REPORT

Project:

DATABASE B-TREE INDEX PERFORMANCE

Author:

Marcin Rabiza

Table of contents:

1. Introduction

An index is a data structure that increases the speed of performing search operations on a table. Using indexes allows you to narrow down the number of records to be searched, instead of going through the whole table.

If a table is stored as a linked list, finding specific item is associated with traversing the list until it is found, what determines algorithm's time linear complexity *O(n).* Using separate data structures such as indexes allows to reduce time complexity to logarithmic one *O(log n)* by sacrificing some space for index storage.

Such an index can have a form of a binary search tree - it has a root and elements stored in separate branches. Having data stored in that structure is it easier to locate an element if the one does not want to traverse the whole tree. However, if one of the branches have more elements than the others the tree becomes unbalanced. The worst case scenario with unbalanced trees is the one still have to traverse all the nodes (eg. one long branch) to find specific record.

In order to optimize the search, the solution is to add new elements to the tree in a way that the height of a tree remains the same for all branches. Due to fact, that disk access is, in general, expensive, several elements are stored as a node in structure called *page*. When traversing the tree, the one access the disk only to receive the pages instead of individual elements. The page has a fixed maximus size, so when a new element is added to the page and the page becomes oversized, it is splitted in order to form another branch and the tree remains balanced.

This generalization of a binary search tree in that a node can have more than two children is called a ***B-tree***. Balancing a B-tree is a cost the one have to pay with every insert, but in worst case scenarios it still performs in logarithmic time.

Using B-tree index increases overall performance in certain cases. For example, adding an index to a Boolean column indicates a tree with two large branches. In that case, a storage engine still have to traverse a list of pages, what makes using the index pointless. On the other hand, adding an index to a column containing very different data like IDs or random numbers it will result in a tree that has to be rebalanced often, since the insert goes to very different pages causing them to split. Therefore, when doing more inserts on a random data a primary key B-tree index is not a feasible solution either. Having that in mind, it should be noted that B-tree indexes work best for:

- Columns frequently used in predicates
- Columns used to join tables
- Columns with high cardinality
- Columns rarely updated
- OLTP systems.

2. Tables and Queries

For this report, three tables of different sizes but of the same structure have been used. The tables were generated using designed SQL script in SQL*Plus query tool.

Tab. 1. Sizes of a tables used for the report.

TABLE SIZE	NUMBER OF RECORDS
Small	2 000
Medium	200 000
Large	20 000 000

Each table contains following types of attributes:

- *integer,*
- *float,*
- *date,*
- *char(8)* - short string of a fixed length of 8 letters,
- *char(32)* - long string of a fixed length of 32 letters,
- *varchar2(8)* - short string of a variable length up to 8 letters,
- *varchar2(32)* - long string of a variable length up to 32 letters.

Tables were filled with data using random numbers generators in a following way:

- with a random int number from 1 to 10 000,
- with a random float number from 1000 to 100 000,
- with a random date (*sys_date – random value*)
- with one of the randomly generated, but semantically correct 8-letter words starting on a letter "m": *montcalm, misology, maleness, mcnamara, manichee, melchers, manuring, marjoram, moderate, menthene*.
- with one of the randomly generated, but semantically correct up to 8-letter words starting on a letter "m": *mikado, murcia, mum, media, merk, misce, melville, monsieur, metanira, modeller.*
- with one of the randomly generated 32-letter words starting on a letter "m": *microelectronicsmicroelectronics, mechanotherapistmechanotherapist, meteorologicallymeteorologically, maladministratormaladministrator, machiavellianismmachiavellianism.*
- with one of the randomly generated up to 32-letter words starting on a letter "m": *multituberculatemultituberc, malapportionmentmalapportionmen,*

multiflagellatedmultiflagellat, municipalizationmunicipalizat, magnetogeneratormagnetogener.

Following queries have been implemented to test experimentally:

Select 1
```
select count(*) from TABLE;
```

Select 2
```
select * from TABLE where INTEGER_NUMBER < 10;
```

Select 3
```
select * from TABLE where RAND_DATE between '14-FEB-2018' and '24-FEB-2018';
```

Select 4
```
select * from TABLE where BIG_FIXED_STRING = 'microelectronicsmicroelectronics';
```

Select 5
```
select * from TABLE where SMALL_VARIABLE_CHAR = 'murcia';
```

Insert
```
insert into TABLE
(INTEGER_NUMBER,FLOAT_NUMBER,RAND_DATE,SMALL_FIXED_STRING,BIG_FIXED_STRING,SMALL_VARIABLE_CHAR,BIG_VARIABLE_CHAR)
   values (501,321.12,'14-FEB-2020', 'moderate','machiavellianismmachiavellianism', 'mikado',
'municipalizationmunicipalizat');
```

Update
```
update TABLE set SMALL_VARIABLE_CHAR = 'moderate' where SMALL_FIXED_STRING='moderate';
```

Delete
```
delete from TABLE where RAND_DATE='14-FEB-2020';
```

During the experiment indexes were placed on each attribute. The queries have been tested with and without indexes separately.

3. Experiments and results

A series of experiments have been conducted on a generated data. The experiments involved using another SQL script for running different queries presented in a chapter 2. The same script was also used for setting time measurements, index create/drop orders and saving output data to two different TXT files – one for non-indexed tables and one for indexed tables.

The first part of the experiment was aimed at planned (according to Execution Plan) and elapsed time comparison for different queries. It should be noted, that in case of *insert* statement Execution Plan time was not generated. The results are presented on two separated figures below.

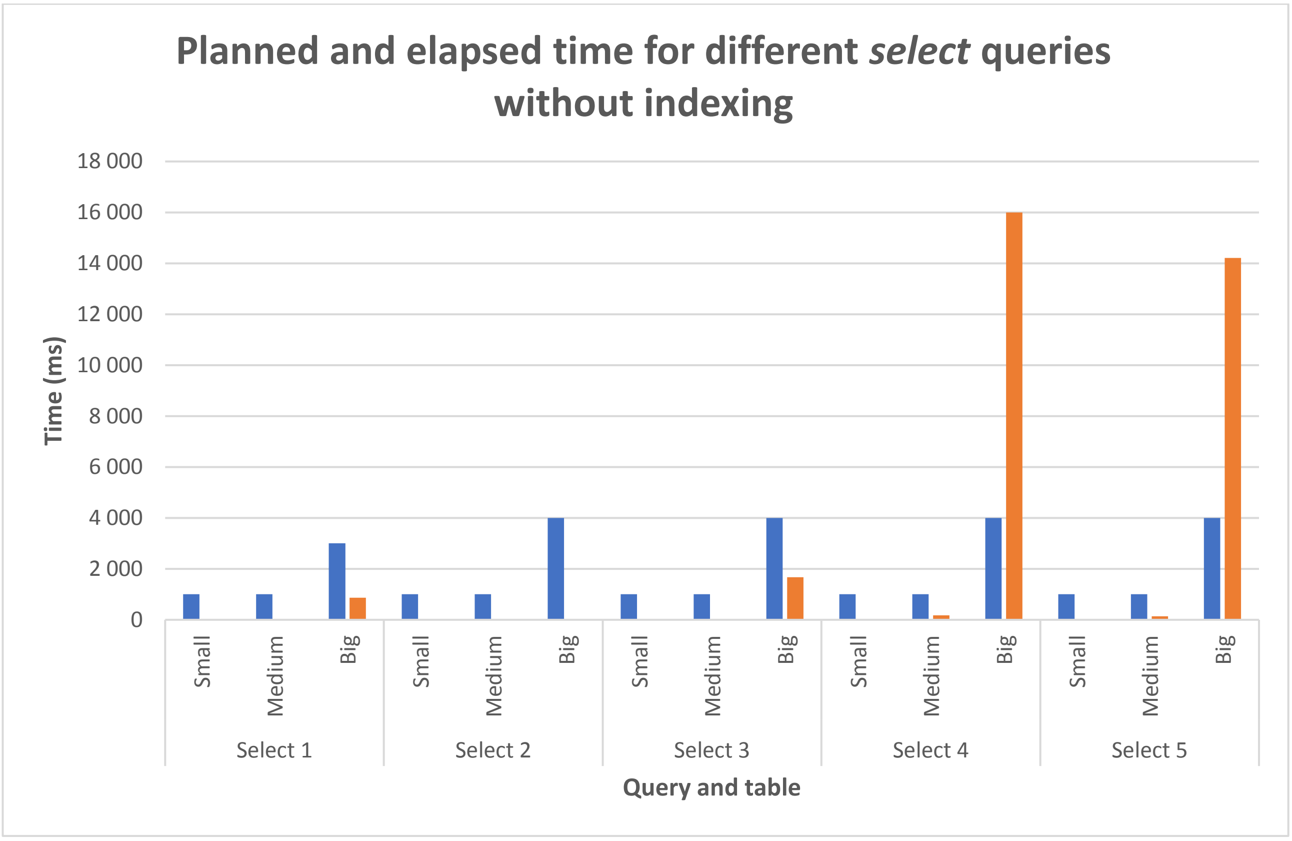


Fig. 3.1. Planned and elapsed time for different *select* queries for tables without indexes. Blue series – time planned accordingly to Execution Plan, orange series – elapsed time.

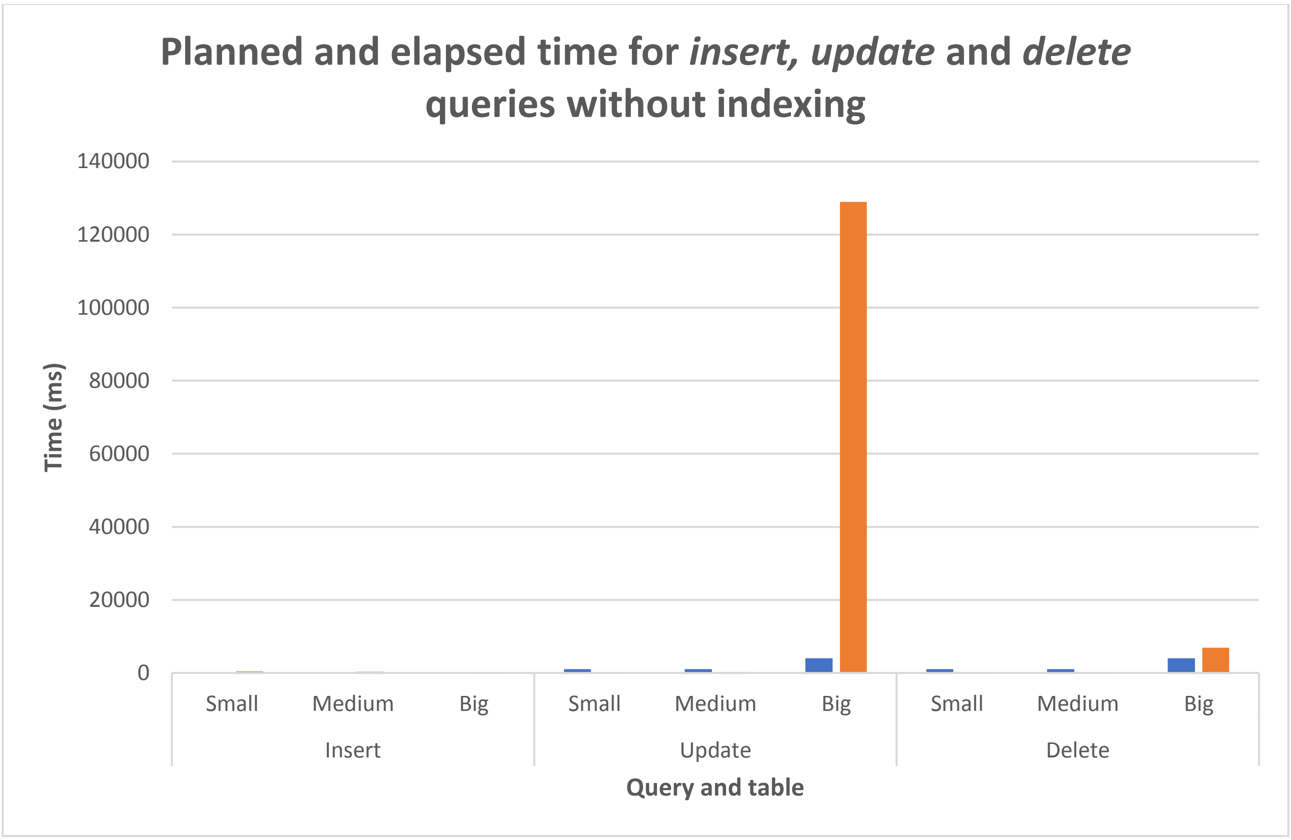


Fig. 3.2. Planned and elapsed time for *insert*, *update* and *delete* queries for different tables without indexes. Blue series – time planned accordingly to Execution Plan, orange series – elapsed time.

As may be seen, elapsed time of query execution tend to be much shorter than Execution Plan time for predicates concerning numerical attributes such as *integer*, *float* or *date*. However, execution time becomes longer than evaluated time for predicates concerning string attributes. The biggest difference between evaluated and elapsed time and the longest elapsed time may be observed for *update* statement for the 20-milion record table.

Figure 3.3 presented below shows that *inserting* tend to be very low-cost operation.

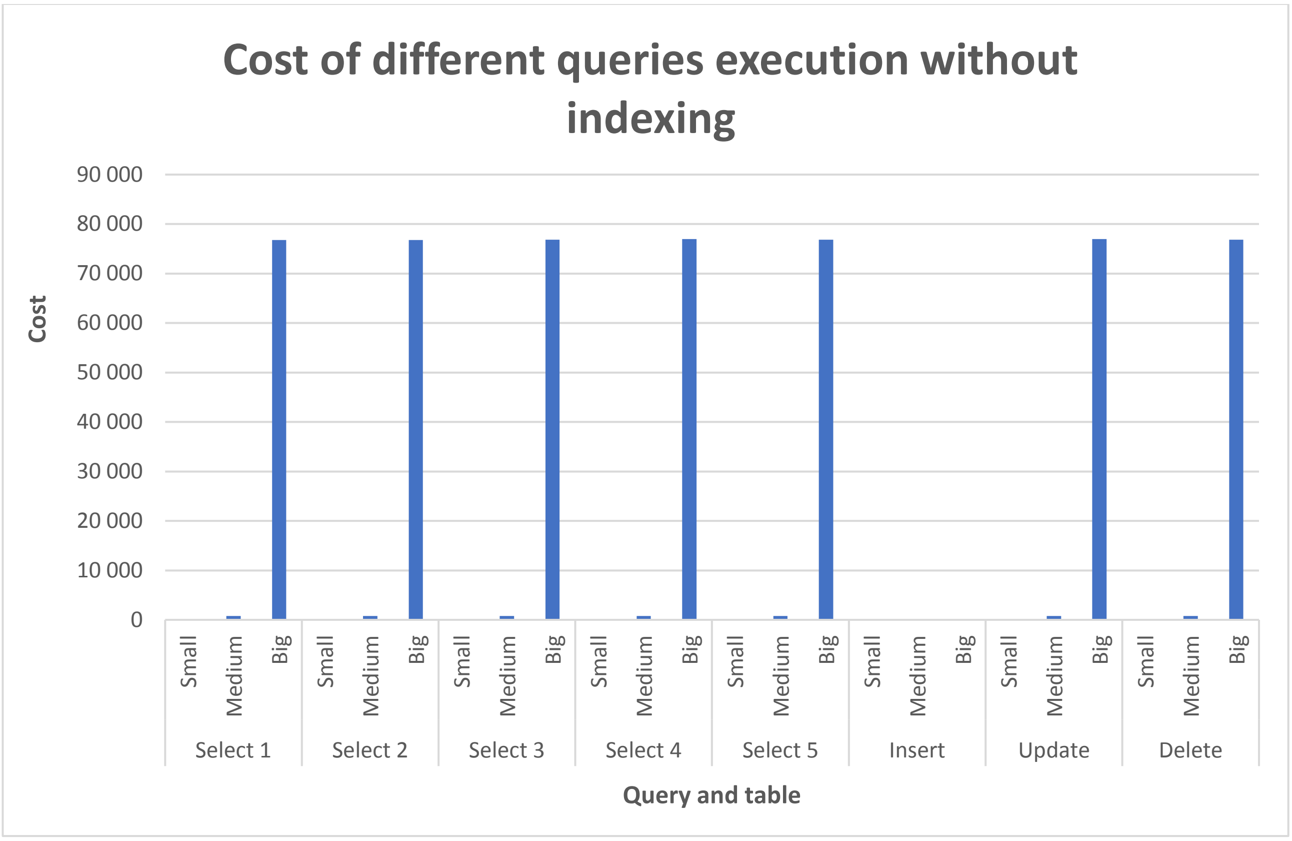


Fig. 3.3. Estimated cost of different queries execution for tables without indexing.

The above analyzes summarize the basic parameters of the dataset under study. As the next step of the experiment, the indexes were assigned to all tables columns. According to queries Execution Plans, indexes have been used for the following queries: *Select 2*, *Select 3*, *Select 4*, *Select 5* and *Delete*. Results concerning planned (according to Execution Plan) and elapsed time comparison for different queries for indexed tables are shown on two separated figures below.

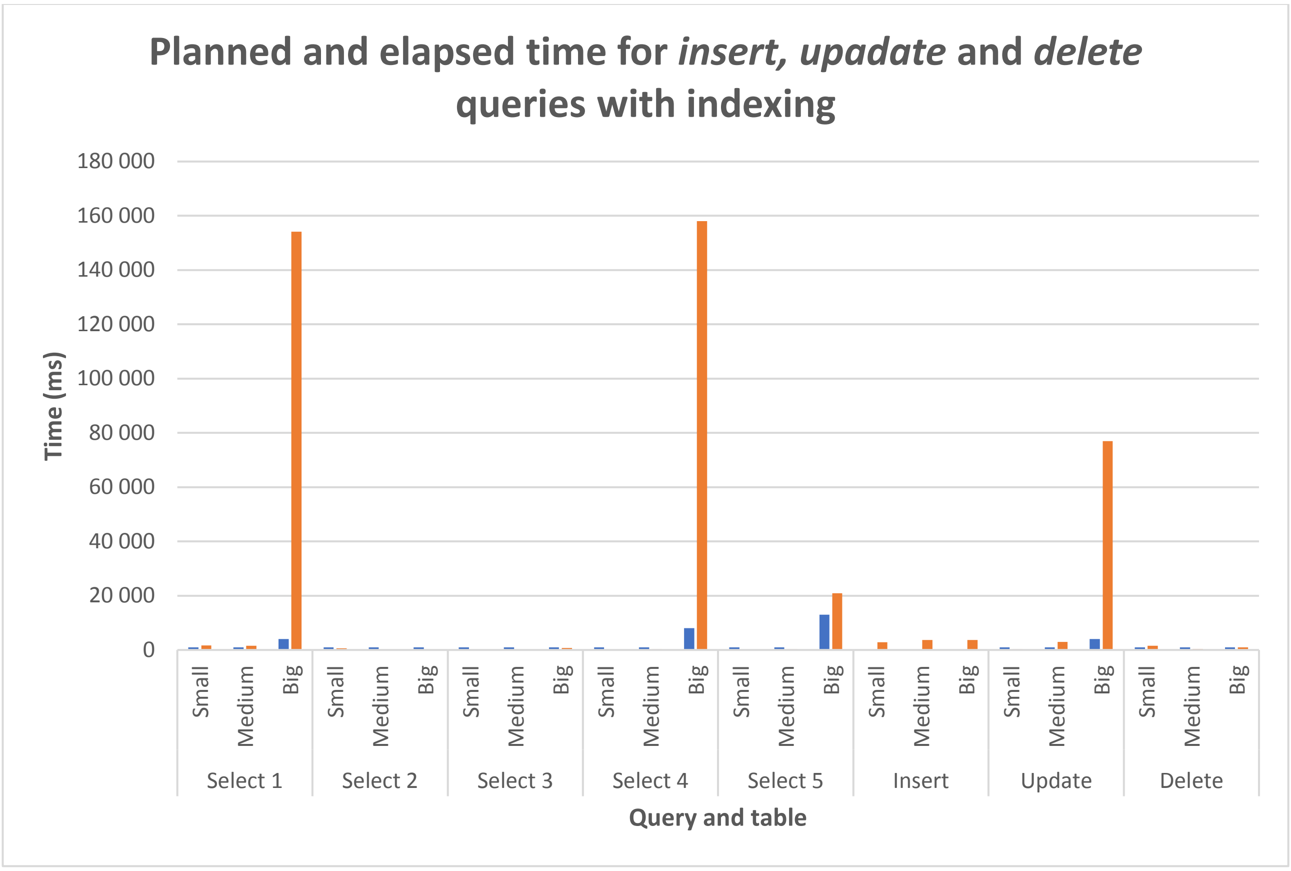


Fig. 3.4. Planned and elapsed time for *insert, update* and *delete* queries for different tables with indexed columns. Blue series – time planned accordingly to Execution Plan, orange series – elapsed time.

Figure 3.6 presented below shows that using indexes may significantly reduce cost of query execution depending on a type of attribute. However, for a few cases overall cost and time are higher while using indexes, what cannot be seen in a table, but is noticeable in a raw data.

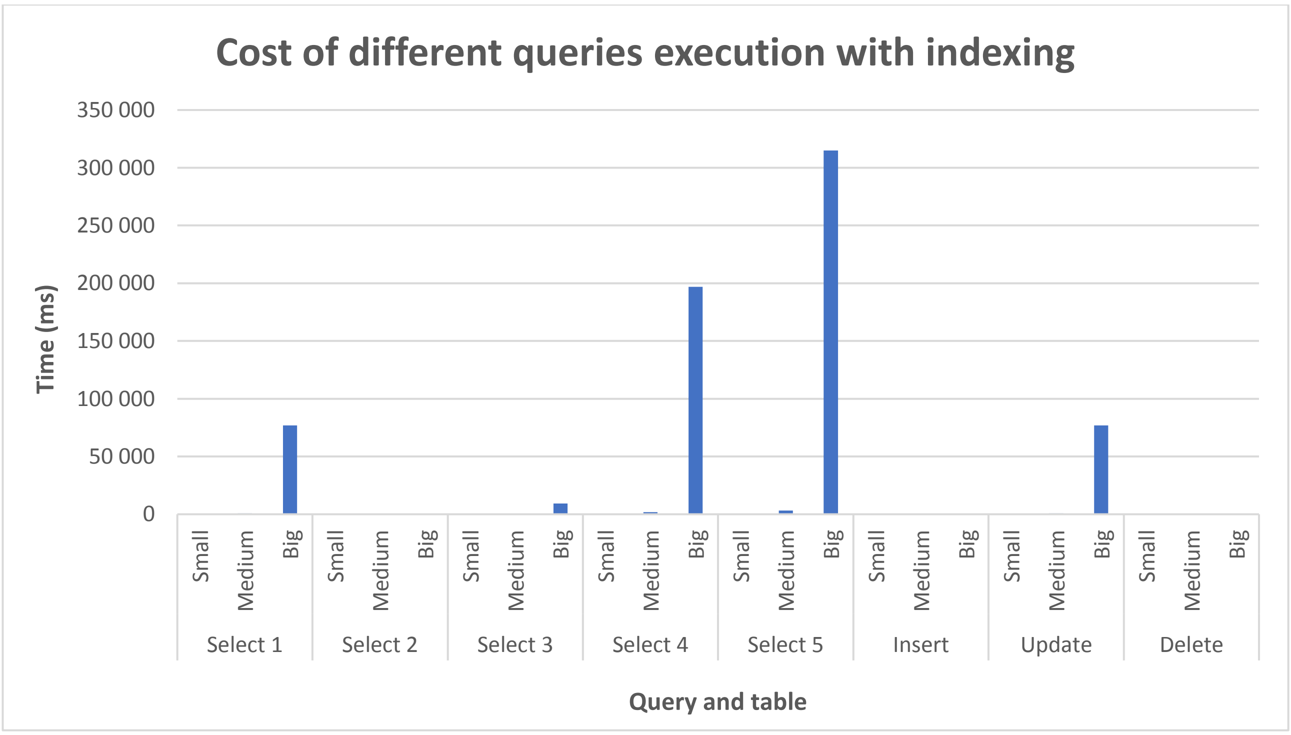


Fig. 3.5. Estimated cost of different queries execution for indexed columns.

The next part of the study is aimed at time and cost comparison for queries using indexes on indexed and non-indexed columns. The results are presented on two separated figures below.

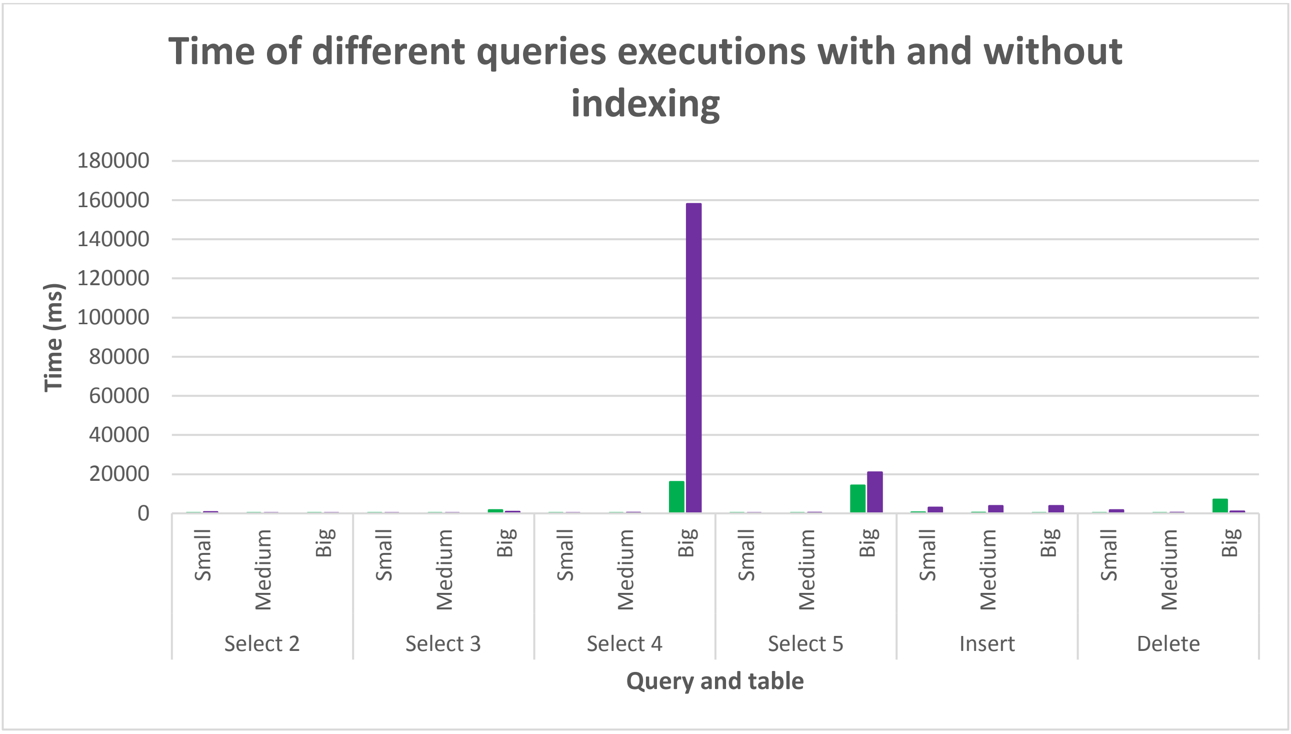


Fig. 3.6. Elapsed queries using indexes executions time with and without indexing. Green series – non-indexed columns, purple series – indexed columns.

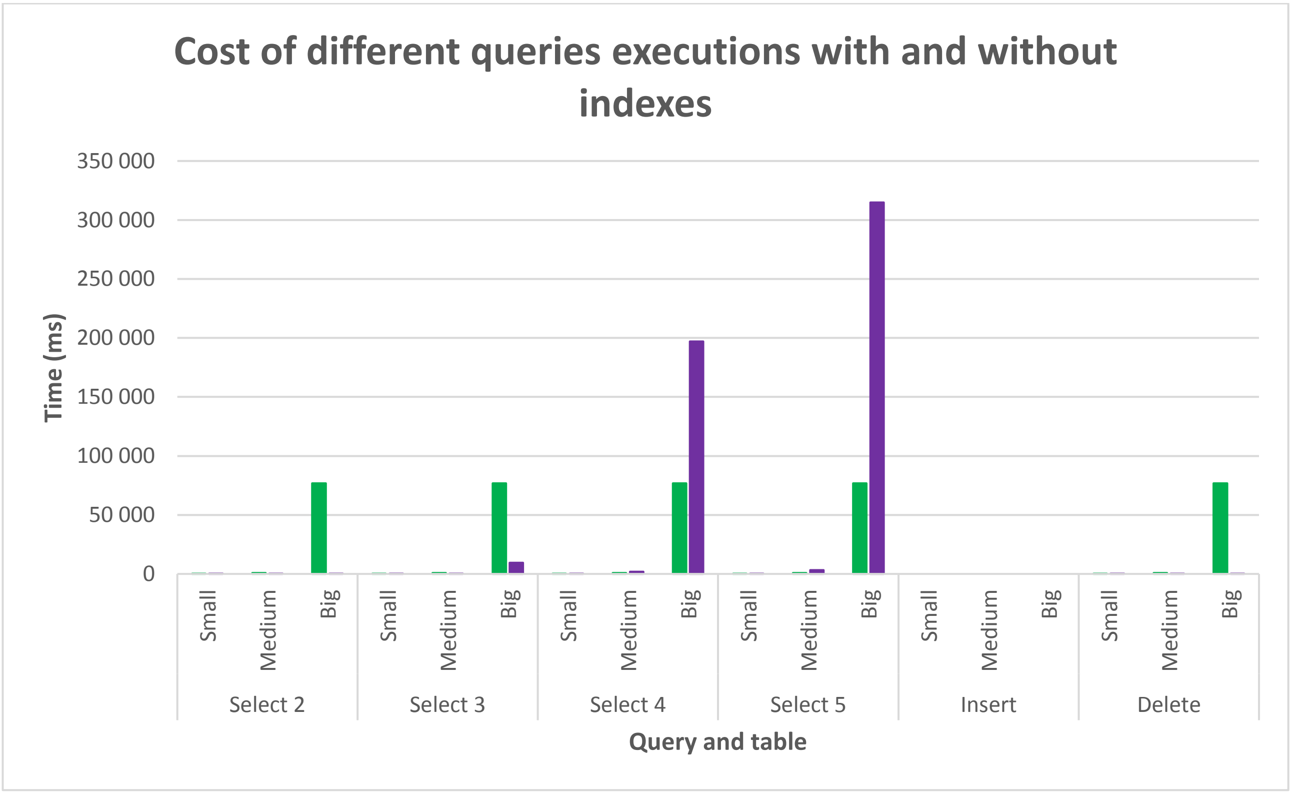


Fig. 3.7. Estimated cost of queries using indexes executions with and without indexing. Green series – non-indexed columns, purple series – indexed columns.

The last part of the study shows time statistics for creating and dropping indexes on different attributes.

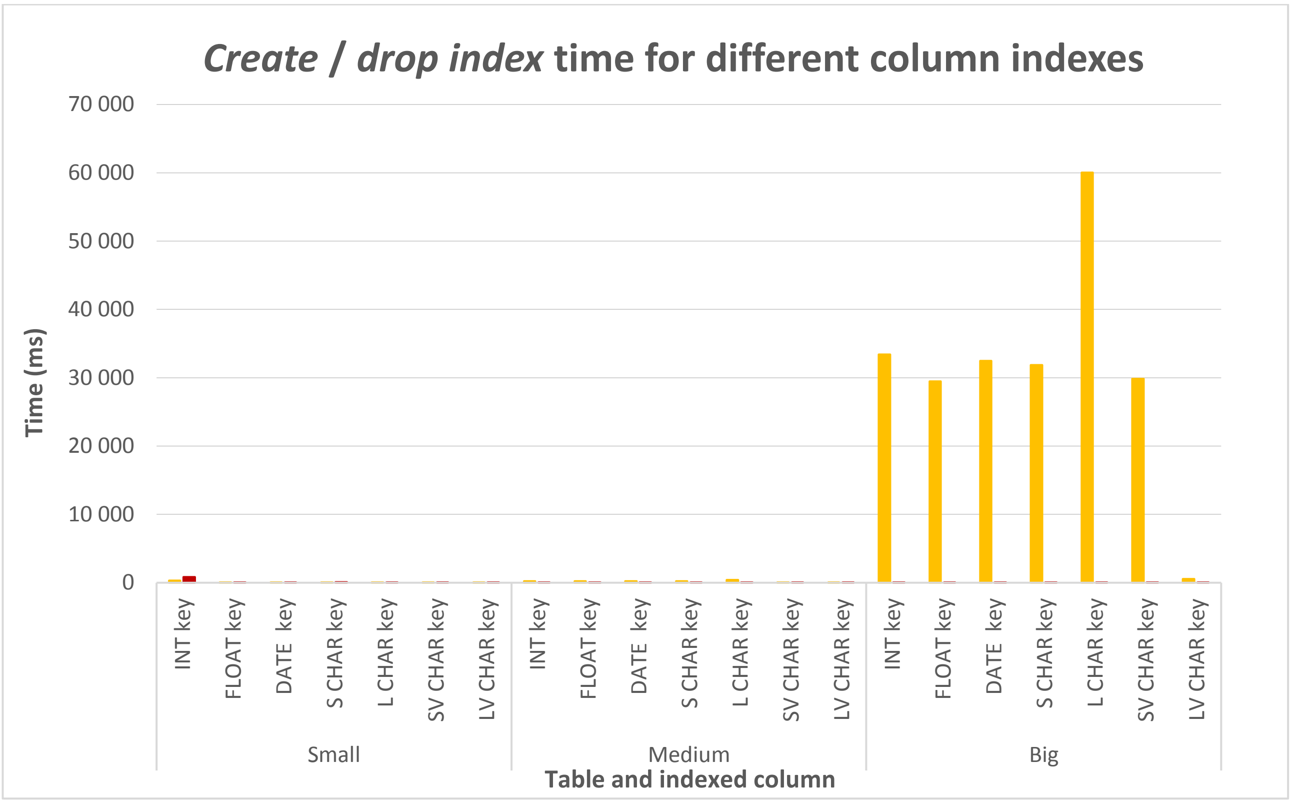
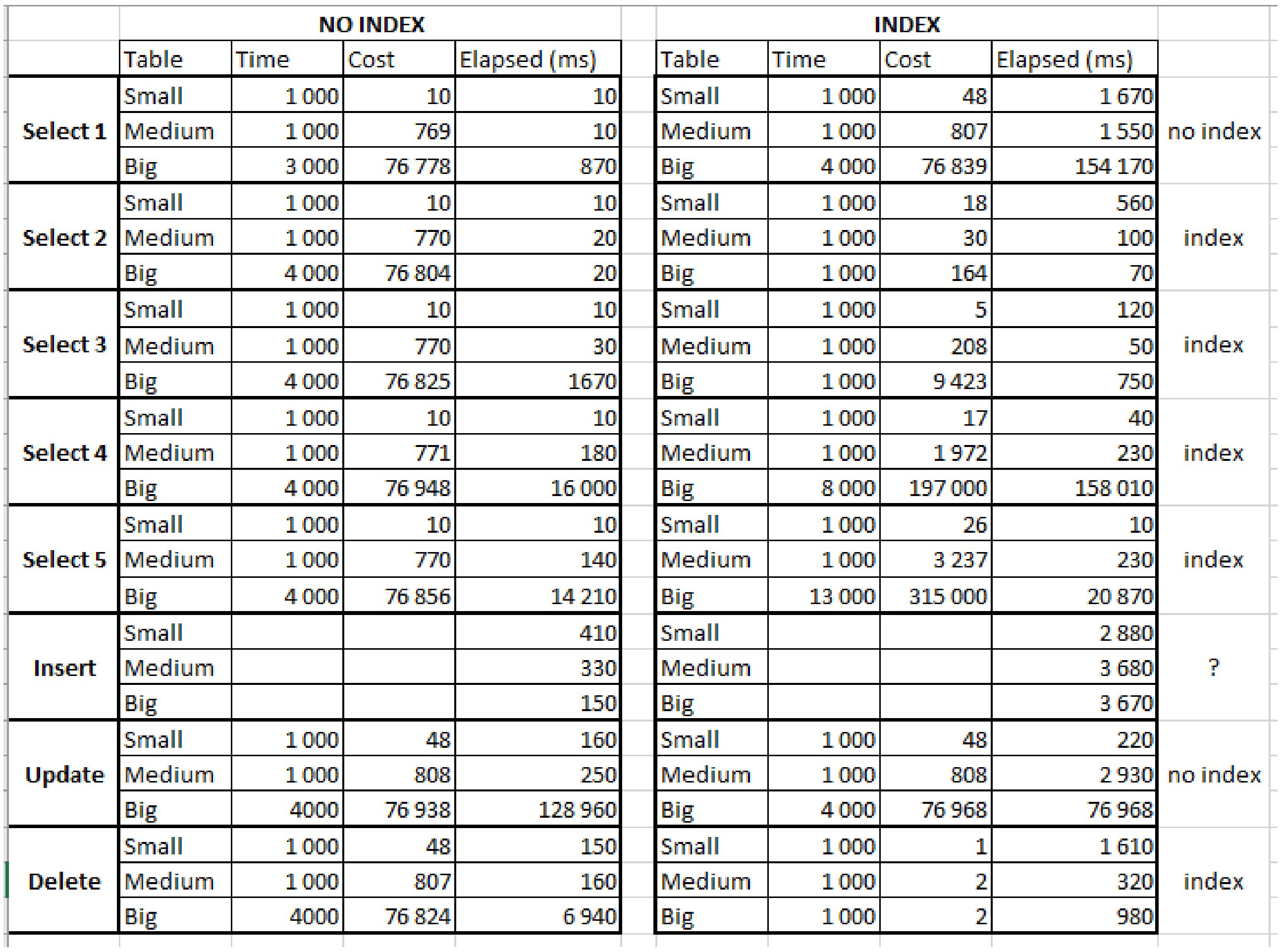


Fig. 3.8. Time statistics for creating and dropping indexes on different attributes.

Due to fact, that not all valuable information can be retrieved from the presented charts, it is also advisable to inspect the piece of a raw data presented on a figure below.



	NO INDEX				INDEX				
	Table	Time	Cost	Elapsed (ms)	Table	Time	Cost	Elapsed (ms)	
Select 1	Small	1 000	10	10	Small	1 000	48	1 670	no index
	Medium	1 000	769	10	Medium	1 000	807	1 550	
	Big	3 000	76 778	870	Big	4 000	76 839	154 170	
Select 2	Small	1 000	10	10	Small	1 000	18	560	index
	Medium	1 000	770	20	Medium	1 000	30	100	
	Big	4 000	76 804	20	Big	1 000	164	70	
Select 3	Small	1 000	10	10	Small	1 000	5	120	index
	Medium	1 000	770	30	Medium	1 000	208	50	
	Big	4 000	76 825	1670	Big	1 000	9 423	750	
Select 4	Small	1 000	10	10	Small	1 000	17	40	index
	Medium	1 000	771	180	Medium	1 000	1 972	230	
	Big	4 000	76 948	16 000	Big	8 000	197 000	158 010	
Select 5	Small	1 000	10	10	Small	1 000	26	10	index
	Medium	1 000	770	140	Medium	1 000	3 237	230	
	Big	4 000	76 856	14 210	Big	13 000	315 000	20 870	
Insert	Small			410	Small			2 880	?
	Medium			330	Medium			3 680	
	Big			150	Big			3 670	
Update	Small	1 000	48	160	Small	1 000	48	220	no index
	Medium	1 000	808	250	Medium	1 000	808	2 930	
	Big	4000	76 938	128 960	Big	4 000	76 968	76 968	
Delete	Small	1 000	48	150	Small	1 000	1	1 610	index
	Medium	1 000	807	160	Medium	1 000	2	320	
	Big	4000	76 824	6 940	Big	1 000	2	980	

Fig. 3.9. Raw data generated in the experiment.

4. Conclusions

1) Query optimizer estimated index-based query execution plans highly inaccurately. Real execution times were much shorter or longer than estimated ones, but for tested data it is impossible to point out the inaccuracy as the function of table size.
2) Selecting records with predicate using numerical attributes (Select 2 and 3) generally costs less while using indexes. As long as the data shows longer execution time on indexed *integer* column, it seems probable, that elapsed time for indexed columns decreases quickly while transversing bigger tables.
3) Indexes seems to perform better on tables of larger sizes.
4) Insertion performed on an indexed column turns out to be more time-consuming than without index – most likely because of the fact, that for each insertion, an index needs to be updated.
5) Indexing on string attributes (Select 4 and 5) had negative impact on overall query performance. It may be associated with the fact, that words filled into records were drawn from the array of 10 elements, what implies low cardinality of the attribute in a bigger tables.
6) Update statement takes no advantage of using indexes – it performs a full table scan.
7) Delete statement performed on an indexed column determines longer time for deleting records from small tables, but shorter time for deleting records from bigger ones. Furthermore, in case of deleting using indexes cost is significantly reduced.
8) Creating an index takes relatively long time. For small datasets, frequently updated or low-cardinality columns it decreases overall performance.