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

December 1, 2017

Databases IIB: DBMS-Implementation — Exercise Sheet 9 —

Part a) and c) to e) will be discussed in class, you only have to submit the homeworks. This week, there are two deadlines: Part f) should be submitted until December 13, 12:00, and the programming exercise g) to i) until December 20. But please think about the questions in a) before the meeting — all of them, not only the three you selected for Homework f).

Repetition Questions

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

- What is a "segment" in Oracle? What data can be stored in a segment?
- While the name "segment" is specific to Oracle, the task is required for any DBMS. Can a DBMS use a function of the operating system for this task? Name some advantages and disadvantages.
- Why is the relationship between tables and segments in Oracle normally "1:1", but in general "n:m"? Give at least a name of the constructs that cause exceptions to the "1:1" rule.
- If you want to program a segment manager for your own DBMS, what would be its interface? The segment manager is above the buffer manager in the DBMS architecture, but it needs also direct access to the file manager (which is below the buffer manager).
- What is an "extent" in Oracle? Why does Oracle allocate storage in extents, and not in single blocks?
- How can a segment manager find free storage space in a file or tablespace if a segment is created or needs to grow? Oracle has two different solutions:
  - "`EXTENT MANAGEMENT DICTIONARY`" (the older version), and
  - "`EXTENT MANAGEMENT LOCAL`" (the newer version). This has variants with "`UNIFORM SIZE`" and "`AUTOALLOCATE`".

  How would you do this if you programmed a DBMS?
- How can one specify the initial amount of storage that is reserved for a table? Why can that be useful if you know or can estimate the size of the table?
- If you do not specify storage parameters for a segment (e.g. for a table), where are the default values defined in Oracle?

- What is a ROWID or TID (tuple identifier)? What are the main components?
- A main source of ROWIDs are indexes. Give a very high level view of an index. What would be a (greatly simplified) interface of an index?
- What are the functions of the row manager in an DBMS? What are the two basic ways in which rows can be accessed?
- Discuss advantages and disadvantages of guaranteeing stable ROWIDs for the entire lifetime of a tuple.
- How can you see the ROWID of a tuple in Oracle? How can you get a readable representation of the components.
- How can you select a row with a given ROWID in Oracle?

Some Intersting Information

b) Have a look at at least one of the following documents:

- MySQL Internals Manual
  [https://dev.mysql.com/doc/internals/en/]
- MySQL Coding Guidelines:
  [https://dev.mysql.com/doc/dev/mysql-server/latest/]
  [PAGE_CODING_GUIDELINES.html]
  This also refers to the Google C++ Style Guide:
  [https://google.github.io/styleguide/cppguide.html]
- Oracle Database Administrator's Guide: Managing Datafiles
  [https://docs.oracle.com/cd/B28359_01/server.111/b28310/dfiles001.htm]
  This contains an explanation of absolute and relative file numbers that appear in the data dictionary tables.
- Bianca Schroeder, Garth A. Gibson: Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?
  FAST'07: 5th USENIX Conference on File and Storage Technologies.
  [http://www.usenix.org/event/fast07/tech/schroeder/schroeder.pdf]

In-Class Exercises

c) Find out whether our database uses local extent management, and if yes, whether with a uniform extent size or "AUTOALLOCATE". You find the answer in the columns EXTENT_MANAGEMENT and ALLOCATION_TYPE in the table DBA_TABLESPACES. The value SYSTEM for ALLOCATION_TYPE means "AUTOALLOCATE". It is possible that the settings are different for different tablespaces, so you should also print the tablespace name.

d) Which different extent sizes are used in our database? Have a look at DBA_EXTENTS. Find a database object with large extent sizes and list all its extents ordered by EXTENT_ID.

e) How much free space is there in each tablespace or each data file? There are different approaches to get this information, check whether they return the same values. For instance, you can add the segment size for each segment in a tablespace and substract it from the tablespace size. You can also add the extent sizes in each data file and subtract it from the data file size. You can also have a look at DBA_FREE_SPACE.

Homework Exercises (Homework 9A, Deadline December 13)

f) Select any three questions from a) and write answers to them. I hope that not all students select the first three items on the list. Please write in total at least 150 words (if you write less than ten lines, I might have to count the words).

Homework Exercises (Homework 9B, Deadline December 20)

g) Create a file `btype.h` with the declaration of an enumeration type `btype_t` for types of blocks. Probably, you will need the following constants:

- `BTYPE_INVALID` for blocks that are not really initialized,
- `BTYPE_EMPTY` for blocks that are not yet used,
- `BTYPE_HEADER` for the first block in the file with meta-data and control information,
- `BTYPE_BRANCH` for branch blocks in the B^+-tree,
- `BTYPE_LEAF` for leaf nodes in the B^+-tree.

The exact set of values of this type can be extended later, at the moment it is only important that the type is declared (so that you can use it in the class declaration for blocks). For instance, if one wants to implement standard heap files to store tuples of relations, one would add one or more block types.

If you want, you can add a function `btype_name` that returns a string representation of the enumeration constant (i.e. it has an argument of type `btype_t` and returns the constant name). That might be useful for debugging output, but is not required.

You could also define a function `btype_valid` with an argument of type `btype_t` that returns `true` if the input value is really one of the defined enumeration constants. Since we will read blocks from a file, it might be good to check the integrity of the data structures. But again, it is your decision whether you want such a function.

h) Define a class `block_c` for database blocks (in the file `block.h`). You will later define classes for specific kinds of blocks which might be subclasses of this class (depending on the approach you take, see below).

Define a constant `BLOCK_SIZE` in `block.h` for the size of database blocks. Choose the value `8192` (8 KByte). The objects of the class `block_c` should be exactly `BLOCK_SIZE` bytes large. You can check that by printing `sizeof(block_c)` in the main program.

When reading and writing data, always units of 8 KByte are moved between main memory and disk (this file IO will be next weeks homework). Since `block_c` objects are written to disk and later loaded again at a different memory location, normal pointers in them would not be helpful (at least, they could be used only while the block is pinned in memory).

Now there are several possibilities how you can work with the different types of blocks.

- One option that I used earlier is that the `block_c` objects contain mainly an array of characters for the data. More precisely, I used a union of the `char` array, a `short` array, and an `int` array so that I can access the space in the block with different types. Of course, one has to be careful that one does not overwrite a particular position in the block with a value of a different data type.

In my program, I had subclasses for special types of blocks. These subclasses added methods, but no new attributes (because the size of the object is already `BLOCK_SIZE`). That means of course that one has to construct the data structures in the blocks on a low implementation level (basically, with byte offsets in the block — of course, I defined constants and macros for computing the offsets).

- Another approach is to see the `block_c` objects only as a physical container for objects that are specific for each type of block. Then you have several "block contents" or "data" classes and put inside the `block_c` class again a union, but this time a union of classes on a higher abstraction level.

  Whereas in the previous approach, all the special cases of blocks depend on `block_c`, in this approach, `block_c` depends on all contents classes. The contents classes would not be subclasses of `block_c`.

  If you follow this approach, you need to define an "empty contents" as the first member of the union. This must ensure that the `block_c` objects have the right size. Use a character array to make it fit.

  There is also the problem that the contents classes need the constant `BLOCK_SIZE` and also need to know the overhead of the block header that is the first part of `block_c` (before the union). However, `block.h` vice versa needs to include the definitions of the block contents classes. One solution would be to define the block size in a general include file `ver.h` for parameters that can be configured. You could also introduce a class `bhead_c` for the block header, and (if needed) `bfoot_c` for the block footer. Then the block contents classes could compute the remaining storage size in a block without including `block.h`.

Interface of the class `block_c`:

- The class `block_c` must have at least the method `type()` that returns the type of the block (i.e. a value of type `btype_t`).
- The constructor of the class should initialize the block type to `BTYPE_INVALID` to show that the main contents part contains garbage. In many cases, the object will be immediately overwritten with data read from disk, therefore it seems unnecessary to set the entire contents to 0.
- There should be a method `init_empty` that does an initialization as an empty block. This must set the block type to `BTYPE_EMPTY` and ensure a defined contents (all 0 bytes). Even if bytes in the block are not yet used, it would be strange if the blocks in the file contain garbage in unused positions. Later other initialization methods must be added for other block types.
- It is recommended (but not required) that there is a basic check for the integrity of a block. For instance, if the user opens a file that was not constructed by this program, there should be an error message. For instance, you could define a method `check_err()` that returns an error message as string or a null pointer if everything is ok. You could store a certain bit pattern ("magic number") at the beginning of every data block which is used to recognize blocks written by your program.

- It might also be useful (but is again optional) to store the block number inside the block object. Then one would define a method `block_no()` to query the block number and a method `set_block_no()` to set the block number. The block number is a 32-bit value (databases can be large). One could declare it as `unsigned`, but then might get compiler warnings for type incompatibilities (one has to write explicit casts). In my earlier program I defined a type `bno_t` (in `bno.h`) for block numbers, but maybe that was too much.
- A well-known method to recognize partially written blocks is to store a certain bit pattern at the start and end of every block and invert it each time before the block is written. When reading a block, one checks whether the two bit patterns are identical. If you want to do that, you can define a method `prepare_write()`, that inverts the bit patterns and is called once before the block is written.

i) You have to change the objects of your buffer manager so that they contain a pointer to objects of class `block_c` or contain such an object as an attribute. That also means that memory must be actually allocated to buffer blocks (up to the defined limit).

In the next homework, we will actually read and write blocks from/to a file on disk. Our goal is to implement a simple B^+-tree.