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

January 20, 2017

Databases IIB: DBMS-Implementation — Exercise Sheet 12 —

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 12A is January 24, 12:00 (before the problem/lab session). For Homework 12B, it is January 31.

Repetition Questions

a) What would you answer to the following questions in an oral exam? The following are questions about special data structures for relations and indexes:

- The standard storage structure for a table is a heap file. Name at least one other data structure that can be used in Oracle to store table rows.
- What is an index-organized table? Name advantages and disadvantages compared with a heap file and a normal index. What is the problem if one wants to create additional indexes for an index-organized table? Explain a solution to this problem.
- What is the purpose of a cluster in Oracle? What is the main advantage? What is the price one has to pay for that?
- What is a hash cluster? How can one find a row with a given key value in one block access?
- What is a bitmap index? In which situations should one consider bitmap indexes?
- What is an advantage of a partitioned table?

The following questions are about the execution of queries (query evaluation plans):

- A query evaluation plan (or "access plan") is relatively similar to an expression (operator tree) of relational algebra. What are the main differences?
- Name some operations that appear in Oracle QEPs.

- How can one view the query evaluation plan the the Oracle optimizer has chosen for a query?
- How can one check whether Oracle really uses an index? What are the options if Oracle does not use an index that one has created to speed up a particular query?
- Why is it good to avoid the materialization of temporary results during the evaluation of a QEP, i.e. the storage of the result relation of a subtree of the QEP?
- For which operation is it unavoidable to materialize temporary results?
- Are there situations in which the storage of intermediate results might be advantageous? Discuss advantages and disadvantages of pipelined/lazy evaluation.
- Explain the interface of nodes in the QEP operator tree if one wants to use pipelined/lazy evaluation.
- Explain the parameters `sort_area_size` and `sort_area_retained_size` of the Oracle server. Why does it make sense to choose `sort_area_retained_size` smaller than `sort_area_size` if memory is scarce?
- Where does Oracle store temporary data for sorting if memory is not sufficient?

In-Class Exercises

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

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

Homework 12A (Deadline: January 24)

Consider a database with the following tables:

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

This is a simplified version of the well-known example database from Oracle. Suppose that the employee table EMP contains 10000 rows, the department table DEPT contains 25 rows, and each department has approximately the same number of employees. A manager (MGR) supervises on average 10 subordinates. Furthermore assume that 10 rows are stored per block in both tables. The following query is given:

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

Describe how the query can be evaluated if one has the following indexes:

c) None.

d) An index on EMP(EMPNO) and an index on DEPT(DEPTNO) (both are UNIQUE).

e) Like d), but in addition a third index on EMP(MGR).

f) If you could select an index (including an index on a combination of attributes), which would you choose? The goal is that the above query is executed as fast as possible.

Homework 12B (Deadline: January 31)

This exercise continues the implementation project.

g) Please extend the buffer manager such that it actually reads blocks from a data file and writes them back. First define a class `file_c` in `file.h` and `file.cpp`. As already mentioned in the definition of `buf_c` (Homework 8B), files are identified by small non-negative numbers (type `int`). For instance, it would be possible that another module (which you do not have to write) reads a "control file" as in Oracle that contains the names and IDs of all files that constitute the database. When ROWIDs/TIDs refer to tuples in other files (e.g. from an index file to the file of the corresponding relation), then file IDs must remain stable.

   An advantage of this method is that you can define system-dependent details in `file.cpp`, while the interface `file.h` does not necessarily need to include definitions of streams or other methods to access files. If you want to structure your program in this way, `file.cpp` would contain the definition of an auxiliary class that does the "real work", and contains an attribute for the stream. file pointer, file descriptor or file handle. One would then manage a small array of objects of this class which maps the IDs to the objects with the real file data.

   The class `file_c` should offer the following static members in its public interface (basically it translates these static function calls with a file ID to normal method calls of the auxiliary (possibly system-dependend) class for the objects in the file array):

   - A constant `FILE_MAXID` for the maximal file ID (e.g. 20). The array of the auxiliary objects would be one larger than this value.
   - A static method `open(id, name)`, to open the file `name` with ID `id`. The file must be opened for reading and writing in binary mode (i.e. line ends should not be mapped from Windows to Unix). The function should return `true` if the file was successfully opened, and `false`, if not (e.g. the file does not exist or the access rights were not sufficient). The ID cannot be currently in use (i.e. no other file can be open with the same ID), or else the function should return `false`.
   - A static method `filename(id)`, that returns the file name for the file with ID `id`. The return type should be `const char *`. Of course, the file must be opened before this function may be called.
   - A static method `size(id)`, that returns the size (in blocks) of an open file with ID `id`. The file size of all files that are processed with this class must be a multiple of the block size `block_c::BLOCK_SIZE`. Again, the file must be opened before this function may be called.
   - A static method `read(id, blockno, ptr)`, for reading the block with number `blockno` from the file with ID `id`. The result (`BLOCK_SIZE` bytes) should be written to the memory at address `ptr` (this is a pointer to an object of class `block_c`). The method should return `true`, if the operation was successful, and `false`, if not (e.g. the file was too short). Again, the file must be open when this method is called.
   - A static method `write(id, blockno, ptr)` that stores the block at address `ptr` in the file with ID `id` at the block position `blockno`. Of course, the file must be open when with method is called.
   - A static method `sync(id)` to ensure that all written blocks were really written to disk (and are not waiting in a buffer of the operating system for later writing, which might not happen in case of a power failure). Of course, the file `id` must be open. If you are having problems with the buffering of the operating system you should at least call the method `flush()` to empty the buffer of the streams library.
   - A static method `close(id)`, that closes the file with ID `id`.
   - A static method `create(id, name, numblocks)`, that create a file named `name` and writes `numblocks` empty blocks to the file. Afterwards, the file should be

open as if `open(id, name)` has been called.

- A static method `extend(id, numblocks)` that grows the file with ID `id` by `numblocks` empty blocks. The file must be open when this method is called.

Please note that when you call the library function `read` in a class that has its own method `read`, you must write `::read`. The classical UNIX/POSIX-function to set the read/write position in a file is `lseek`. The C stdio library function for file pointers is `fseek`. If you use C++ streams, read about the methods `seekg` and `seekp`. The following web page might be useful:

[http://stackoverflow.com/questions/15670359/fstream-seekg-seekp-and-write]

You can find out how long a file is by positioning at the end of the file (i.e. position 0 counted from the end) and then ask for the current position counted from the beginning (`lseek` returns this position as a result, for C++ streams there are `tellg` and `tellp`).

In my program, I have used UNIX file descriptors. The method `sync()` contains the following calls:

```
        // Call OS sync function:
        int n = -1;
#if _POSIX_C_SOURCE >= 199309L || _XOPEN_SOURCE >= 500
        n = fdatasync(FD);
#elif  _BSD_SOURCE || _XOPEN_SOURCE || _POSIX_C_SOURCE >= 200112L
        n = fsync(FD);
#else
        CHECK_IMPOSSIBLE("file_c::sync: No fdatasync and no fsync");
#endif

        // Check return value:
        if(n != 0) {
                alert_c::err_sync(Filename, strerror(errno));
                return false;
        }
```

In my program, the class `alert_c` is used to store error messages. You are not required to program error messages, but it might be useful to think about error handling. You may extend the interface of a class. My macro `CHECK_IMPOSSIBLE` generates a runtime error, similar to a failed `assert`.

h) Please write a test program that creates a file with 50000 blocks. If your class for blocks does not contain the block number, extend it in this way. Then check the total runtime for the block accesses from `refstring2.txt`, where now the blocks are really read from the file. Check that the block number of the block returned by the `pin`-method is correct. On Linux/UNIX systems, the total runtime of a program `p` can be measured with `/usr/bin/time p`. Since the test data file must be created only the

first time, your program should have a command line argument “`init`” for creating the database file (as long as this is the only command line parameter value it suffices to check whether `argc` is 1). If this parameter is not present (i.e.codeargc is 0) your program should only do the block lookups and print the hit ratio. I will measure the time for each submitted program in this second run when the data file already exists.