INFO-H-417: Database Systems Architecture – Lab 1

Lecturer: Stijn Vansummeren
Teaching-Assistant: Stefan Eppe
`http://cs.ulb.ac.be/public/teaching/infoh417`

Main Relation Construction

In this second part, we will build a relation in our database engine and print it out on screen.

1. The database currently contains an empty relation named `Foo`. Describe the Schema of this relation. In particular: what are the attributes of this relation, what size occupies each attribute, and what is the length of a record?

   The relation is described in the header of the `IOTest` file. The `Foo` relation has two fields `A` and `B`. Each of these fields stores a 4 bytes integer. Hence, each record in this relation is 8 bytes long.

2. Populate the table with a single record, filling in the `createRecord(id)` procedure to create a record whose A-field is 42 and B-field is 1. Add this record to the `Foo` relation, in the function that populates the database.

   The code should read as follows:

```
private static Record createRecord(int i) {
    Record r = new Record(FOO_SCHEMA);

    r.putInt(FOO_A, 41);
    r.putInt(FOO_B, 1);

    return r;
}

private static void createDatabase(File file) throws Exception {
    [...]
    log("Loading " + NUMRECORDS + " records... ");
    foo.addRecord(createRecord(0));
    log("done");
    [...]
}
```

3. Update `createRecord(id)` to match its documentation. Fill the relation with `NUMRECORDS` records.

   The code should read as follows:

```
private static Record createRecord(int i) {
    Record r = new Record(FOO_SCHEMA);

    r.putInt(FOO_A, i % 10);
```

```
        r.putInt(FOO_B, i / 10);

        return r;
    }

    private static void createDatabase(File file) throws Exception {
        [...]
        log("Loading " + NUMRECORDS + " records... ");
        for (int i = 0; i != NUMRECORDS; ++i) {
            foo.addRecord(createRecord(i));
        }
        log("done");
        [...]
    }
```

The goal here is to create a table that is not sorted according to either A or B. As such, the table has no "natural" structure that we can exploit to fetch records efficiently when looking for a given value in the A field. In Lab 2, this will help us study how B-Tree indices can help speed up query processing.

4. Based on the schema of the relation, give a rough estimate of the number of blocks used to store 50, 1000, and 10000 records. Note that each block has a size determined by the `BLOCKSIZE` constant (expressed in bytes).

   Given a record size of 8 bytes we need $400 = 8 \cdot 50$ bytes to store 50 records. With the given `BLOCKSIZE` of 512 bytes, 1 block is therefore sufficient to store the whole relation. Similarly, we need 16 blocks for 1000 records and 157 blocks for 10000 records.

   These are rough approximations. In pratice, we need some additional information. For instance, in our prototype database, each block contains, in addition to the records themselves, some header information that contains a pointer to the previous block, another pointer to the next block, and the total number of records stored in the block. The header takes 12 bytes in total per block. Furthermore, one byte of bookkeeping is stored with each record to indicate whether the given record has been deleted or not.

   The actual computation is therefore closer to (8+1) bytes per record, while blocks only have 500 bytes available to store the actual records. That is, we need 1 block for 50 records, 19 blocks for 1000 records, and 181 blocks for 10000 records.

   In the next exercise, we check these figures against the actual implementation by counting the number of read requests sent to the disk when scanning the `Foo` relation.

I/O operations

In this third part, we edit the disk manager to provide statistics on the number of blocks read from the disk.

1. Open the `DiskManager` class. Describe the various methods that it supports. In particular, pay attention to the methods that may load blocks from disk.

   The disk manager supports the following public operations:

   - `getNumBlocks`/`getBlockSize`: get statistics on the blocks managed by this database.
   - `getDBMetaBlockAddress`: get the block containing all the database main data (i.e. where the relations catalog is stored, and where the list of free blocks is to be found).
   - `writeBlock`: write the specified block at the specified address on disk.
   - `readBlock`: read a block from the specified address on disk.

- `allocateBlock`: gets the address of a block from the free list, and remove the block from the free list.
- `freeBlock`: add the specified block address back into the free list.

Note that `readBlock` is the only method that loads blocks from disk.

2. Edit the `DiskManager` class to add the capability to count the number of blocks read from the disk. In particular, provide a public method to get this statistics.

   The idea is to add an integer member field to the DiskManager class that gets incremented at each call to `readBlock`.

3. Print the statistics of the number of blocks read before and after printing the `Foo` relation on screen, when `NUMRECORDS` is 50, 1000 and 10000. Compare to your first estimate.[1]

   The statistics show that we need to:

   - read 1 block to scan a relation comprising 50 records;
   - read 19 blocks to scan a relation comprising 1000 records; and
   - read 184 blocks to scan a relation comprising 10000 records.

   This agrees well with our approximation when taking into account the various bookkeeping information.[2].

   However for all practical purposes, even the first crude approximation shows that scanning the whole relation is costly and should be avoided.

4. Can we reduce the number of reads if we only need to print the records for which $A = 4$? Explain.

   No. Our method of querying the `Foo` relation is to scan it in its entirety. Since the records are not ordered on A, we cannot stop scanning at any point, and we need to read each and every block of the relation.

[1] Many of the differences can be explained by headers that occur in each `MainBlock` and bookkeeping information attached to each record. The constructor of `MainBlock` hints at what kind of additional information is stored.

[2] There is a small discrepancy for the 10000-record relation which is due to the actual storage mechanism in the database prototype that keeps track of which blocks in the relation are full (and can hence not be used to insert records into)