Physical Data Organisation and Index Structures

Project Assignment

2011-2012

Description

Implement the BTree indexing mechanism as described in Section 14.2 of "Database Systems: The Complete Book". On the course's website you'll find the basis of a simple DBMS written in `Java`. It contains most of the components of a typical DBMS implementation, and is organized as follows:

- The byte sequences that we read from and write to disk are represented in `Java` by means of the `java.nio.ByteBuffer` class. Since you will also have to use this class, it is useful to first read its documentation.
- The classes derived from `be.ulb.db.Schema.AttributeType` represent the types of values that we can store in an attribute. Derived classes exist for integers, strings, booleans, block addresses, and record addresses. This class also offers the functionality to read and write an instance of an `AttributeType` from/to a `ByteBuffer`.
- Schema information is represented by means of the

  `be.ulb.db.schema.Schema`

  class. It provides the functionality to (1) create a new schema given the names of the attributes and their corresponding types; (2) calculate the number of bytes needed to represent a record with the given schema; (3) retrieve the type of an attribute in the schema; (4) and so on.
- Records are represented by means of the `be.ulb.db.Record` class. Only fixed length records are supported. A `Record` has an internal `ByteBuffer` that is used to store the actual sequence of bytes contained in the record. Every record has an associated `Schema`. The `Record` class provides functionality to read and write the separate fields of the record. This class also enforces that values read or written are compatible with the types declared in the `Schema`.
- The `be.ulb.db.diskmanager.DiskManager` class is responsible for the actual reading and writing of `ByteBuffers` to secondary storage.

This class is also responsible for the creation of the initial database file, and for the allocation of blocks on disk. For the purpose of this project, you will not have to use the diskmanager directly. The address of a block on disk is represented by means of the `be.ulb.db.BlockAddress` class.

- As already mentioned, when disk blocks are loaded into memory, they are represented by means of `ByteBuffer` objects. It is the responsibility of the buffer manager (`be.ulb.db.buffermanager.BufferManager`) to manage these in-memory buffers, and to make them available to the other components of the DBMS. In particular, the `Buffermanager` will try too 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.

- Disk blocks are hence read from secondary storage into a `ByteBuffer` object. These are, however, only "raw" sequences of bytes that need to interpreted. Nothing prevents us from storing information in particular kinds of blocks (e.g., index blocks) different from the way in which actual records are stored in other blocks. For each separate representation strategy, a new class needs to be derived from `be.ulb.db.Block`. This derived class needs to provide a suitable interface to add records (and the like) to the block, remove records, search records, etc. A concrete example is given by the `be.ulb.file.MainBlock` class, which 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.

- Loading a block on disk into a `ByteBuffer`, and linking this `ByteBuffer` to a `Java` object derived from `Block` is done by means of the `pinBlock` method of the `BufferManager`. Releasing the `Block` object (and its associated `ByteBuffer`) is done by means of the `unpinBlock` method. Allocating room for a new block on disk can be done by means of `newBlock`, while releasing this space can be done by `freeBlock` (both in `BufferManager`).

- Collections of block form `files`. The framework has two kinds of files: `be.ulb.db.file.DirectoryFile` and `be.ulb.db.file.MainFile`. A `DirectoryFile` is meant to store "compact" files from which data is rarely deleted. The `MainFile` is better suited to store the data of a relation. Both files are collections of `MainBlocks`, and are hence unordered.

- When we add a record to a file, the address of that record in the file (represented by a `be.ulb.db.RecordAddress` object) is returned. A `RecordAddress` consists of the `BlockAddress` in which the record is stored, together with the index of the record inside the block (we do not use pointer swizzling).

- The abstract class `be.ulb.db.index.Index` describes the functionality that an index should provide. Each `Index` derivative should be able to add *(key value, record address)* pairs, delete such pairs, and retrieve all addresses with a particular key value. Key values are represented by means of `Record` objects that satisfy the `Schema` of the `Index`.

- Relations are represented by means of the `be.ulb.db.Relation` class. `Relation` objects have a `MainFile` in which they store their data, and know the indexes that are available on the file. `Relation` objects offer the functionality to create new indexes, remove indexes, or use indexes. They also allow adding records to the relation, deleting records, and searching for records. The related indexes are updated accordingly. Indexes are used during searching when possible. All information concerning a relation (its name, the address of its first block, its schema, the indexes, and so on) are saved in the system catalog, which is represented by means of the `be.ulb.db.catalog.Catalog` class.

- The `be.ulb.db.Database` class provides the functionality to create a new database or load an already existing database. This class is responsible for creating instances of the `BufferManager`, the `DiskManager` and the `Catalog`.

In particular

- Implement the BTree index mechanism (section 14.2) in the class

  `BTIndex.java`

  of the framework. Use the framework described above. It suffices to only allow unique key values when adding to the index (duplicate search keys hence do not need to be supported). Be sure to implement *all* functionality: insertion, search, and deletion (including the necessary reorganisations when inserting and deleting!).

- The best way to proceed is to derive a new class from `Block` in which you can add, search, and delete *(key value, address)* pairs in a sorted manner. An example of how you can store records in a block is given in `be.ulb.db.file.MainBlock`. It is also advisable to implement a `toString()` method that can display the contents of a block in a human-readable way. This `toString()` method will be most useful during the testing and debugging of your project!

- The derived `Block` class can then be used to implement your `BTIndex` class. Again, it is advisable to implement a `toString()` method that can display the contents of your BTree in a human-readable manner.
- You can use `Tester.java` to test your implementation.

Careful! You are *not* allowed to modify the framework itself!

Assignment

1. This project work contributes 6/20 to the overall grade, and the written exam contributes the remaining 14/20 points. You will be judged on the correctness and completeness of your solution, as well as on the cleanness of your code. That implies that you should:
   - Abstract by means of classes whenever appropriate.
   - Use and re-use function whenever appropriate.
   - Use exceptions correctly.
   - Add sufficient and clear comments to your code.
2. This project work should be done *individually*. This implies that you *individually* write your code and individually solve the problems that occur. Fraud occurs from the moment that source code is shown to another student, or source code is exchanged (be it willingly or not). We have advanced software to identify project solutions that may be subject to fraud. Violators will be punished according to article 34 of the exam regulations shown in Figure 1.
3. This assignment is obligatory. If you do not make the assignment, you cannot pass the course in the first exam session.
4. You will have to create a mercurial repository[1] in the `INFO-H-417` repository group at `http://informa2.ulb.ac.be` to submit your code. The username and password to login to this system correspond to your ULB/VUB NetID. The repository will be named `project-<student>`, where `student` corresponds to your username. It is recommended that you create this repository *as soon as possible* to avoid last minute technical difficulties
5. Your solution should be pushed to the repository *no later than 27 may 2012*. You get a penalty of -1/6 points for each day that your solution is delayed.

[1]`http://mercurial.selenic.com/wiki/Tutorial`

Art.34 In case of fraud or plagiarism during an examination or during a test at an interim date during the academic year, or in relation with the preparation of written reports or papers, the course professor reports the case in writing prior to the jury deliberation to the relevant academic authority levels for disciplinary matters. A copy of that fraud report is addressed to the jury chairmen. The student can ask to be heard by a jury chairperson prior to the jury deliberation, in presence of the related course professor. Without prejudice to the disciplinary processes at the University Faculty level, in case of fraud the student points for the related course are brought down to 0/20. The jury further can:

- *decide to cancel the examination session;*
- *decide to refuse the student access to both examination sessions of that academic year.*

Figure 1: Excerpt of the Exam Regulations concerning fraud.