The Effect of Partitioning and Indexing on Data Access Time

Veronika Šalgová, Karol Matiaško
University of Žilina
Žilina, Slovakia
Veronika.Salgova, Karol.Matiasko@fri.uniza.sk

***Abstract*—An increasing amount of data is stored and managed in today's information systems. For this purpose, relational databases are used very often. Fast access to data is becoming increasingly important and great emphasis is placed on its improvement. The data access time can be significantly improved by creating partitions and index structures, or their combinations. From the point of view of efficiency, it is not appropriate or necessary to access all data. They can be divided into smaller parts, which will facilitate the execution of certain operations and bring efficiency, whether in terms of time or performance. As far as indexing is concerned, there is also an opportunity to access only the data sought, thus avoiding the Table Access Full method. This paper deals with the effect of partitioning and indexing on data access time, which is compared in seven different scenarios of different combinations of partitions created over tables and indexes.**

I. Introduction

Relational databases are one of the most often used techniques to store data of the information systems. Despite the fact that they were first defined in the 60s of the last century, they are still very powerful to cover the current environment and technology demands. An important indicator is the time required to perform basic operations to retrieve data. Reducing data access time is becoming increasingly important and it receives a lot of attention. A large amount of data is used in many areas. Higher data access times bring undesirably higher costs. For such large data, it is necessary to process them as optimally as possible and thus reduce the costs of different forms. A suitable solution for powerful optimization of query execution is building indexes on data sets. An indexing strategy is the design of an access method to a searched item. Accessing a smaller group of data that are partitioned instead of going through all the data can bring great efficiency as well. The created indexes, partitions, and also their interconnection can have a significant effect on data access time [2], [6]. The aim of this paper is to compare various situations that differ in the usage of the index or the attributes on which the index is created. Each situation was tested on a table with created partitions or without partitions and also with a combination of partitioned and non-partitioned indexes.

II. Indexing

An important and powerful part of the optimization of query processing is an index structure that can significantly improve the performance of the database [1]. The index itself is used for direct access to the row inside the database by using ROWID on the bottom leaf nodes. ROWID is the locator for the data and consists of these layers: identification of the data file, in which the row resists, the pointer to the block, and position inside it [3]. With ROWID, table data can be easily retrieved with a minimum number of reads executed. Various structures of an index can be used in database systems, but the most widely used structure is the B-tree, respectively B+ tree. It is a balanced structure, which means it has the same length from each leaf to the root. It does not degrade over time and it remains balanced [4]. However, if there exists no suitable index, every single row needs to be read to find the desired information in the table. This method is known as *Full Table Scan* and is one of the most expensive operations. To ensure efficiency and robust performance, it is necessary to limit the usage of Full Table Scan methods [7], [11], [13].

III. Partitioning

A technology physically dividing certain large objects of relational databases into smaller parts based on the logical division of the data is called *partitioning*. It is possible to divide tables, indexes, and index-organized tables into smaller sections, as is shown in Fig. 1. These database objects are then enabled to be managed and accessed at a finer level of granularity. A subdivided part of a database object is called *a partition*. The main advantages of creating partitions in databases are their manageability, higher performance, availability reasons, or load balancing. [5] Performance is increased by working only on the data that is relevant. Availability is improved on the basis of individual partition manageability. Costs are decreased by storing data in the most appropriate manner. Partitioning has also an easy implementation because it is not required applications and queries to be changed [8], [9], [14].

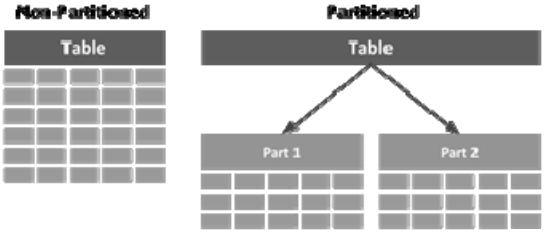

Fig. 1. Partitioned table [6]

There are three techniques by which partitions can be created. They are based on information allocation processes, which are *Hash*, *Range*, and *List*. According to them, there exist following partitioning techniques:

- Hash Partitioning
- Range Partitioning
- List Partitioning

When creating partitions, each row in a partitioned table must be assigned only to a single partition. The data is divided into partitions based on a value called *a partition key.* This key can consist of one or more columns, which varies according to different partitioning techniques. It is important to select a partition key to be a column that is almost always used as a filter in queries. When it is so, it is not necessary to access all the data but only the relevant partitions. It means, *partition elimination* is used, and it can bring a significant performance improvement when querying large tables.

In our experiments, the *range partitioning* technique was used. It maps data into partitions according to ranges of defined partition key values. Ranges must be specified for each partition so that it can include rows for which the partitioning expression value belongs to a given range. Ranges should be contiguous, but they cannot overlap with each other. In general, range partitioning is useful in cases when data can be logically segregated by some values. It is the most common one among the partitioning techniques and is able to take advantage of partitioning elimination in many cases, including the use of exact equality and ranges – *less than, greater than, between*, and so on. It offers the possibility to use a *MAXVALUE*, which represents a value that is always greater than the largest possible value. It serves as the least upper bound so it can catch all values that exceed the specific ranges.

IV. PARTITIONED INDEX

In the same way as tables, indexes can be divided into partitions as well. It is possible to have partitioned tables without partitioned indexes, but also to have a non-partitioned table with partitioned indexes. Fig. 2 shows a non-partitioned table with two types of indexes. The index on the left side is partitioned into 3 partitions, and the index on the right side is non-partitioned.

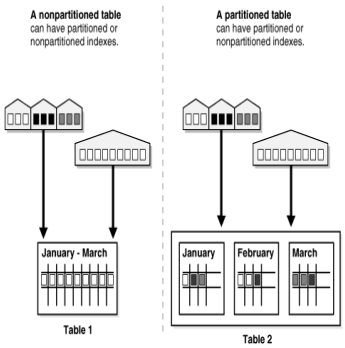

Fig. 2. Indexes of a non-partitioned table

Fig. 3 shows a table partitioned into 3 partitions related to different months. Two different indexes are linked to it, the partitioned index on the left side and the non-partitioned index on the right side.

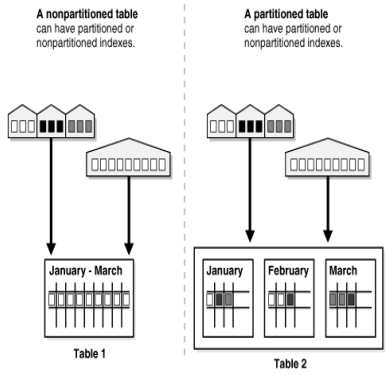

Fig. 3. Indexes of a partitioned table

An independently partitioned index is referred to as a *global index*. A partitioned index automatically linked to a table's partitioning method is referred to as a *local index.*

A. Local partitioned index

A great availability is offered by a local partitioned index, which is also easier to manage than other types of partitioned indexes. All of its keys refer only to rows stored in a single underlying table partition. It offers equipartitioning as each partition of a local index is associated with exactly one partition of the table. For this reason, the index partitions are automatically kept synchronized with the table partitions, so each table-index pair can become independent. It means they are both added, dropped, or split in the same way. Any actions making the data in one partition unavailable or invalid affect only a single partition. A new partition cannot be explicitly added to a local index. It can be added only when a new partition is added to the underlying table as well. Similarly, a partition cannot be explicitly dropped from a local index. It can be dropped only when a partition from the underlying table is dropped as well. Fig. 4 shows a structure of a local partitioned index and its relationship to the table partitions. The three top objects refer to index partitions and the bottom three objects refer to table partitions [10], [15].

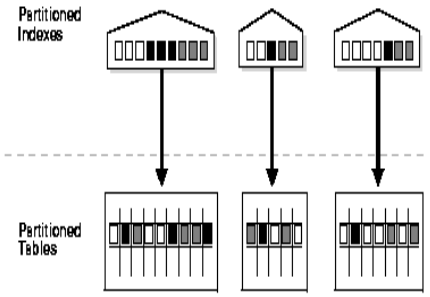

Fig. 4. Local partitioned index

A local index is created by specifying the LOCAL keyword as follows:

```
CREATE INDEX index_name
ON table_name(attribute1[, attribute2, …]) LOCAL;
```

1) Local prefixed index: A local partitioning index is prefixed if it is partitioned on a left prefix of the index columns.

2) Local non-prefixed index: A local partitioning index which is not partitioned on a left prefix is referred to as a non-prefixed index. This type of index cannot be unique unless the index key is a subset of the partitioning key.

B. Global partitioned index

The partitioning key independent from the table's partitioning method is making global partitioned indexes more flexible. The fact that all rows in the underlying table can be represented in the index is ensured by a partition bound called *MAXVALUE*, which is higher than the highest partition of a global index. Fig. 5 shows a structure of a global partitioned index and its relationship to the table partitions. The three top objects refer to index partitions and the bottom three objects refer to table partitions. Thus, unlike a local partitioned index, in this index, its partitions bind to different partitions of the table [12].

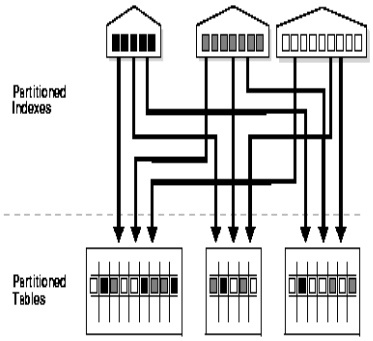

Fig. 5. Global partitioned index

A global index is created by specifying the GLOBAL keyword as follows:

```
CREATE INDEX index_name
ON table_name(attribute_name) GLOBAL
PARTITION BY RANGE (attribute_name) (
  PARTITION partition1_name VALUES LESS THAN (x),
  PARTITION partition2_name VALUES LESS THAN (y),
  …
  PARTITION partitionN_name VALUES LESS THAN
                                        (MAXVALUE)
);
```

V. METHODOLOGY

Data access time characteristics have been obtained by using Oracle 18c database system based on the relational platform.

Experiment results were provided using Oracle Database 18c Express Edition Release 18.0.0.0.0 – Production Version 18.4.0.0.0. Parameters of the used computer are:

- Processor: Intel(R) Core(TM) i5-3317U; 1.70GHz
- Operation memory: 8GB
- HDD: 500GB

To compare the data access time, two tables were created. Both tables had identical data and contained 1,000,000 rows with 4 columns of Integer, Varchar2(30), Date, and Integer data type. One table was created without partitions and the other one was divided into 25 partitions according to the *range partitioning* technique which was applied on the last integer column storing data about percentage. It means that each partition contained a range of 4 distinct percentage values. Data were generated into partitions evenly, so each partition had approximately 40,000 rows. A non-partitioned table was created as follows:

```
CREATE TABLE table_not_partitioned
(id INT PRIMARY KEY,
 name VARCHAR2(30),
 dateB DATE,
 percents INT);
```

A table with 25 partitions was created as follows:

```
CREATE TABLE table_partitioned
(id INT PRIMARY KEY,
 name VARCHAR2(30),
 dateB DATE,
 percents INT)
PARTITION BY RANGE (percents) (
  PARTITION p1 VALUES LESS THAN (5),
  PARTITION p2 VALUES LESS THAN (9),
  ...
  PARTITION p24 VALUES LESS THAN (97),
  PARTITION p25 VALUES LESS THAN (101)
);
```

Situations without created indexes were tested over each of these two tables. After that, different types of indexes were created for these tables as well. The different scenarios concerned a non-partitioned and a partitioned index.

Partitioned indexes were created as global, local prefixed, and local nonprefixed indexes. In the case of partitioned indexes, the partitions were created in the same way as in the tables, and thus they related to a percentage column that was divided into 25 ranges. Seven executed scenarios are shown in Table I. Each scenario indicates whether the table was partitioned or not, and which type of index was created on the table.

TABLE I. EXECUTED SCENARIOS

	TABLE	INDEX
1	nonpartitioned	-
2	partitioned	-
3	nonpartitioned	partitioned (global)
4	nonpartitioned	nonpartitioned
5	partitioned	nonpartitioned
6	partitioned	partitioned (local prefixed)
7	partitioned	partitioned (local nonprefixed)

The global partitioned index in scenario no. 3 was created on the attribute *percents* as follows:

```
CREATE INDEX index_global_percents
ON table_not_partitioned(percents) GLOBAL
 PARTITION BY RANGE (percents) (
  PARTITION p1 VALUES LESS THAN (5),
  PARTITION p2 VALUES LESS THAN (9),
  …
  PARTITION p24 VALUES LESS THAN (97),
  PARTITION p25 VALUES LESS THAN (MAXVALUE)
 );
```

The non-partitioned index in scenarios no. 4 and no. 5 was created in a similar way for both of the tables on the attribute *percents* as follows:

```
CREATE INDEX index_percents_nonpartitioned
ON table_not_partitioned(percents);
```

The local prefixed partitioned index in scenario no. 6 is partitioned on a left prefix of the index columns and was created on the attribute *percents* as follows:

```
CREATE INDEX index_local_percents
ON table_partitioned(percents) LOCAL;
```

The local non-prefixed partitioned index in scenario no. 7 is not partitioned on a left prefix of the index columns. It was created on the attribute *dateB* and *percents* as follows:

```
CREATE INDEX index_local_percents
ON table_partitioned(dateB, percents) LOCAL;
```

VI. Results

Data access times of executed seven scenarios are shown in Table II in milliseconds. The left column represents the number of accessed ranges.

When comparing scenarios no. 1 and no. 2 in which there are tables without any indexes created, it can be concluded that when accessing data related to successively from 1 to 10 partitions, the access time was significantly shorter for a partitioned table. When selecting data for one partition, it was only 4 milliseconds compared to 52 milliseconds. The data access time of the partitioned table increased with the increasing number of accessed partitions. There was a change in access to 12 partitions. The partitioned table data access time was 67 milliseconds, which was more than the non-partitioned table data access time of 61 milliseconds. Thus, in this situation, the number of 12 partitions represented the limit when the partitioned table was no longer more advantageous than the non-partitioned table in terms of data access time. A comparison of scenarios no. 1 and no. 2 are shown in Fig. 6 and Fig.7.

TABLE II. Data access times of 7 scenarios

	S(1)	S(2)	S(3)	S(4)	S(5)	S(6)	S(7)
1	52	4	5	6	6	6	6
2	53	6	7	8	9	12	9
3	51	7	9	13	16	15	18
4	54	9	14	16	16	18	20
5	58	14	19	25	28	26	25
6	60	16	23	33	32	29	29
7	59	20	26	59	60	29	33
8	61	23	35	61	62	27	36
9	60	24	33	62	64	26	38
10	58	27	41	66	67	30	39
11	61	32	38	63	69	31	42
12	61	67	40	66	65	32	45
13	60	566	43	68	64	64	49
14	59	619	45	70	67	39	50
15	61	672	50	75	65	41	52
16	55	732	53	66	68	59	55
17	58	797	54	66	66	54	60
18	60	805	61	69	69	51	61
19	59	920	63	70	70	49	65
20	58	861	59	71	67	54	74
21	56	981	63	73	68	51	70
22	58	986	65	71	69	62	67
23	57	1049	64	72	71	1064	71
24	57	1070	69	68	72	1065	71
25	58	1153	69	67	72	1067	72

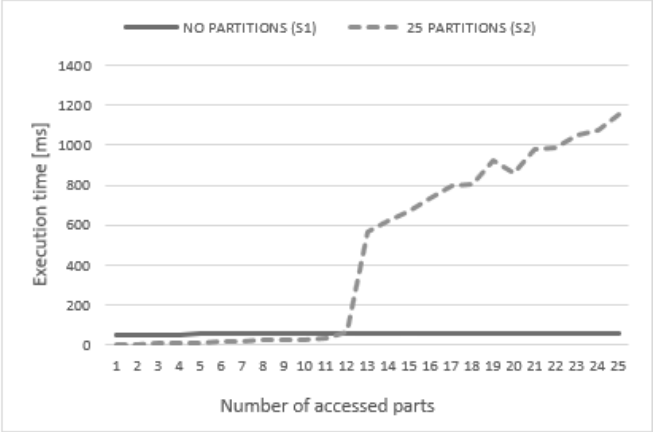

Fig. 6. Data Access Time – Scenarios 1, and 2

Fig. 7. Data Access Time – Scenarios 1, and 2 - Enlargement

Fig. 8 shows data access times of scenarios no. 1, no. 3, and no. 4 concerning the table without partitions. Scenario no. 1, which was without any indexes, kept data access time between 51 and 61 milliseconds, without major fluctuations. Scenario no. 4, which concerned the nonpartitioned index, had a much shorter time from the beginning. When accessing data that would belong to one partition, the access time was only 6 milliseconds. With the increasing number of partitions that would be accessed, the access time gradually increased to 8, 13, 16, 25, 33, and 59 milliseconds, and the access time to the data in the range of 7 partitions was the same as in the scenario no. 1, for 61 milliseconds. Subsequently, the access time increased slightly from this limit. Scenario no. 3 started with an access time of 5 milliseconds and had the lowest time in most cases, up to access data that would cover 18 partitions. After this limit, the most advantageous scenario was the one without any indexes.

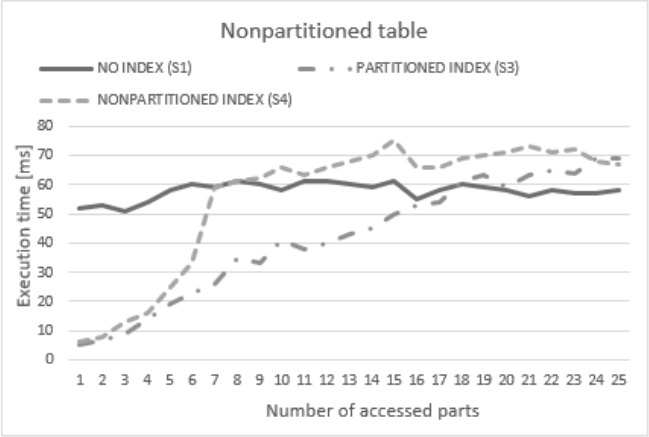

Fig. 8. Data Access Time – Scenarios 1, 3, and 4

The situations regarding the partitioned table are shown in Fig. 9. Scenario no. 2, which was without any indexes, had the lowest time in the first half of the cases, compared to scenarios no. 5, no. 6, and no.7. Since the access to 11 partitions, the access time has grown significantly, and since the access to 12 partitions, scenario no. 2 had the largest data access time. Scenario no. 5, concerning the nonpartitioned index, had a much smaller increase. Its access times ranged from 6 to 72 milliseconds. The only major growth in time was between access to 6 and 7 partitions, where the time increased by 32 milliseconds to 60 milliseconds. Subsequently, the times did not have such a large growth, they always increased by a maximum of 3 milliseconds. Scenario no. 6, concerning the local prefixed partitioned index, had access times to 1 to 22 partitions ranging from 6 to 62 milliseconds. Since access to the 23 partitions, the time has increased very significantly, reaching a time similar to that without any index, of more than 1060 milliseconds. Scenario no. 7, concerning the local nonprefixed partitioned index, had very similar access times as the nonpartitioned index scenario, in the same range of 6 to 72 milliseconds.

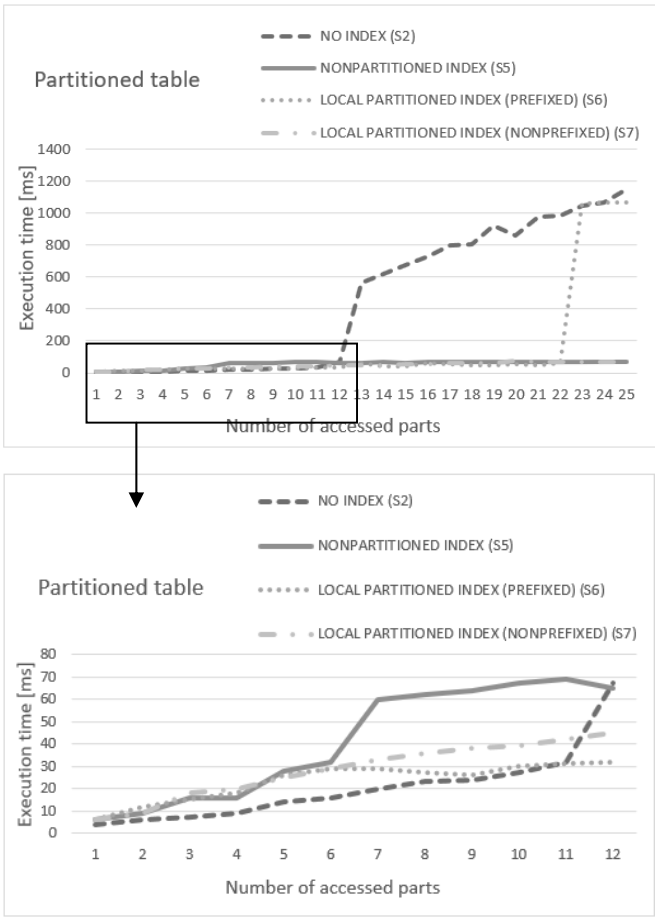

Fig. 9. Data Access Time – Scenarios 2, 5, 6, and 7

VII. Conclusion

The efficiency of data access is one of the most important tasks in ensuring system performance. The amount of data is constantly growing, and therefore it needs to be processed in an efficient way. When using large amounts of data, great emphasis is placed on data access time. Creating indexes, partitions and their various combinations can bring significant improvement of data access time in many situations.

In the experiments, the access times for the data in the table with the created partitions and the table without any partitions were compared in the DBS Oracle environment. We tested several situations that differed in the type of indexes created on both tables, such as non-partitioned, global partitioned, local partitioned prefixed, and local partitioned non-prefixed indexes.

The results showed that data access time differed a lot in various scenarios. For the first ranges of selected data, the worst access time was caused by using the non-partitioned table with no indexes. Here, the access time was at the beginning about 9-13 times worse than in the remaining scenarios. However, from about half of the accessed ranges had a significant slowdown in access time scenario with the partitioned table with no indexes and it started to be about 10 times slower than a scenario with the non-partitioned table with no indexes.

All scenarios in which partitioning, indexing, or their combinations were used started at approximately the same data access time of 4, 5, and 6 milliseconds. Regarding access to data belonging to the first 11 partitions out of 25, the scenario with the partitioned table with no indexes had the best results. However, after this limit of 11 partitions, it significantly became the slower scenario.

From the results of experiments, it can be deduced that the use of partitioning, indexes, or their combinations can significantly speed up data access time. However, with frequent access to a large amount of data to which a large number of partitions or index nodes are bound, the access time is similar as in the scenario without partitions and indexes, or in some situations, it may be even much worse.

Therefore, it is important to consider the appropriate choice of partition and index types, depending on the frequency and the volume of accessed data.

ACKNOWLEDGMENT

This publication is supported by the *Grant system* of the University of Zilina.

REFERENCES

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

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

[3] S. Ceri, M. Negri, and G. Pelagatti, "Horizontal data partitioning in database design", ACM SIGMOD international conference on Management of data, 1982, pp. 128-136.

[4] R. Čerešňák, M. Kvet, "Comparison of query performance in relational and non-relational databases", Transportation Research Procedia, 2019, vol.40 (pp. 170-177), 2352-1465.

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

[6] M. Desai, R. Mehta, and D. Rana, "A Survey on Techniques for Indexing and Hashing in Big Data", 2019, doi: 10.1109/CCAA.2018.8777454.

[7] G. Graefe, "Sorting And Indexing With Partitioned B-Trees." *CIDR*. Vol. 3. 2003.

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

[9] J. Janech, M. Tavač, and M. Kvet, "Versioned database storage using unitemporal relational database," *2019 IEEE 15th International Scientific Conference on Informatics*, Poprad, Slovakia, 2019, pp. 000031-000036, doi: 10.1109/Informatics47936.2019.9119269.

[10] D. Kuhn, S.R. Alapati, and B. Padfield, "Partitioned Indexes", 2016, In: Expert Oracle Indexing and Access Paths. Apress, Berkeley, CA. doi: 10.1007/978-1-4842-1984-3_6

[11] M. Kvet, "Relational Data Index Consolidation," *2021 28th Conference of Open Innovations Association (FRUCT)*, Moscow, Russia, 2021, pp. 215-221, doi: 10.23919/FRUCT50888.2021.9347614.

[12] M. Kvet and M. Kvet, "Relational Pre-indexing Layer Supervised by the DB_index_consolidator Background Process," *2021 28th Conference of Open Innovations Association (FRUCT)*, Moscow, Russia, 2021, pp. 222-229, doi: 10.23919/FRUCT50888.2021.9347573.

[13] M. Kvet, V. Šalgová, M. Kvet, and K. Matiaško, "Master Index Access as a Data Tuple and Block Locator," *2019 25th Conference of Open Innovations Association (FRUCT)*, Helsinki, Finland, 2019, pp. 176-183, doi: 10.23919/FRUCT48121.2019.8981531.

[14] C. Qi, "On index-based query in SQL Server database", 2016 *35th Chinese Control Conference (CCC)*. doi: 10.1109/chicc.2016.7554868.

[15] M. Wang, M. Xiao, S. Peng, and G. Liu, "A hybrid index for temporal big data", 2017 Future Generation Computer Systems, 72, 264–272. doi: 10.1016/j.future.2016.08.002