Prof. Dr. Stefan Brass
Institut für Informatik
MLU Halle-Wittenberg

January 12, 2017

Databases IIB: DBMS-Implementation
— Exercise Sheet 11 —

As requested by the students, the repetition questions a) will not be discussed in class unless somebody asks for the solution to a specific question. So please have a look at them before the meeting and decide which questions do you wish to be discussed. Of course, you can also ask any question of your own on the topics of the course.

We will work on Part b) to f) in the lab session, you only have to submit solutions to the homework exercises, i.e. Part g) to k). The official deadline is January 17, 12:00. (before the problem/lab session).

Repetition Questions

a) What would you answer to the following questions in an oral exam?

- What is the difference between a binary search tree and a B-tree?
- Why does one store many search key values in a node of a B-tree? How is the maximal number determined?
- Why is a B-tree balanced? What are the exact conditions? What is the worst possible case, and what sequence of insertions gives this B-tree? Why does one not require perfect balancing?
- What is the complexity of searching in a B-tree? What is the complexity of insertion?
- What is the difference between a B-tree and a B^+-tree? Why do most database systems use B^+-trees?
- Explain the algorithm for insertion in a B^+-tree (for simplicity, assume that it is a unique index, and the search key values have all the same size).
- Give an algorithm for creating a B^+-tree on an attribute for a given table (of course, one can do this as many insertions on an empty B^+-tree, but there is a better way).
- For what operations in relational algebra can a B^+-tree be used? You can also give typical SQL queries and explain how they are evaluated with a B^+-tree index.

- What is the difference between an index on the column combination (A, B) and an index on the column combination (B, A)?
- Suppose you have two indexes, one on A, and one on B, and the query contains equality conditions for them with known constants, e.g. $A = 1234 \wedge B = 56789$. How can you use both indexes?
- Compare a full table scan with an access to a table via an index and looking up the tuples for the ROWIDs. What does one have to know about the relation to decide which query evaluation plan is better? Given an example where using the index is worse than doing a full table scan.
- What is an index-only query evaluation plan? Given an example for a query that can be evaluated in this way.
- What are disadvantages of indexes? In which cases you should not define an index on a column of a relation?

In-Class Exercises

b) Create a table `R` with 100.000 rows and the following columns:

- an integer as a key attribute `A`,
- a second column `B` that contains also integer values, where you repeat each value 10 times, with the copies in consecutive tuples,
- a third column `C` like `B`, but with the copies being far from each other (in different blocks),
- and a fourth column that is a fixed string of length 10.

Modify a PL/SQL procedure we have previously used for creating larger tables. Give the primary key a name (it will be used as index name). Do not delete this table, it will also be used next week when we look at query evaluation plans.

c) Create an index on `B`, and anthor index on `C`. Do an `ANALYZE TABLE`.

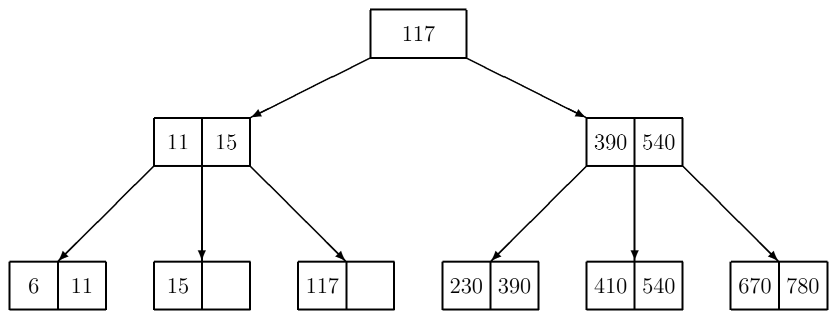
d) Estimate the size of the table and check that your estimate is approximately right. Check also the length of the tuples that you computed.

e) Have a look at the data in the data dictionary tables `IND` (short for `USER_INDEXES`) and `USER_IND_COLUMNS`. Find the data of your indexes.

f) Write a query that prints the names of all indexes on your table `R` together with the number of block accesses that will be needed to evaluate a selection with a given constant for the index attribute. You will need the columns `BLEVEL` (hight of the B-tree minus 1), `AVG_LEAF_BLOCKS_PER_KEY`, and `AVG_DATA_BLOCKS_PER_KEY`.

Homework Exercises (Homework 11A)

Consider the following B^+-tree of height 3 with maximally 2 entries per node:



What is the resulting B^+-tree after the following operations? Please use the given B^+-tree as input for each operation, and not the result of the previous operation.

g) Insertion of 13.

h) Insertion of 9.

i) Insertion of 290.

j) Deletion of 15.

Homework Exercises (Homework 11B)

k) Let us write a small simulation to illustrate the effect of the random accesses to rows on the number of blocks that must be read from disk. You are supposed to write this program in C++, but I will also accept solutions in Java. Note that this program is not part of the project, it is standalone and small (definitely below 100 lines).

Suppose that a table has 100 000 rows, and each block contains 10 rows, i.e. there are 10 000 blocks. The first block contains rows 0 to 9, the second block contains rows 10 to 19, and so on.

Now we want to access 1% of the rows at random, i.e. you use a random number generator 1000 times to select a row number between 0 and 99 999. You will need a boolean value for each of the blocks that shows whether the block contains at least one selected row. Thus, you use a boolean array of size 10 000. The entries of array must be initialized to "false". For each randomly selected row, you determine the block number in which the row is stored (by simply dividing the row number by 10). Then you set the boolean value corresponding to that block to "true".

In the end, you count how many of the blocks where touched, i.e. how many entries in the array are “true”, and print this as a percentage of the total number of blocks.

It would be good if you define constants such that you can easily change the parameters, especially the number of rows per block. However, this is not required.

The classical function for generating random numbers is `std::rand` declared in the header file `<cstdlib>`. It returns integer numbers between 0 and some large number `RAND_MAX`. See e.g.

[http://en.cppreference.com/w/cpp/numeric/random/rand].

We do not need high quality random numbers, therefore this might be enough.