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

January 27, 2017

Databases IIB: DBMS-Implementation — Exercise Sheet 13 —

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.

You only have to submit solutions to the homework exercises, i.e. Part c) to h). The official deadline for Homework 13A is January 31, 12:00 (before the problem/lab session). For Homework 13B and 13C, it is February 16.

Repetition Questions

a) What would you answer to the following questions in an oral exam? (Note that also the written exam might contain such a question.)

- Explain how the Mergesort algorithm works. How are sorted "runs" merged? How can it be implemented with four files?
- What is the complexity of Mergesort (and of sorting in general)?
- How can the Mergesort algorithm be improved if the input is already partially sorted?
- How can the Mergesort algorithm be improved if some memory is available (not enough to do the complete sort in memory)?
- Name four different join algorithms.
- Explain the nested loop join. What is the complexity of the nested loop join if both tables have n rows?
- How can the nested loop join be improved to make use of available memory?
- Explain the merge join. What is the complexity of the merge join if both tables have n rows, and the join attribute is a key on one side?
- Compare nested loop join and merge join. What can the nested loop join do that is not possible with the merge join?
- Explain the index join. What is its complexity? In which case is the index join clearly better than nested loop and merge join? What is an advantage of the merge join?

- Explain the hash join. What is its basic idea?
- Name some equivalences of relational algebra that can be used for algebraic query optimization.
- Why is it usually better to do a selection before a join? Explain an example where this is not better.
- Suppose an SQL query declares tuple variables over four relations `R1`, `R2`, `R3`, `R4` under `FROM`. Which structure of joins will a normal DBMS consider? What is a "bushy join"?

In-Class Exercises

b) We will continue to discuss the exam from 2015/16, which you find here:

[http://www.informatik.uni-halle.de/˜brass/dbi17/exam15.pdf]

c) Consider again the EMP-DEPT database (a simplified version of a database used in Oracle tutorials):

- `EMP(`<u>`EMPNO`</u>`, ENAME, SAL, DEPTNO`→`DEPT, MGR`°→`EMP)`
- `DEPT(`<u>`DEPTNO`</u>`, DNAME, LOC)`

Which QEP does Oracle use for the query from Homework 12A?

```
SELECT   EMPNO, SAL, DNAME
FROM     EMP E, DEPT D
WHERE    E.DEPTNO = D.DEPTNO
AND      E.MGR = 7839
ORDER BY SAL
```

Of course, the real example tables have only a few rows. Create an exactly fitting index. Does the query evaluation plan change? Try also

```
ALTER SESSION SET OPTIMIZER_MODE = first_rows
```

Does this change the query evaluation plan?

Homework 13A (Deadline: January 31)

Please read the slides 6-36 to 6-65 in the full version of the chapter about query evaluation:

[http://users.informatik.uni-halle.de/~brass/dbi17/old_c6_qeval.pdf]

In particular, have a look at the Oracle query evaluation plans for the given example queries.

d) Consider the following query to the EMP-DEPT database:

```
SELECT EMPNO, ENAME, JOB, DEPTNO
FROM   EMP E
WHERE  EXISTS(SELECT EMPNO
              FROM   EMP U
              WHERE  U.MGR = E.EMPNO)
```

There is a unique index `EMP_EMPNO_PK` on `EMP(EMPNO)`. For the optimizer mode `all_rows`, which tries to optimize the total runtime of the query, an earlier version of Oracle generated the following query evaluation plan:

```
0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 Card=6 Bytes=150)
1 0    MERGE JOIN (SEMI) (Cost=6 Card=6 Bytes=150)
2 1      TABLE ACCESS (BY INDEX ROWID) OF EMP (TABLE) (Cost=2 Card=14 Bytes=294)
3 2        INDEX (FULL SCAN) OF EMP_EMPNO_PK (INDEX (UNIQUE)) (Cost=1 Card=14)
4 1      SORT (UNIQUE) (Cost=4 Card=13 Bytes=52)
5 4        TABLE ACCESS (FULL) OF EMP (TABLE) (Cost=3 Card=13 Bytes=52)
```

(You do not have to understand the cost estimates of the optimizer. The table `EMP` has 14 rows. However, for one of the rows, the attribute `MGR` contains a null value.)

For the optimizer mode `first_rows`, which tries to return the first row as quickly as possible, that version of Oracle produced the following query evaluation plan:

```
0    SELECT STATEMENT Optimizer=FIRST_ROWS (Cost=10 Card=6 Bytes=150)
1 0    NESTED LOOPS
2 1      NESTED LOOPS (Cost=10 Card=6 Bytes=150)
3 2        SORT (UNIQUE) (Cost=3 Card=13 Bytes=52)
4 3          TABLE ACCESS (FULL) OF EMP (TABLE) (Cost=3 Card=13 Bytes=52)
5 2        INDEX (UNIQUE SCAN) OF EMP_EMPNO_PK (INDEX (UNIQUE)) (Cost=0 Card=1)
6 1      TABLE ACCESS (BY INDEX ROWID) OF EMP (TABLE) (Cost=1 Card=1 Bytes=21)
```

Please explain both query evaluation plans in your own words. How is the query evaluated? Also explain why Oracle chooses these different plans for the different optimization goals.

e) Let a relation `R(A,B,C,D)` be given. The attribute types are:

- `A`, `B`, `C`: `NUMERIC(3)`,
- `D`: `VARCHAR(1000)`.

Attribute `A` is the key of the relation. Consider the following query:

```
SELECT A, D
FROM   R
WHERE  B = 5
AND    C BETWEEN 30 AND 100
```

There are the following indexes:

- `I1` on `R(A)`,
- `I2` on `R(B)`, and
- `I3` on `R(C)`.

First consider the evaluation of the query with a full table scan. Give the Oracle query evaluation plan (in the graphical tree notation or the textual indented notation) for this execution of the query. Furthermore, give an SQL query to the data dictionary that returns an estimate of the number of block accesses that the query will need. Of course, you can assume that the `ANALYZE TABLE` has just been done, so the size information in the data dictionary is available and current. If data should be missing in the Oracle data dictionary, explain what information would be needed.

f) Now do the same for access via `I2` (with condition `B = 5`): Give the Oracle QEP and an SQL query that returns an estimate of the number of blocks that will be accessed.

Homework 13BCD (Deadline: February 16)

This exercise continues the implementation project. If you finish it, it counts as three homework exercises, but it is optional (it does not count for the number of homeworks you have to complete).

g) Please write a class `btree_c` that implements a mapping from an integer key value to an integer data value with a B^+-tree (in a file on disk using your buffer manager). There are several possible extensions, but they are not required in this exercise because they would take too much time:

- This exercise considers only a mapping of single attributes of a fixed type (`int`). One could also implement a template for mappings between two arbitrary row types, so that one could handle attribute combinations.
- A variant of this would be that the length of the tuples is determined at runtime, e.g. in arguments to the constructor of a B-tree object. The template class would have to be instantiated and compiled for each combination of row types that the

application needs. The dynamic version is slightly slower (because the tuple size cannot be compiled in), but is more flexible.

- Another variant of this is that each tuple has its own length. That is even more flexible and could handle also strings of varying length.
- The homework exercise only requires a `UNIQUE` index (i.e. a map, not a multi-map).
- For simplicity, it is also ok if the file size is determined when it is created and afterwards remains fixed, i.e. insertion fails if more blocks would be needed. Of course, you can also extend the file if you want to program that. Then you should think about extending the file not by a single block, but instead a larger piece (in the hope that the operating system selects consecutive blocks on disk).
- We also consider only a simple lookup for a given key value. A B-tree would normally support also other possibilities to access its data, namely range queries and a full scan of all entries in the leaf nodes.
- A final simplification is that you do not need to support transactions. E.g., if an insertion splits several blocks, and there is a power failure after some of these blocks were written, the data structure will be destroyed. For a real B-tree library, one would have to handle this case, but this is too much for a weekly homework.

The class `btree_c` should have the following interface:

- `btree_c(int id, const char *name)`:
  The constructor is called with a file ID and a file name. It only has to store this information for later use in `create` or `open`.
- `bool create(int size)`:
  This method creates a new file with the name and ID given in the constructor, and initializes it as an empty B-tree. This method has an argument for the number of blocks that should be the initial file size. Basically, the method `create` of the class `file_c` is called. If file creation fails, `false` is returned, otherwise `true`.
- `bool open()`:
  This method opens the file with the data given in the constructor. It calls the method `open` of the file class and may do some checking, e.g. load the first block of the file and check whether it has the right block type. The method returns `true` if the opening was successful, `false` otherwise.
- `bool insert(int key, int value)`:
  This method inserts a key-value pair (both of type `int`) into the B-tree. If the insertion fails (e.g., because the key is already present in the B-tree), the method returns `false`. If the insertion was successful, it returns `true`.
- `const int *lookup(int key)`:
  This method searches the given key value in the B-tree. If it is not found, it returns a null pointer. If it is found, it returns a pointer to the corresponding data value (stored somewhere in the database block buffer). The solution with the pointer makes it possible to have one additional value (the null pointer) besides all possible `int` values. Furthermore, if the implementation is extended to other data types or entire tuples, one would not have to copy a possibly large data value. Of course, the pointer is to read-only memory (it is `const`). The user of this class cannot change data values in the B-tree.

- `bool checkpoint()`:
  This method ensures that all modified blocks are really written to disk. You have to extend the `buf_c` class, as already mentioned in Homework 8B. You are also allowed to immediately write modified blocks, in which case `checkpoint()` does nothing. It should return the information whether all blocks were successfully written.