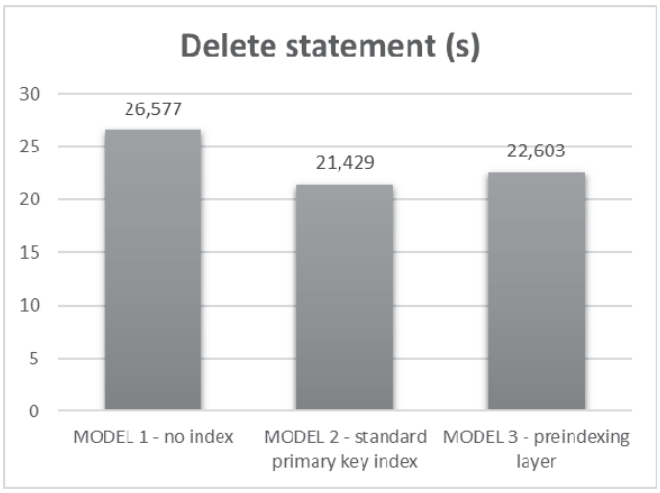

Fig. 8. Delete statement performance (in seconds)

The previous evaluation analysis was based on changing the structure of the index by adding new tuples there, either directly, or by the flat pre-indexing layer. In such a case, the DBIC process rebalances the index to ensure performance, however, it is done outside the main transaction. In this section element, the performance of the Select statement is reflected. Model 1 is straightforward. Whereas no index is present, sequential scanning operated by the TAF method must be used. Model 2 provides the best solution, all data tuples are indexed, so the Index Range Scan method can be used directly. In our experiment, the condition is done based on the primary key, 10 percent of data is obtained. Performance of the Model 3 can be located between Model 1 and Model 2. The main index is scanned in the tree principle by traversing it from the root to the leaf, where the Rowid values are extracted, but we are approaching in parallel the pre-indexing layer, which is scanned sequentially. This layer is small and does not consist of any fragmentation. Access is therefore effective, it is modelled by the linear linked list. Fig. 9 shows

the results. The additional demands of Model 3 in comparison with Model 2 are 0,707 seconds. That time just reflects the necessity of union results from the two data workers based on the index and pre-index layer.

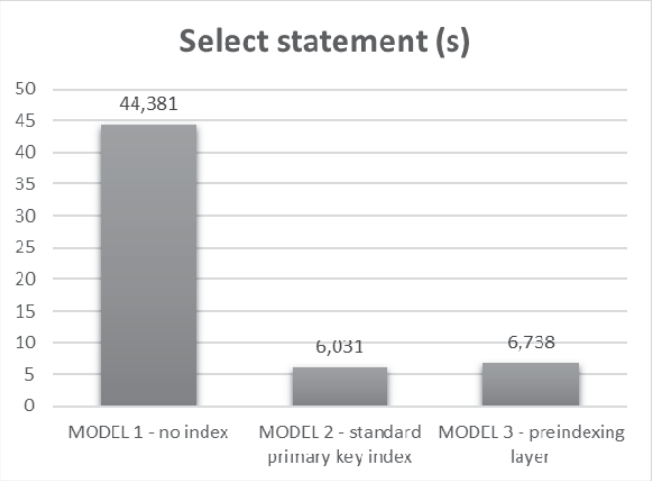

Fig. 9. Select statement performance (in seconds)

VII. CONCLUSIONS

Relational database systems belong to the very often used approach to model, store, and access data currently. The strong support of the transactions extends the robustness by ensuring data consistency anytime. The amount of the data is still rising in conventional systems, by shifting the solution to the temporal modeling all changes over time, the problem is even far deeper. Effective data access is therefore inevitable part of the whole system, whereas database management is just one element of the complex information system architecture. To ensure reliability and fast access, indexes on the data tables are created and managed by all operations influencing data automatically. Thus, for the Insert, Update and Delete statements, adding an index to the system brings new demands whereas it must reflect the change by consolidating and balancing it. Vice versa, locating data in the database is far more effective, whereas direct access to the relevant block can be done. Thus, for the data location of the Insert, Update and Delete statements supervised by the Select operation, significant benefits can be identified.

Adding new indexes to the system has, therefore, a positive effect for the data retrieval, but the negative aspect of the consolation and rebalancing is present. This paper aims to provide a pre-indexing layer to limit the necessity of the index balancing method as the most demanding index operation inside the main transaction. All change data are stored in a flat index modelled using the linked-list. After passing them into such a structure, the transaction can be approved and ended immediately. Background process DBIC monitors the flat index automatically and incorporates ongoing changes into the index itself, but in a separate transaction. Thanks to that, the performance of the destructive data manipulation operations is ensured, as is evident from the experiments. If the database optimizer selects the index path method for the query evaluation, additional worker processes are allocated to host the index access, as well as to scan the flat index in the pre-indexing layer. By using such a strategy, the performance of the Select statement is degraded only in a minimal manner (based on our experiment environment, the additional demands are 1.593%). Comparing to other operations, like Insert statement, by which the saving is more than 6%.

In the future, we would like to evaluate several structures directly in the pre-indexing layer and their impact on global performance. Shifting the solution into the table partitioning and data distribution is part of our emphasis, as well.

ACKNOWLEDGMENT

This publication was realized with 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 Europen 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] Bottoni, P., Ceriani, M.: Using blocks to get more blocks: Exploring linked data through integration of queries and result sets in block programming, IEEE Blocks and Beyond Workshop (Blocks and Beyond), 2015.

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

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

[6] 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

[7] 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

[8] Feng, J., Li, G., Wang, J.: Finding Top-k Answers in Keyword Search over Relational Databases Using Tuple Units, IEEE Transactions on Knowledge and Data Engineering (Volume: 23, Issue: 12, Dec. 2011) , 2011.

[9] Honishi, T., Satoh, T., Inoue, U.: An index structure for parallel database processing, Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing, 1992.

[10] Kriegel, H., Kunath, P., Pfeifle, M., Renz, M.: Acceleration of relational index structures based on statistics, 15th International Conference on Scientific and Statistical Database Management, 2003

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

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

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

[14] 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

[15] Shen, J., Zhou, T., He, D., Zhang, Y.: Block Design-Based Key Agreement for Group Data Sharing in Cloud Computing, IEEE Transactions on Dependable and Secure Computing (Volume: 16, Issue: 60), 2019.

[16] 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.

[17] 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

[18] 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.

[19] Tilgner, M., Ishida, M., Yamaguchi, T.: Recursive block structured data compression, Proceedings DCC '97. Data Compression Conference, 1997.

[20] 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

[21] Yu, Y., Yao, Y.: Application of Keyword Dynamic Query Software in Relational Database based on Big Data, International Wireless Communications and Mobile Computing (IWCMC), 2020.