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

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

Academic Year 2014–2015

Architecture of our Database System

In this first part, we will explore the architecture of a database framework.

1. Download the database implementation from the course wiki (`http://cs.ulb.ac.be/public/_media/teaching/infoh417/btree-java.tgz`).
2. Compile this project from eclipse and run it. It should indicate that it creates a database.
3. Read the following description of the framework, and make sure you can access the following classes from eclipse: `DiskManager`, `BufferManager`, `MainBlock`, `Relation`, and `IOTest`

The `be.ulb.db.diskmanager.DiskManager` class is responsible for the actual reading and writing of blocks from secondary storage. The blocks read and written from secondary storage are specified by means of their address, represented as `be.ulb.db.BlockAddress` objects. The `DiskManager` is also responsible for reserving space on the secondary storage when creating the database, and maintaining a list of free blocks for future allocations.

It is the responsibility of the buffer manager (`be.ulb.db.buffermanager.BufferManager`) to keep track of which blocks are loaded in main memory, and to make them available to the other components of the DBMS. In particular, the `Buffermanager` will try to minimize the number of I/O operations, to reuse buffers that are no longer used, etc. More information concerning the tasks of a buffer manager can be found in Section 15.7 of TCB.

Various kinds of blocks exists, each corresponding to a different strategy for representing data in memory and on disk. All blocks are derived from the base class `be.ulb.db.Block`. For instance, the `be.ulb.file.MainBlock` class provides the functionality to store records with the same schema in an unordered fashion in the block, to read the records back from the block, and to remove them.

The records inside a `MainBlock` can be identified through a `be.ulb.db.RecordAddress`: these addresses are composed of a `BlockAddress` that identifies the `MainBlock` in which the record is stored, and of a slot identifier that indicates where the record is stored inside the block. For storing whole relations, `MainBlock`s are linked together, and managed by `be.ulb.db.file.MainFile` objects. The `be.ulb.db.Relation` class provides facilities to organize records inside a relation, by relying on `MainFile` for the actual storage. The `Relation` class also provides management and maintenance of index structures over the relation.

Finally, the `be.ulb.db.Database` class provides the functionality to create a new database or load an already existing database. In particular, this class is responsible for creating instances of the `BufferManager`, the `DiskManager` to manage blocks. It also creates a `Catalog` that provides access to all relations in the database.

The `TestIO` class serves as a test bench for our `Database`. As provided, it only instantiates an empty relation in a new database, and scans this relation. In the next parts of this lab, we will first modify the code to load test data in our database, and then count the number of I/O operations needed for reading these data from the disk.

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?

2. Populate the table with a single record. For this purpose, fill in the `createRecord(id)` procedure to create a record whose A-field is 42 and B-field is 1. For this purpose, you will need to create a `new Record` object, passing `FOO_SCHEMA` as the schema. You will need to `put` the different values in the record.

   Add this record to the `Foo` relation, in the function that populates the database. Run the program and check that the record appears on screen.

3. Update `createRecord(id)` to match its documentation. Instead of adding just one record, fill the relation with `NUMRECORDS` records.

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).

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.

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.

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]

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

Buffers Management

Supplemental exercises:

1. Open the `BufferManager` class. Check the `pinBlock` method. How does it differ from reading blocks in the `DiskManager`?

2. Edit the `BufferManager` class to add the capability to count the number of blocks pinning requests. In particular, provide a public method to get this statistics.

3. Compare the number of pin requests to the number of blocks read before and after printing the `Foo` relation on screen, for different values of `NUMRECORDS`. Do they differ significantly? Explain.

4. Print the `Foo` relation a second time. Is there a change in the number of blocks read and of pin request? Explain.

[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.