A Performance Evaluation of Some Function-Based Indices in RDBMS

Marko Vještica*, Slavica Kordić*, Milan Čeliković*, Vladimir Dimitrieski*, Ivan Luković*, Jovana Vidaković**

* University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia
** University of Novi Sad, Faculty of Sciences, Novi Sad, Serbia
marko.vjestica@uns.ac.rs, slavica@uns.ac.rs, milancel@uns.ac.rs, dimitrieski@uns.ac.rs, ivan@uns.ac.rs, jovana@uns.ac.rs

***Abstract* — When using Data Warehouse systems, companies need many reports that include time consuming queries on large amounts of data. They often use various functions for business analyses that could make queries more difficult to execute. Users are not always satisfied with the execution time of queries. There are different techniques to reduce execution time of queries. In this paper, we considered an influence of indexing techniques on the execution time of queries to gather reports faster. We created queries that contain functions and we measured their performance without using indices or using different function-based indices. Test cases were separated into two sections. First, we tested queries with more than one function nested one in the other in the *WHERE* clause. The second part of test cases presents queries with substring functions in the *WHERE* clause with different selectivity. They were also tested with different aggregation functions in the *SELECT* statement. We present results of an evaluation of query execution time using some function-based indices.**

I. INTRODUCTION

The execution time of queries and transactions is important in database systems, especially with large amounts of data. Introducing additional data structures, like materialized views and indices, could improve query execution time, but it demands additional memory in a database system, and it may impair execution time of transactions. It is because data updates in the database causes updates of these additional data structures. All of this is even more noticeable in Data Warehouse systems storing a lot of data and executing many intensive queries.

Data Warehouse systems are used in many domains (e.g. manufacturing, telecommunications and healthcare). A lot of data is stored, aggregated and analyzed in an On-Line Analytical Processing (OLAP) manner. We are witnessing an explosion of data coming from various sources in many different application domains [1, 2]. Data Warehouse systems are query-intensive and can access millions of records to provide reports and valuable insights to companies but at the expense of computing resources. The reason behind this is companies' need to gather and analyze as much data as possible in order to stay competitive and improve their business [2]. That makes storing, analyzing and querying data even more challenging. In order to provide reports to the companies in a relatively short amount of time, the query execution time is one of the most important parameters of Data Warehouse systems that needs to be optimized. It is mostly influenced by the number and size of blocks that are needed to be gathered from a memory and by the time needed to calculate expressions and functions in queries. These queries may have time consuming functions, which results are needed in many reports.

Data Warehouse systems are mainly read-only databases, so it is more important to have short query execution time than fast transactions [3]. Because of the read-only property, it could be beneficial to apply different materialized views or indices.

There are recommendations that could help users choose appropriate indices [4], but there are only few guidelines that present query execution performance measured in the real environment. In this paper, we are considering the indexing techniques to make execution time of queries shorter and we will present testing results of the different queries executed in the real environment.

We tested queries with nested functions and with different substring functions in the *WHERE* clause. Our first assumption was that the DBMS could use indices based on the nested functions only to make the execution of outer functions faster. The second assumption was that the DBMS could use indices based on the substring functions to accelerate execution of queries with substring functions that are not same as the indexed ones. We need to check these assumptions by measuring query execution performance. If these assumptions are right, the reports could be gathered faster and memory space could be saved by not creating index for every function in queries, which presents our goal. We also want to provide guidelines to choose an appropriate index for queries with different aggregation functions and selectivity, based on the measurements in the real environment with large amounts of data. Our results provide new insights into advantages of using function-based indices.

Apart from Introduction, this paper is organized as follows. Section II presents related work of the query and index performance evaluation that are important for our work. Section III contains an overview of indexing techniques related to this paper. Section IV describes testing procedure that is used to measure execution time of queries. Section V presents results and analyses of different queries execution with or without function-based indices. Section VI contains conclusions and an overview of results presented in the previous section.

II. RELATED WORK

The performance optimization and tuning of query execution time are research topics in the field of the databases. There is a problem of appropriate index selection for different kinds of queries. If the right index is

chosen, it could reduce query execution time, otherwise it could even prolong it. When an appropriate index exists, in many cases only the index is accessed so the table access can be reduced or even eliminated.

In [5], it is stated that many indexing techniques exists in the literature, but there is no guidance to compare and select them. Many indices were analyzed and categorized based on the found literature, but there were no testing results in the real environment. As far as we know, there are not many studies that present query execution performance for the different indexing techniques, especially those based on the functions, which inspired us in our research.

Our testing procedure is like the one in the studies of Medina [6, 7], which will be explained in the Section IV. They measured real performance of the queries on the Oracle RDBMS. In these studies, evaluation of indexing techniques for necessity-based fuzzy queries in classical RDBMS were performed. Performance was also measured using indices in the audit environment in [8], with the execution of specific query, but none of these studies used function-based indices.

In many papers, it is presented new type of indices, algorithms or it is presented new cost model to calculate and evaluate query performances using those indices. Similar work was done in [9], mostly comparing Bit-sliced and Projection indices with different aggregation functions. We should also mention a performance study [10], in which a new framework was presented to compare indices in Data Warehouse systems. Bitmap and tree-based indices for range queries were compared, based on the nine parameters that could influence performance. In our paper, we have two variable parameters: the number of records in a table and in indices and the number of records filtered with the *WHERE* clause, which were both discussed in [10].

None of the examined studies have tests cases and measurement of query performance using function-based indices. We tested queries with nested functions in the *WHERE* clause. If there are two functions in the *WHERE* clause, one nested in another, we wanted to test if an inner function only may be indexed and the DBMS could use that index to calculate results of an outer function. Then we indexed both the inner and the outer function and measure performances of the same query with or without those indices. We also tested if the DBMS could use indices based on a substring function that differs from the ones in the queries and evaluate their performance. Using different nested functions and substring functions are usual in many reports and we wanted to test if there are ways to optimize queries execution time needed for those reports. With these queries we also have measured and evaluated an influence of the two variable parameters.

III. INDEXING TECHNIQUES

Indices are memory structures that can reduce query execution time by reduce memory reading time and memory access frequency, but additional memory is needed for indices to be stored. In this paper, we are evaluating the execution time of queries when function-based indices are created. In Oracle RDBMS, a function-based index may be created as a B-tree index or a bitmap index that are indexing one or more functions. In this section we will present an overview of those index types.

B-tree indices have relatively good performance for different queries, like a range search or exact match. This kind of index type doesn't provide good performance with attributes that have few distinct values. In that case, it is better to use bitmap indices [11, 12].

Bitmap indices are created in a Data Warehouse system, because they have good performance where the databases are mainly read-only. An advantage of their usage is when executing queries with bitwise operators in them or if there is a need to count the number of records in the table with or without additional condition. It also uses less memory space than most of the indices. Oracle server places bitmap indices in the B-trees to provide better query execution performance [11, 12].

Function-based indices are used to index results of one or more functions or expressions that are often used in queries. Executing some functions on a large amount of data could be time consuming and using function-based indices could be helpful in that case. With function-based indices there is no need to access a table and execute indexed functions on data, but only to access the index to get results [11, 12].

In section V, we will present testing results of queries execution using bitmap and B-tree function-based indices. Based on the advantages and disadvantages that we already described, we will discuss results in our test cases.

IV. TESTING PROCEDURE

For testing purposes, we used a computer with Intel(R) Core(TM) i7-6700HQ CPU @ 2.60 GHz, 4 Cores, 6 MB Cache; 16 GB DDR4 RAM, 2133 MHz; NVIDIA GeForce GTX 960M GPU, 4 GB DDR5; 1 TB HDD, 7200 rpm. We were testing with Oracle Database 11g Enterprise Edition RDBMS and Microsoft Windows 10 64-bit operating system. Before each execution of a query, database block buffer cache was cleared in order to always get data blocks from a hard drive. Otherwise, cashed blocks could be obtained which would make performance comparison invalid. *HINT* clause was also used, because Query optimizer may decide that it is better not to use certain index and for testing purposes, we need to force the index usage. Every query was tested ten times, and an average execution time was calculated.

We used tables with columns of various types, different number of records ranging from one to eight million and queries of different selectivity, depending on the tests we performed. Records were randomly generated with a code generator, in which we could change the number of records and the number of unique values of the columns.

We didn't create any constraint, which was also done in [8], because it may hinder our test results. For example, creating a primary key constraint, the Oracle RDBMS would also implicitly create a B-tree index over it.

V. RESULTS

In this section we are presenting testing results of some function-based indices. First, we present queries with more than one function in the *WHERE* clause. Functions are nested and we tested queries execution time with or without indices that are based on those functions. Queries with different kind of nested functions are usual in many reports and we wanted to test if it is possible to reduce

query execution time and if we need to create every single combination of those functions. The second part of this section presents testing results of different substring functions. Indexed results of a substring function are a subset, a superset or there aren't any intersecting points with the results from substring function in the queries. There is need to process and analyze a part of string values in some reports and we wanted to test if a DBMS can use one indexed substring function to obtain results from other substring function. We also tested cases when there is a different number of records in a table, different selectivity and different aggregation function.

A. *Queries with nested functions*

Queries with nested functions were tested on a single table having ten columns. It has one column reserved for a primary key, one column for a testing purpose and eight auxiliary columns. Auxiliary columns are needed to increase the table size. All columns are of the *Number* type. The table contains eight million records and test column has one hundred unique values.

To demonstrate how function-based indices could reduce execution time of queries, logarithmic function has been indexed. The query used for testing the logarithmic function is:

```
SELECT COUNT(*) FROM table_10 WHERE
    LOG(testcolumn + 1, 1000) = 3;
```

Without the index, the query execution time averaged 190,958ms or nearly 3 minutes and 11 seconds. Logarithmic function is a time-consuming function, but with a function-based index there is no need to process that function, only to access the index. With a B-tree index based on a function that is in the *WHERE* clause, time was reduced to an average of 35ms, and with a bitmap index based on the same function, it was reduced to an average of 19ms. It is about 5,456 times faster with the B-tree index and about 10,050 times faster with the bitmap index than without indices. The reason why the bitmap index had nearly 1.84 times better performances than the B-tree index is because of a *COUNT* function and few different values of the test column. Using the *COUNT* function in queries, Bitmap indices have better performances than B-tree indices in general. With this test, we wanted to show how function-based indices could prove useful in Data Warehouse systems, in which transactions are not usual and some functions are used frequently.

In the cases when there are nested functions in queries, we wanted to test if the DBMS could use indices based on the inner function only. Using these indices, we also tested performance of query execution time in comparison to the time using indices based on both functions or without indices. For the test we used a query with a *POWER* function and a nested *MOD* function:

```
SELECT COUNT(*) FROM table_10 WHERE
POWER(MOD(testcolumn, 11), 2) = 49;
```

In Figure 1, we present an average execution time of that query in the situations when there is no index, with a bitmap index based on a *MOD* function and with a bitmap index based on both *POWER* and *MOD* functions. It is shown that there is a possibility to use indices based on the inner functions only to reduce query execution time. That time is reduced further if there is an index based on all functions, which is a consequence of non-execution of the outer functions. In that case, there is only a need to access the index. In our test case, when using the index based on the inner function only, an execution time is about 36 times less than the execution time of the query without using indices. That means we don't have to create indices for every combination of functions used in our system, but only for some of them and others could be for inner functions only, which could save some memory in the system. Indices with inner function only could be reused in many queries with different outer functions. Using the index based on both functions, an execution time of the query is nearly 157 times less than execution time of the query without indices. It is better than the query execution time using the index with the inner function only, but in many cases improvement of 36 times could be enough.

We also wanted to prove with our test cases that the execution of functions requests time to process results. With the same table and the same data, we used similar query, but without functions:

```
SELECT COUNT(*) FROM table_10 WHERE
                    testcolumn = 40;
```

Without indices, an average execution time of that query was in average 4,515ms. It is 190ms less than the query execution time with *POWER* and *MOD* functions without indices, which was expected. It means that in summary *POWER* and *MOD* functions need approximately 190ms to execute. We also expect that the *POWER* and *MOD* functions separately needs less time than in summary. That could be shown from the previous results. For the same number of records and the same data,

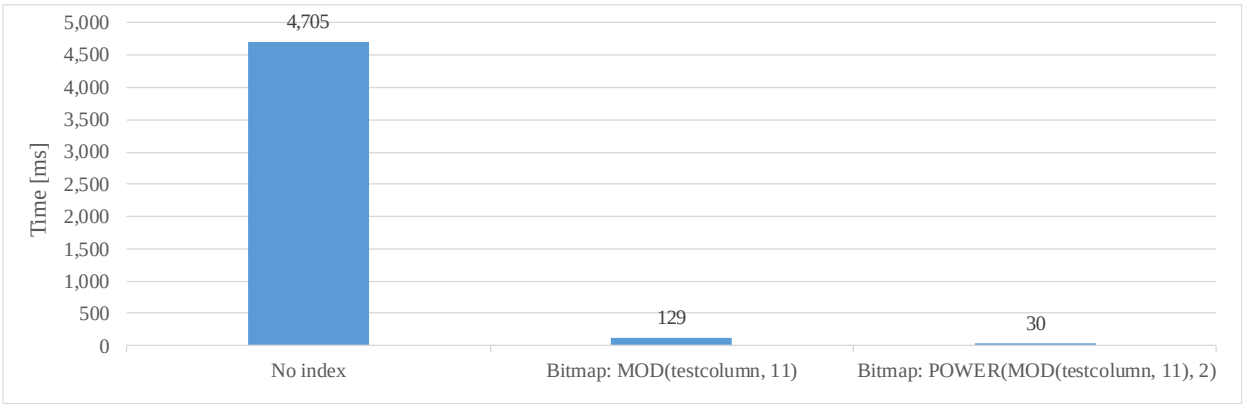

Figure 1. Results of tests on query with *POWER* and *MOD* functions in the *WHERE* clause

the query execution time using the index with both functions is 30ms and using the index with inner function only is 129ms. That means that *POWER* function needs approximately 99ms to execute. If we subtract that time from the execution time of both functions, the test results show that *MOD* function needs approximately 91ms to execute. These results are also something that was expected to be.

B. *Queries with substring function*

Queries with a substring function were tested on a single table having twelve columns. It has one column reserved for a primary key, one column for a testing purpose and ten auxiliary columns. Auxiliary columns are needed to increase the table size. All columns have a type of *Varchar2*, except of the column reserved for the primary key, which has a type of *Number*. The table contains from one to four million records and test column has 100,000 unique values.

We created an index based on a substring function and we had a query with a substring function in the *WHERE* clause that is a subset of the indexed substring function. We assumed that it will only be needed to access the index and process the results. To make it clear, we will present an example of creating bitmap index instruction:

```
CREATE BITMAP INDEX ind_str ON table_str
        (SUBSTR(testcolumn, 2, 4));
```

Two test queries were created to prove our assumption. The first one is a subset of the indexed substring function and the second one is also the subset of the indexed substring function, but with included lower bound:

```
SELECT COUNT(*) FROM table_str WHERE
    SUBSTR(testcolumn, 3, 2) = 'ra';

SELECT COUNT(*) FROM table_str WHERE
   SUBSTR(testcolumn, 2, 3) = 'Cra';
```

When we tested an execution time of these queries, we got opposite results of what we assumed. The Oracle RDBMS could not use index access only to process the results, but it also had to access the table to get the results. That is the reason why the execution time of both queries were about 14 times longer when using the created bitmap index than the execution time without using indices. It is only needed to access the table in cases without using indices. Similar test results were obtained in [8], when the same query was executed, but in the one test case it was needed only to access the index and in the second test case, it was also needed to access the table.

Nearly the same results were obtained in the next two test cases. The third tested query of this subsection is without a substring function, to have superset of the indexed substring function:

```
SELECT COUNT(*) FROM table_str WHERE
        testcolumn = 'ECra1D4z';
```

The fourth tested query is with a substring function in the *WHERE* clause that doesn't have any intersecting points with the indexed substring function:

```
SELECT COUNT(*) FROM table_str WHERE
    SUBSTR(testcolumn, 7, 2) = '4z';
```

After the results of the first two tests from this subsection, we expected similar results to be obtained from the last two tests. With the last two tests, we wanted to prove that the DBMS can't use indices if the substring function is not the same. It is only left to test a query execution time if in the *WHERE* clause is the same indexed substring function:

```
SELECT COUNT(*) FROM table_str WHERE
  SUBSTR(testcolumn, 2, 4) = 'Cra1';
```

We tested that query with one, two and four million records in the table and with a B-tree index, bitmap index and without indices. In Figure 2. we presented an average execution time of that query with the B-tree index and bitmap index for different number of records. The test case without indices is not presented in the Figure 2. because of a big difference between values and we wanted to do some additional analyses with the results when using indices. Without indices, an average execution time was 3,382ms, 6,638ms and 13,232ms for one, two and four million records respectively. When we used the same substring function as it is in the indices, we can see an improvement of the query execution time. The reason is because there is only need to access the index and not to access the table. If we analyze presented results, we can see that the number of records has influence on query execution time without using the indices, which is something that is expected, but it doesn't influence much when indices are used. That is because the *COUNT* aggregation function only needs to count the number of records in the index. The B-tree index shows a bit better performance, because there are many unique values of the indexed attribute in comparison to the number of records in the table.

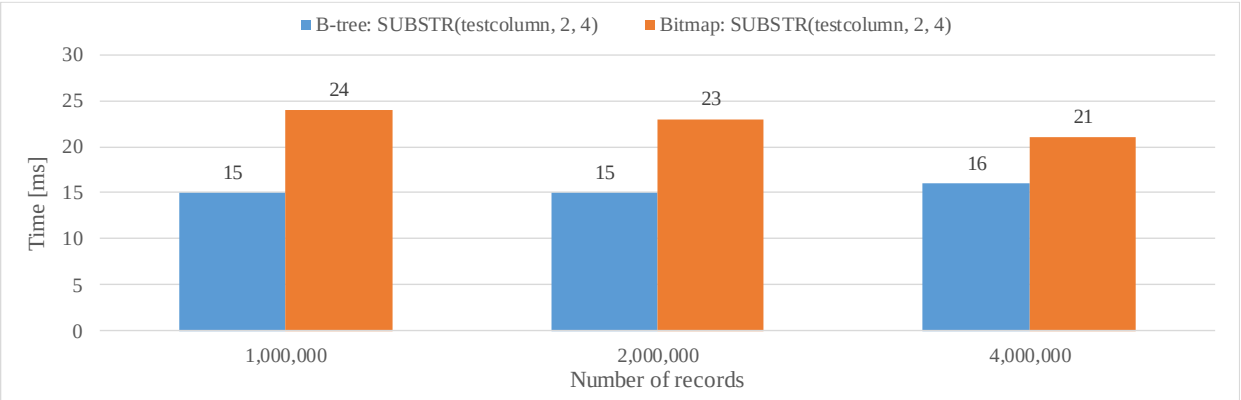

Figure 2. Results of tests on a query with *COUNT* aggregation function and the same indexed substring function in the *WHERE* clause

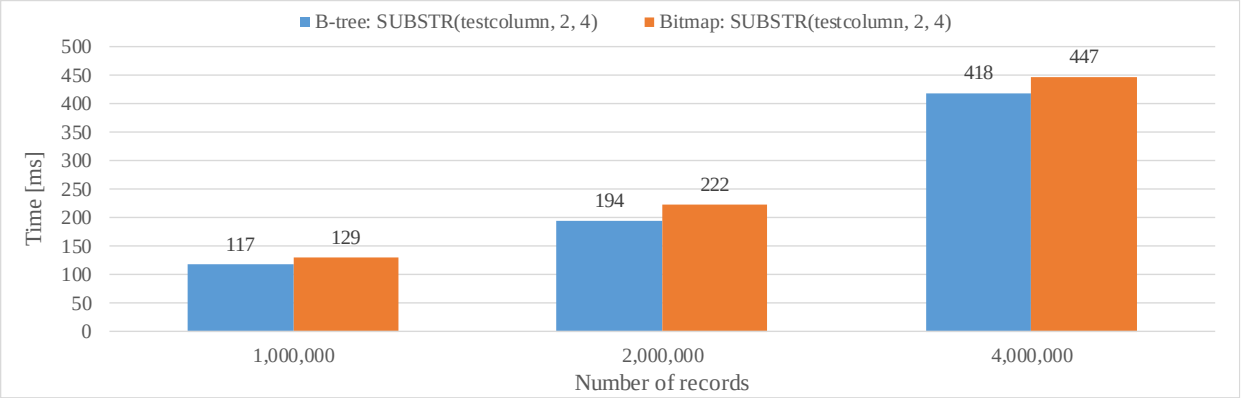

Figure 3. Results of tests on a query with *MAX* aggregation function and the same indexed substring function in the *WHERE* clause

B-tree index wasn't a lot better of bitmap index in our test case, like it was in [7], because in that study there was a need to access the table to get all attributes that weren't indexed by the bitmap index and they were all indexed by the B-tree index. Using that B-tree index, only the index access was needed and that is the reason why B-tree index was much better.

If there is other aggregation function that needs to access the table to obtain query results using indices, we assumed that the number of records would influence the query execution time more. We tested the same queries as the last five, but instead of using the *COUNT* aggregation function, we used *MIN* and *MAX* aggregation functions on the whole test column. Similar results were obtained with *MIN* and *MAX* functions, so we will present results with the *MAX* function only.

In comparison with the initial four equivalent queries with the *COUNT* aggregation function, the results were similar. Then we created test query with *MAX* aggregation function and substring function in the *WHERE* clause that is the same as indexed function:

```
SELECT MAX(testcolumn) FROM table_str
WHERE SUBSTR(testcolumn, 2, 4) = 'Cra1';
```

Without using the indices, the results were like those of the same query with the COUNT aggregation function. In Figure 3. we present an average execution time of the last query using the B-tree index and the bitmap index. There is a difference between those results and the equivalent ones with the *COUNT* function. The influence of the number of records is nearly linear to the query execution time. This is because of the *MAX(testcolumn)* function. We need the maximum of a filtered test column and in the index is only stored the part of the test column values, so it is needed to access the table. It is still in average 32 times better performance than the test case without the indices.

In the last test case, the *WHERE* clause would filter 1 record per 100,000 records in the table. That is the reason why the B-tree index provided better performance than bitmap index for different number of records in the table. We discussed that B-tree indices are better when there are many different values of an indexed attribute and that bitmap indices are better when there are not many different values. In the last test case, if we increase the number of records that will be filtered by the *WHERE* clause, maybe then the bitmap index would prove better performance than the B-tree index. We fixed the number of records in the table with 2 million records and we changed selectivity of the queries.

We had three different queries on the table with 2 million records:

```
SELECT MAX(testcolumn) FROM table_str
WHERE SUBSTR(testcolumn, 2, 4) = 'Cra1';

SELECT MAX(testcolumn) FROM table_str
WHERE SUBSTR(testcolumn, 3, 2) = 'ra';

SELECT MAX(testcolumn) FROM table_str
WHERE SUBSTR(testcolumn, 3, 1) = 'r';
```

The *WHERE* clause of the first, second and third query will filter 1, 19 and 1,603 records per 100,000 records in

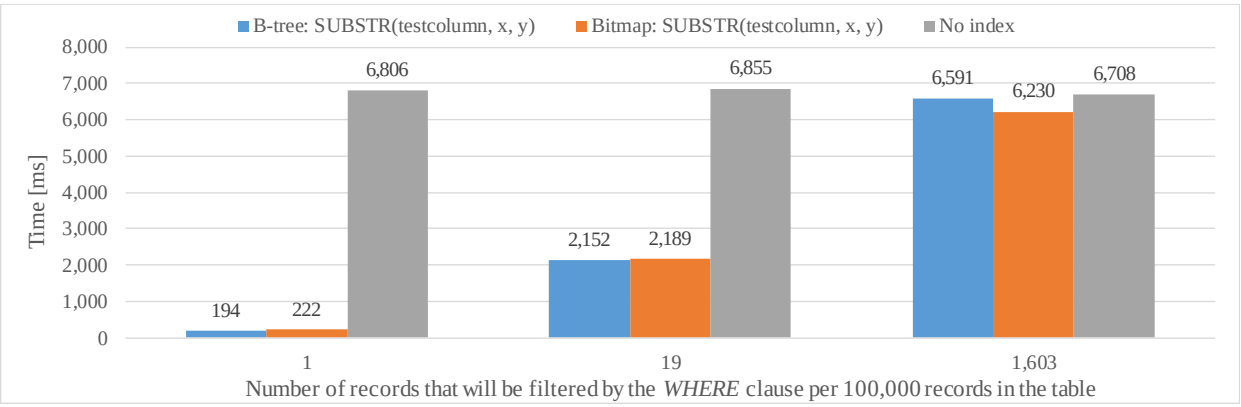

Figure 4. Results of tests on queries with *MAX* aggregation function and the same indexed substring function in the *WHERE* clause for different selectivity

the table respectively. We also created the B-tree index and the bitmap indexed based on functions that are in the *WHERE* clause of those queries. In Figure 4. we present an average execution time of those queries using the B-tree and bitmap indices and without using indices. We can see that the increase of the index selectivity requires more execution time of the queries for both indices. Without indices there isn't notable selectivity increase influence. That means at some point indices would impair performance of the query execution, which was also noticed in [6, 7]. As the selectivity increases, we can also notice that the bitmap index slightly gets better of the B-tree index, as it was expected.

From the results of the last test, we can conclude that indices are not always useful. Similar conclusion could be derived from the tests with the bitmap index and queries in which the substring function was not the same as the indexed one. It is the best if there is a way to only access the index. If there is table access also, it could prolong the query execution time.

VI. CONCLUSIONS

In this paper, we measured different query execution time using function-based bitmap and B-tree indices or without using any index. Our goal was to test if the DBMS can use indices with nested functions and indices with different substring functions.

The Oracle DBMS can use indices based on the inner function only to process results of a query, which proved our assumption. The performance was much better than without using any index. That means that we don't have to create indices with all combinations of functions that are usually needed for reports. One index with the inner function could be used in many reports in the combination with other outer functions. That could save a lot of memory in the system.

We also indexed part of the string attribute in a table and tried to use that index with queries that have a substring function that represent subset, superset or difference of the indexed string. The Oracle DBMS couldn't use that index, which wasn't what we expected, but it could use the index in the case when there was the exact indexed substring function. We also presented results of selecting different aggregation functions: *COUNT* and *MAX*. If there isn't table access, like when we used the *COUNT* aggregation function, the results showed that the number of records in the table doesn't influence much query execution time. If there is both index and table access, like when we used the *MAX* aggregation function, an increase of the records number or an increase of the selectivity can influence the query execution time. At some point, using indices would not be beneficial. That means not all indices would improve query execution time and we need to make sure that any created index will not impair performance.

All those results could make guidelines to decide which index could reduce an execution time of some queries more and how to balance between the execution time and a memory usage. That could be useful for many companies, because there is a lot of needed reports that could include large amounts of data.

Analyzing the result data obtained during the research, we found new questions that need to be answered. We stated that memory could be saved by indexing inner function only and not indexing outer function. In future works, we will measure how much memory space could be saved that way. We will also test if another DBMS could use indexed substring function to improve execution time of the queries with different substring functions. The Oracle DBMS couldn't use those indices, but there is possibility that other DBMSs could. In the last test case, we noticed that when there are many records obtained from a database, indices could impair performance of the query execution and the full table scan could be better option. We will try to find at which point the indices will not be useful, so we could know when there is no need to create them.

ACKNOWLEDGMENT

The research in this paper is supported by the Ministry of Education and Science of the Republic of Serbia, project III-44010.

REFERENCES

[1] M. Chen, S. Mao, and Y. Liu, "Big Data: A Survey," Mobile Networks and Applications, vol. 19, no. 2, pp. 171–209, Apr. 2014.

[2] A. Oussous, F.-Z. Benjelloun, A. Ait Lahcen, and S. Belfkih, "Big Data technologies: A survey," Journal of King Saud University - Computer and Information Sciences, vol. 30, no. 4, pp. 431–448, Oct. 2018.

[3] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," ACM SIGMOD Record, vol. 26, no. 1, pp. 65–74, Mar. 1997.

[4] M. Golfarelli and S. Rizzi, "From Star Schemas to Big Data: 20+ Years of Data Warehouse Research," in A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years, vol. 31, S. Flesca, S. Greco, E. Masciari, and D. Saccà, Eds. Cham: Springer International Publishing, 2018, pp. 93–107.

[5] A. Gani, A. Siddiqa, S. Shamshirband, and F. Hanum, "A survey on indexing techniques for big data: taxonomy and performance evaluation," Knowledge and Information Systems, vol. 46, no. 2, pp. 241–284, Feb. 2016.

[6] J. M. Medina, C. D. Barranco, and O. Pons, "Evaluation of Indexing Strategies for Possibilistic Queries Based on Indexing Techniques Available in Traditional RDBMS: INDEXING STRATEGIES FOR POSSIBILISTIC QUERIES," International Journal of Intelligent Systems, vol. 31, no. 12, pp. 1135–1165, Dec. 2016.

[7] J. M. Medina, C. D. Barranco, and O. Pons, "Indexing techniques to improve the performance of necessity-based fuzzy queries using classical indexing of RDBMS," Fuzzy Sets and Systems, vol. 351, pp. 90–107, Nov. 2018.

[8] M. Kvet and M. Vajsova, "Performance study of the index structures in audited environment," in 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), Barcelona, Spain, 2016, pp. 459–464.

[9] P. O'Neil and D. Quass, "Improved query performance with variant indices," in ACM SIGMOD international conference on management of data (SIGMOD), Tucson, USA, 1997, pp. 38-49.

[10] M. Jurgens and H.-J. Lenz, "Tree Based Indices vs. Bitmap Indices: A Performance Study," International Journal of Cooperative Information Systems, vol. 10, No. 03, pp. 355-376, 2001.

[11] Database Concepts Indices and Index-Organized tables, https://docs.oracle.com/cd/E11882_01/server.112/e40540/indexiot.htm#CNCPT1170, last enter February 2, 2019.

[12] P. Vennapusa, "Oracle Database 10g: SQL Tuning", vol. 1, Student Guide, Edition 2.0, December 2006.