

DBSys Lab Assignments

FALL 2020

Preliminaries

The laboratory assignments for this course involve the development and implementation of algorithms that interface with the existing Minibase code. For this reason, it is advised to get used to the code of Minibase (at the very least, its main classes). If you do not have a background in Java, it is also recommended to check out tutorials on the language (e.g. <https://docs.oracle.com/javase/tutorial/>).

1 Minibase

Minibase is a database management system intended for educational use which contains different parts such as a parser, optimizer, buffer pool manager, storage mechanisms (heap files, secondary indexes based on B+ Trees), and a disk space management system. The goal of Minibase is not just to have a functional DBMS, but to have a DBMS where the individual components can be studied and implemented by students. It has been developed in conjunction with the book Database Management Systems by Raghu Ramakrishnan.

We give next an overview of the Minibase main components. Note that as Minibase is a fully functional DBMS, it implements all functions of a DBMS using Java classes. For what concerns this course, however, only a part of them will be considered.

The most important classes to study are reported in Figure 1 and will be summarized in the following section. In Figure 1, classes that interact with each other (or extend each other) are connected with an arrow. While it is useful to look at all classes (Exceptions may generally be skipped until one is encountered while executing the code), those shown in the figure require more attention.

If you are working on the code using an IDE such as Eclipse, it is normally possible to find the Declaration of a method or class by right-clicking on the method name and selecting **Open Declaration**. This allows to easily navigate around the many classes included in Minibase.

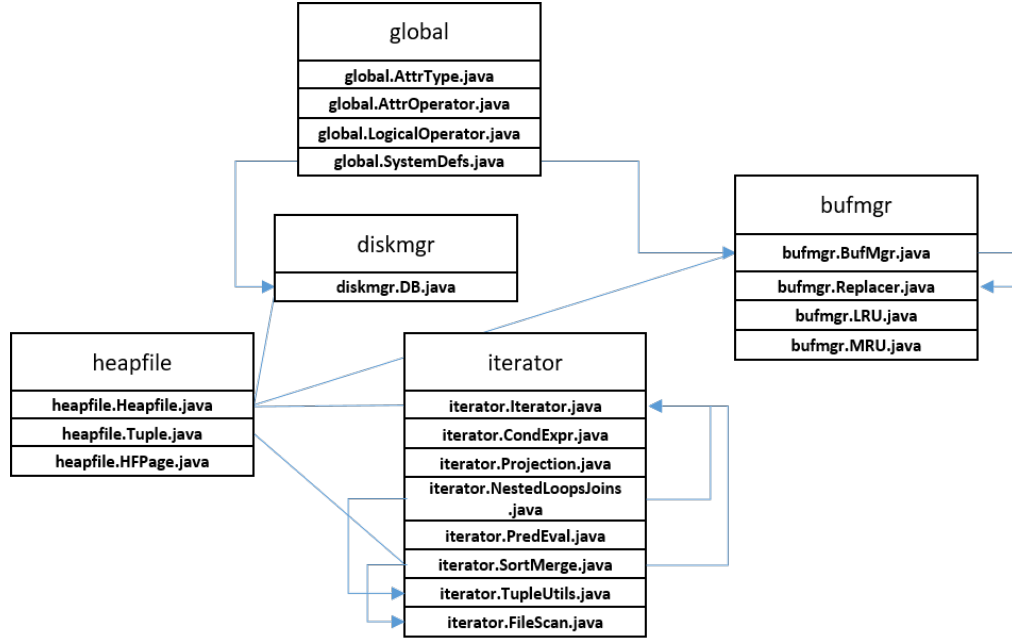


Figure 1: Main classes in the codebase.

1.1 Disk Space Manager

The component of Minibase that takes care of the allocation and deallocation of pages within a database. It also performs reads and writes of pages to and from disk, and provides a logical file layer within the context of a database management system. The database is created by building a Random Access file that contains `num_pages` db pages. All pages are allocated when the database is created and filled with “0”. Minibase works at the byte level, so all pages have the same size in bytes and the database navigates the pages by using `page_size`. Similarly, whenever a page must be fetched by the database, the fetch operation relies on the `page id` to find the byte offset from the start of the database. In Minibase, all functions that work on pages start from the assumption that pages have a well defined size in bytes and format. This will be important in the later assignments. Disk manager classes can be found in the directory `diskmgr/` and the main class is `diskmgr/DB.java`.

1.2 Buffer Manager

The Buffer Manager reads pages from the Heap into a main memory page as needed. The collection of main memory pages (called frames) used by the buffer manager for this purpose is called the buffer pool, which is just an array of Page objects. The buffer manager is used by the code for access methods, heap files,

and relational operators to read, write, allocate or deallocate pages. The Buffer Manager calls the underlying DB class object, which actually performs these functions on disk pages.

Whenever the DBMS requires a page to be added in memory, the buffer manager will try to add it to the buffer and pin it as the frame is now in use. Pinned frames are frames that are currently in use by the DBMS and cannot be removed by the buffer manager until they are unpinned. If a frame is modified, then its changes must be propagated to the heapfile before freeing its space from the buffer.

As the buffer manager is working in the main memory, it has a limited size and (usually) cannot load the entire DBMS in its data structure. The main prerogative of the buffer manager is handling the space in the buffer in the most efficient way possible. When the buffer fills up, the buffer manager needs to decide which frames need to be removed from the buffer to make room for new frames. This is done using according to a replacement strategy. The code provided here contains the implementation of three strategies, Clock (default, in `BufMgr.java`), LRU and MRU.java. These classes extend the `Replacer.java` class.

The code of the buffer manager can be found in the directory `bufmgr/`. The main class is `bufmgr/BufMgr.java`. Classes `bufmgr/LRU.java` and `bufmgr/MRU.java` extend `bufmgr/Replacer.java`, and will be the focus of one of the assignments.

1.3 Heap Files

Heapfiles are used to store unordered sets of records. The code used in the heap is in directory `heap/`. Records are stored as byte sequences, with the `heap/Tuple.java` class handling the conversion to other datatypes. `Heapfile.java` is used to create and handle the heapfile. It is also used to add and remove records, as the addition (or deletion) of a record might require the creation (or deletion) of a page in the heapfile. Heapfiles contain Pages (defined in `HFPages.java`).

The `HFPages.java` class contains all the methods required to handle records, including creation, deletion, search, iteration. It also provides methods to provide information about the pages, such as the number of records and the remaining available space. Records accessed in a `HFPages` are loaded in the buffer manager and pinned. This allows to propagate any change back to the Heapfile.

Class `Tuple.java` defines the structure of a table tuple as a sequence of bytes split according to a known, fixed structure. The structure of a tuple can be set using the method `public void setHdr (short numFlds, AttrType types[], short strSizes[])`, which will then properly space the tuple values according to their offsets. Class `JoinTest.java` shows a simple example of how tuples are prepared according to Minibase's architecture.

1.4 B+ Trees

B+ Trees are the only access methods that are available in Minibase. An access method (or index) facilitates retrieval of a desired record from a relation. In Minibase, all records in a relation are stored in a Heap File. An index consists of a collection of records of the form `<key value, rid>`, where key value is a value for the search key of the index, and rid is the id of a record in the relation being indexed. Any number of indexes can be defined on a relation.

1.5 Globals

The directory `global/` includes some utility functions and some classes that contain constant values that must be accessible to the entire Minibase DMBS. The class `SystemDefs.java` contains the instantiation of the DB and BufMgr classes according to the arguments supplied in a suitable `main` function (e.g. in `JoinTest.java`). More precisely, the newly created DBMS will have the given `dbname`, with `num_pgs` pages, a buffer whose size is `bufpoolsize` and with the given replacement policy. If no replacement policy is provided, then the buffer manager defaults to “Clock”.

Some of the global variables that are included in the directory are:

- `AttrOperator.java`: this class defines the different comparison operators (e.g. `>`, `<`, `=` etc.) that will be used in the conditional expressions used to build queries.
- `AttrType.java`: this class defines the constants that will be used to identify the type of values in a tuple. In turn, the type of each value will be used to choose the correct comparison operator when necessary.
- `LogicalOperator.java` defines the logical operators (AND, OR, NOT) that will be used to combine logical expressions and comparisons.

In all cases, operators and types are defined as integers. `TupleOrder.java` is an enumerator for tuple orders (ascending, descending, and random).

You should refer to these classes when implementing a query parser.

1.6 Query optimization

In Minibase, a query is first parsed and optimized to obtain an evaluation plan. The evaluation plan is structured as a tree of iterators and corresponds typically to a relational algebra operator. Execution is initiated at the root of the plan tree, which successively generates calls on descendant nodes. Execution consists essentially of a data-driven loop of such calls until all the input tuples have been processed. Figure 2 shows this process.

1.6.1 Parser and Optimizer

Given a SQL query, the best plan for evaluating it is found. The optimizer is similar to the one used in System R (and subsequently in most commercial



Figure 2: Query

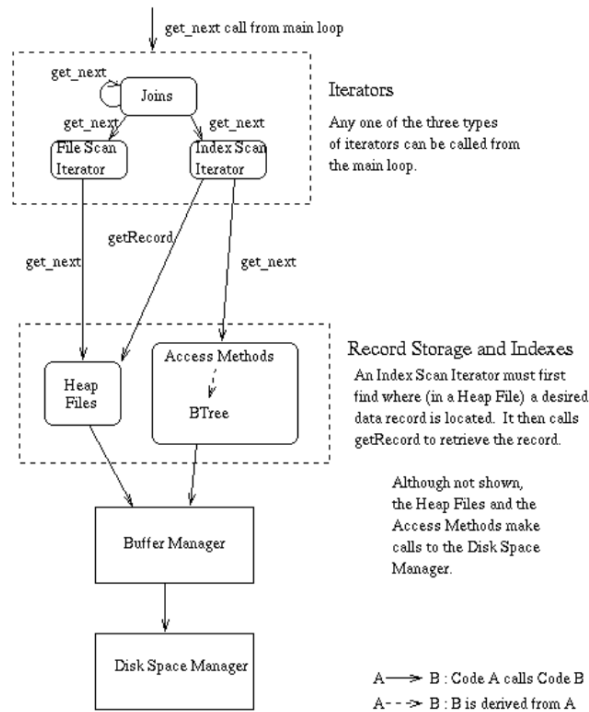


Figure 3: Main Loop Process

relational database systems). In optimizing a query, the optimizer considers information in the catalog about relations and indexes. It can read catalog information from a Unix file, and thus the parser/optimizer can be used in a stand-alone mode, independent of the rest of Minibase.

1.6.2 Execution Planner

It takes the plan tree produced by the optimizer, and creates a run-time data structure. This structure is essentially a tree of iterators. Tuples are returned in response to tree node calls by copying them into dynamically allocated main memory (not the buffer pool). After that, in the **Main loop** the execution is triggered by pulling on the root of the tree. This results in similar calls to iterators lower in the tree. Leaf level iterators retrieve tuples from a relation (perhaps using an index), and intermediate level iterators in the tree correspond to joins. Selections and projections are piggy-backed onto these iterators. See Figure 3 for a diagram of the main loop process.

1.7 More Resources

- More information and details on Minibase, as well as some examples on how to implement common operations can be found at https://research.cs.wisc.edu/coral/mini_doc/project.html
- In your minibase installation folder, the generated documentation can be browsed from the java doc index found in the path `javaminibase/src/javadoc/index.html`

2 Phase 1 - individual

Deadline: 19th Oct 2020, 23:59 (GMT+2)

2.1 Prerequisites

Make sure you have the JDK installed on your machine. Linux is strongly recommended (a VM is fine) to complete these assignments. It is possible to code in a Windows environment, however be aware that this may lead to complications that could be avoided by using Linux, since all the scripts that will be used are expecting a Linux system.

2.1.1 Preparing Minibase

1. Download the Minibase archive file and unzip it.
2. Modify the “Makefile” scripts (in every folder) to reflect your directory structure using a text editor (e.g. “nano” or “gedit” on Linux):
 - Change JDKPATH to the SDK path on your machine. This can be done by using the command `readlink -f $(which java)` in a terminal window. The output of the command should be similar to `/usr/lib/jvm/java-11-openjdk-amd64/bin/java`: the SDK path will in this case is then `/usr/lib/jvm/java-11-openjdk-amd64`.
 - Change LIBPATH to `LIBPATH=./..`
 - If needed, update file `minjava/javaminibase/src/Makefile` from the old version to the new

```
– OLD = cd tests; make bmttest dbtest; whoami;
    make hfttest bttest indextest jointest
– NEW = cd tests; whoami; make jointest
```

2.1.2 Running Minibase in Eclipse

It’s strongly recommended to use an IDE like **Eclipse** (or any Java IDE you’re familiar with) to work on the assignments.

1. After installing Eclipse, launch the IDE.
2. In Eclipse, click on **File** → **New** → **Project**.
3. Select **General** → **Project** from the new window.
4. Choose a name for the project, then **Finish**.
5. Right click on the project name in Package Explorer, then on **Import**.
6. Select **General** → **File System**.

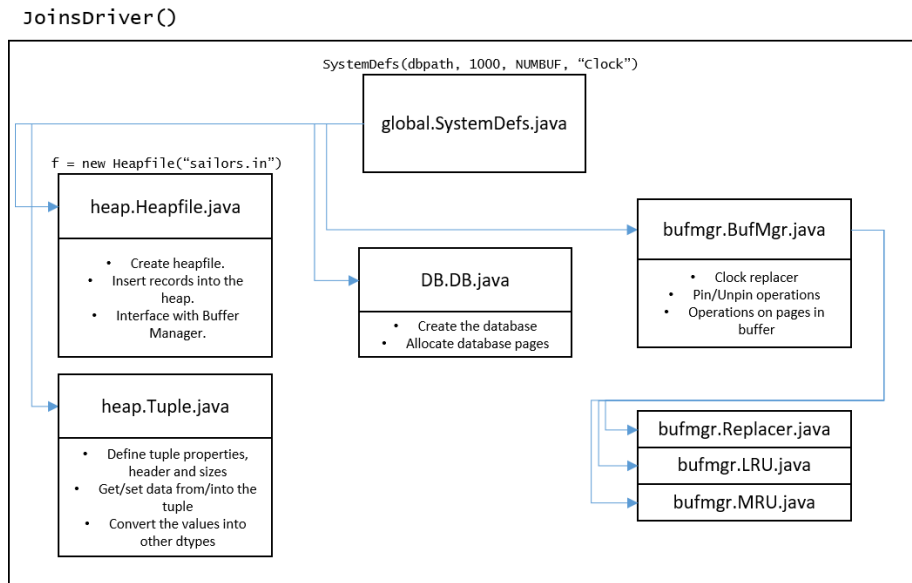


Figure 4: Overview of the major classes involved in `JoinsDriver()`.

- Click on **From directory** → **Browse** and navigate to the minibase folder you extracted (`path/to/folder/minjava/javaminibase`), then click on **open**.
- Click on **Select All**, then **Finish**.
- Click again on the project name, then on **Properties**.
- In **Properties** → **Project Natures**, make sure that "Java" is present. If it is not, click on **Add**, then on **Java**.
- In **Properties** → **Java Build Path** → **Order and Export**, check the box next to **JRE System Library**.
- To make sure that the code works, use the **Package Explorer** to navigate to the "tests" package, right click on `JoinTest.java` and then on **Run as Java Application**.

2.2 Tutorial on debugging

This section is completely optional and is targeted to students that are not experienced with coding in Java and with debugging. If you are well-versed in Java programming and debugging, then you can safely skip this and work on the assignment.

As you will learn, Minibase is a complex system that employs a large variety of classes to work as a proper DBMS. Since it is programmed in Java,

query1()

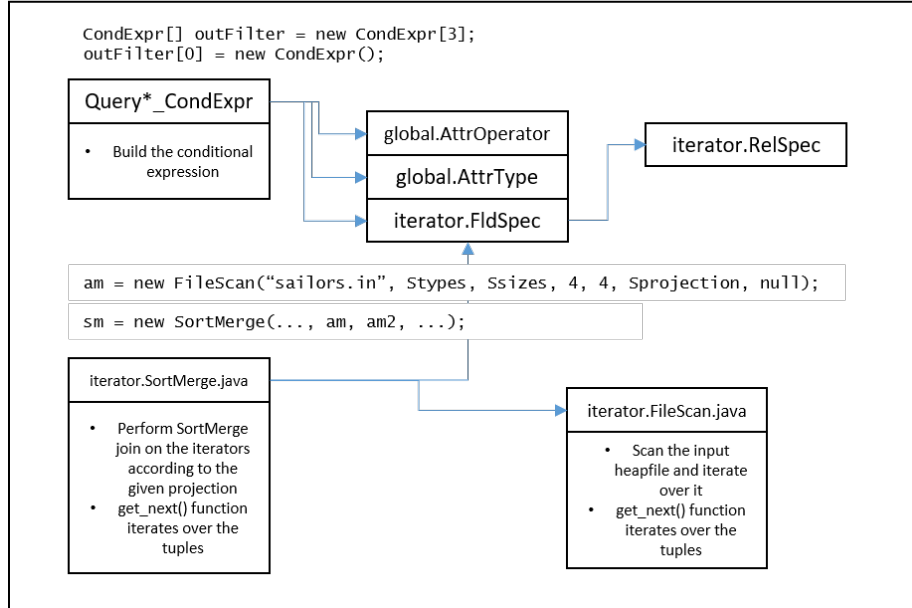


Figure 5: Overview of the major classes involved in `Query1()`.

it relies on the language's object-oriented features to implement the different parts of the code. It may be difficult to parse the flow of execution of Minibase simply from looking at the code, so this section will go through one of the operations carried out by the DBMS when the `JoinTest.java` class is run as an application. Although we'll start from class `JoinTest.java`, some of the other classes we will touch include `SortMerge.java`, `Tuple.java`, `PredEval.java` and `Projection.java`. While it is not necessary to understand what every all methods in each class do, having a high level idea of what's the function of each class will help with navigating the code. Figures 4 and 5 give a high level overview of the major classes involved in the execution of the application. In both figures, classes that have important interactions with each other are connected using arrows. The figures should only be used as a starting point as the code is much more complex than represented there. It is therefore recommended to read the code of the mentioned classes to have a better understanding of what they do.

Specifically, we will look into how the Projection operation is executed, which classes are touched by it and what's its result.

The follow explanation assumes that you are using the debugger provided with Eclipse, however any other IDE may be used (generally, most if not all debuggers include the functions considered here).

Debugging 101

By using a debugger, it will be possible to follow the thread of execution of the code and observe the content of variables of interest.

Text in `monospace` denotes either a function or a debugging command. You can hover the mouse pointer over the debugging icons to see the name of the command they'll execute (e.g. `Step into`, `Step over`).

- To run the application in debugging mode, select `Run` → `Debug`.
- Set breakpoints by double-clicking on the bar to the left of the line numbers.
- The content of variables can be shown by accessing the `Variables` panel in `Window`.
- Once a breakpoint is reached, it is possible to `Step into` or `Step over` the line.
- `Resume` the execution to let the code run until the next breakpoint, or until the execution is complete.
- It is possible to switch to the declaration of a function by pressing `CTRL + Left click` on a Windows/Linux system.

The debugger has many more functions that can be used to debug the code, however for the purposes of this explanation, what was shown here is enough.

2.2.1 Projection in Minibase

This section will consider the code that is executed by `Query 1` in `JoinTest.java`.

1. Start by observing the content of `JoinsDriver`, specifically the format of the Relations involved in the DB. Most of the code in this function revolves around the construction of the DB and the insertion of tuples.
2. Insert a breakpoint next to `Query1` in `runTests`. Run the code in debugging mode: the debugger will stop on that line, without executing it.
3. `Step inside` function `Query1()` to follow the flow of execution. `Query1()` begins by printing the associated query. The query here is performing a join between the `Sailors` and `Reserves` tables and has a condition that the `boat_id` should be equal to `one`. Notice how we have **two** conditions here in the **WHERE** clause of the query. We then project on the **second** column of the `Sailors` tables and the **third** column of the `Reserves` tables.
4. Insert a breakpoint next to `Query1_CondExpr`, `Resume` then `Step inside` the function.

5. `Query1.CondExpr` enunciates the Join operations that will be executed by the DBMS while handling the query. In `JoinTest.java`, all queries are hardcoded to produce an object that can be digested by Minibase. The `CondExpr` class includes information about the operands used in a condition, their types, and the operation used. Since we have **two** conditions here, **three** objects of the `CondExpr` class are instantiated. (The last one is always **null** and indicates the end of conditions.) The instances are filled in `Query1.CondExpr()`. In the next assignments, you will be asked to create new queries that will have to follow the format shown here.
6. Open the **Variables** window to look at the content of `expr`. **Step over** each line and observe how `expr` changes to reflect the variables assignment. Continue with **Step over** until you leave `Query1.CondExpr` and land back in `Query1`.
7. Set a new breakpoint next to the line `FldSpec [] proj_list = new FldSpec[2];`. `proj_list` is filled with the columns over which the Projection will be executed, in the correct order. `proj_list` contains the two instances of `FldSpec` where the first instance corresponds to the second column of the outer table and the second instance corresponds to the third column of the inner table. The types of each projected column are specified in `jtype`.
8. Go to the declaration of `SortMerge` (Right click → **Open declaration**), this will open the file `SortMerge.java`. Observe how `proj_list` is used by the function `TupleUtils.setup_op_tuple` to create the joined tuple.
9. In `SortMerge.java`, find the evaluation of the tuples in the `get_next` function, `PredEval.Eval`. This function evaluates whether the tuples satisfy the conditions set in `OutputFilter`.
10. Set a breakpoint next to `Projection.Join`, then **Step inside**.
11. This function modifies the variable `Jtuple` by filling it with the values present in the outer and inner relations that were joined. The order of the columns is specified in `perm_mat`.
12. In **Window** → **Expressions**, you can see the content of the tuple by adding the expression `Jtuple.getStrFld(1)`. The value should be the name of the first sailor. `getStrFld` is a `Tuple` function, found in `heap/Tuple.java`.
13. Back in the `JoinTest.java` file, add a breakpoint next to `qcheck1.Check(t);` and **Resume**. The console should now show the first query result.
14. **Resume** the execution. If the breakpoint next to `PredEval.Eval` hasn't been removed, you will have to **Resume** multiple times as the `Eval` function evaluates queries that do not satisfy the join condition.

2.3 Assignment

The first assignment has the objective of getting the code to run on your machine and getting acquainted with the Minibase code.

1. To get acquainted with the code of Minibase, find the commands that add Sailors to their relation and search for how strings are handled when they are added to the heapfile. This can be done by using any debugger (e.g., Eclipse's debugger) with properly placed breakpoints. **It is strongly recommended to try and use the debugger to follow through the full execution of the program. Being able to debug your code will be very helpful for the next assignments.** The result of this step should **not** be reported in the typescript file.
2. In `tests`, modify the first query in “`JoinTest.java`” to select the sailors with boat number 2. Use the built-in docs in javadoc (`javaminibase/src/javadoc/index.html`) to navigate the methods, or the online resources at http://research.cs.wisc.edu/coral/mini_doc/project.html
3. From a Linux terminal, execute the jointest by running “`make test`” in the `src/tests` folder. If you are using an IDE, run the `JoinTest.java` file as an application.
4. As the query now returns 2 different tuples instead the original 3, a test is failing; fix it.

2.3.1 Submitting the assignment

The deliverable will be the output of the completed set of tests in file `JoinTest.java`. **All tests in `JoinTest` should pass** to complete the assignment (the other test files can be safely ignored for this part). You will then have to submit **individually** this output by uploading it as a text file on the Moodle platform.

- On Linux, we can save the output of `make test` using the command `script`. Once called, this command will save everything that is printed on the terminal into a file called `typescript` in the folder it was called in. Press “CTRL-D” to stop the `script` command and save the output. **Do not forget to stop the command, otherwise it will continue logging the output and produce a needlessly large file.**
- If you are running the code from Eclipse (or another IDE), copy the full output of the console after running the application and passing all the tests. Then, paste the output in a file called `typescript.txt`.
- Once the `typescript` file is ready, upload it on Moodle `moodle.eurecom.fr`.

2.4 FAQ from first virtual lab

Q1: Can I run Minibase on Windows?

A1: Minibase will not work well in Windows. Refer to Q4.

Q2: Is there a requirement on the Java version?

A2: No, any Java version works. Make sure Java and the JDK is installed on your system. Check [here](#)

Q3: The command `readlink -f $(which java)` returns the error

`readlink: illegal option -- f`

`usage: readlink [-n] [file ...]`

A3: You probably are using MacOS. Check this. The path of the Java dir should be similar to `/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk`.

Q4: How can I integrate Eclipse with WSL 2?

A4: It's easier to run Eclipse from an Ubuntu Virtualbox. Check [here](#)

Q5: The command `readlink -f $(which java)` returns the error `readlink: missing operand`.

A5: Type `java -version` to make sure that Java is installed. If it's not, refer to Q2.

Q6: The code is full of warnings and errors! How do I fix them?

A6: If you managed to correctly run either `JoinTest.java` as an application, or `make tests` you can safely ignore those warnings.

Q7: In Eclipse, there is no "Java Build Path" in the project properties.

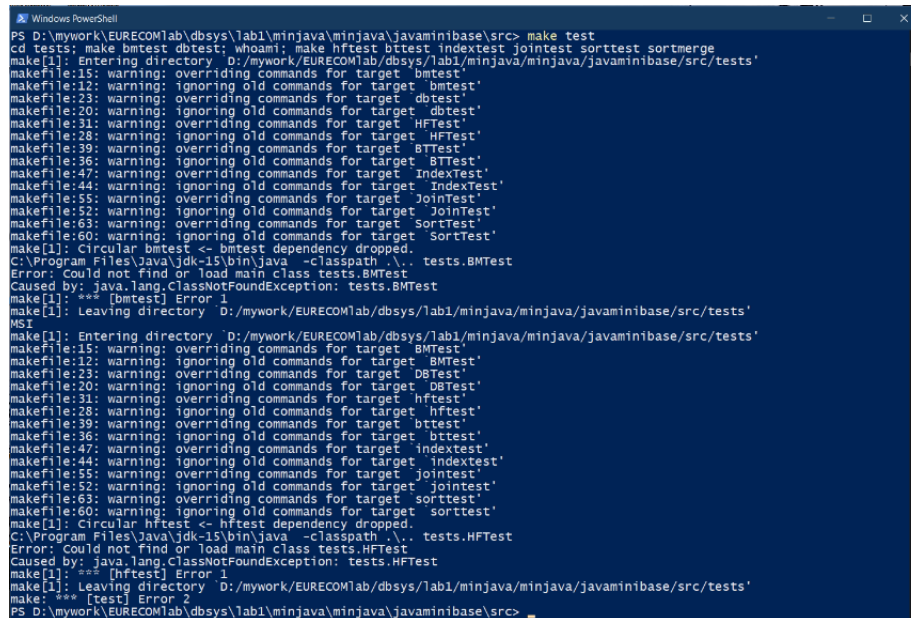
A7: Check Section 2.1.2 in the Lab doc and make sure you followed all steps. Make sure you correctly set the Project Natures.

Q8: I get the following error:

```
(base) spoutnik23@spoutnik23:~/TA/newminibase/minjava/javaminibase/src$ make test
cd tests; whoami; make jointest
spoutnik23
make[1]: Entering directory '/home/spoutnik23/TA/newminibase/minjava/javaminibase/src/tests'
/usr/lib/jvm/java-11-openjdk-amd64/bin/java/bin/javac -classpath ... TestDriver.java JointTest.java
make[1]: execvp: /usr/lib/jvm/java-11-openjdk-amd64/bin/java/bin/javac: Not a directory
Makefile:52: recipe for target 'JointTest' failed
make[1]: *** [JointTest] Error 127
make[1]: Leaving directory '/home/spoutnik23/TA/newminibase/minjava/javaminibase/src/tests'
Makefile:45: recipe for target 'test' failed
make: *** [test] Error 2
```

A8: Check the file `src/tests/Makefile` and modify the `JDKPATH` from `/usr/lib/jvm/java-11-openjdk-amd64/bin/java/bin/javac` to `/usr/lib/jvm/java-11-openjdk-amd64`. Check that `src/Makefile` has `/usr/lib/jvm/java-11-openjdk-amd64` as its `JDKPATH`.

Q9: I get the following error:



```
PS D:\mywork\EURECOMlab\dbsys\lab1\minjava\minjava\javaminibase\src> make test
cd tests; make bmtest dbtest; whoami; make hftest bftest indextest jointest sorttest sortmerge
make[1]: Entering directory 'D:/mywork/EURECOMlab/dbsys/lab1/minjava/minjava/javaminibase/src/tests'
makefile:15: warning: overriding commands for target 'bmtest'
makefile:12: warning: ignoring old commands for target 'bmtest'
makefile:23: warning: overriding commands for target 'dbtest'
makefile:20: warning: ignoring old commands for target 'dbtest'
makefile:31: warning: overriding commands for target 'hftest'
makefile:28: warning: ignoring old commands for target 'hftest'
makefile:39: warning: overriding commands for target 'bftest'
makefile:36: warning: ignoring old commands for target 'bftest'
makefile:47: warning: overriding commands for target 'indextest'
makefile:44: warning: ignoring old commands for target 'indextest'
makefile:55: warning: overriding commands for target 'jointest'
makefile:52: warning: ignoring old commands for target 'jointest'
makefile:63: warning: overriding commands for target 'sorttest'
makefile:60: warning: ignoring old commands for target 'sorttest'
make[1]: Circular bmtest <- bmtest dependency dropped.
C:\Program Files\Java\jdk-15\bin\java -classpath \.. tests.BMTest
Error: Could not find or load main class tests.BMTest
Caused by: java.lang.ClassNotFoundException: tests.BMTest
make[1]: *** [bmtest] Error 1
make[1]: Leaving directory 'D:/mywork/EURECOMlab/dbsys/lab1/minjava/minjava/javaminibase/src/tests'
PS D:\mywork\EURECOMlab\dbsys\lab1\minjava\minjava\javaminibase\src>
make[1]: Entering directory 'D:/mywork/EURECOMlab/dbsys/lab1/minjava/minjava/javaminibase/src/tests'
makefile:15: warning: overriding commands for target 'BMTest'
makefile:12: warning: ignoring old commands for target 'BMTest'
makefile:23: warning: overriding commands for target 'DBTest'
makefile:20: warning: ignoring old commands for target 'DBTest'
makefile:31: warning: overriding commands for target 'HFTTest'
makefile:28: warning: ignoring old commands for target 'HFTTest'
makefile:39: warning: overriding commands for target 'BFTTest'
makefile:36: warning: ignoring old commands for target 'BFTTest'
makefile:47: warning: overriding commands for target 'IndexTest'
makefile:44: warning: ignoring old commands for target 'IndexTest'
makefile:55: warning: overriding commands for target 'JointTest'
makefile:52: warning: ignoring old commands for target 'JointTest'
makefile:63: warning: overriding commands for target 'SortTest'
makefile:60: warning: ignoring old commands for target 'SortTest'
make[1]: Circular hftest <- hftest dependency dropped.
C:\Program Files\Java\jdk-15\bin\java -classpath \.. tests.HFTTest
Error: Could not find or load main class tests.HFTTest
Caused by: java.lang.ClassNotFoundException: tests.HFTTest
make[1]: *** [hftest] Error 1
make[1]: Leaving directory 'D:/mywork/EURECOMlab\dbsys\lab1\minjava\minjava\javaminibase\src\tests'
make: *** [test] Error 2
PS D:\mywork\EURECOMlab\dbsys\lab1\minjava\minjava\javaminibase\src>
```

A9: Check `src/Makefile` at lines 44-45 and make sure that they are the same as below:

```
test:
    cd tests; whoami; make jointest
```

3 Phase 2 - group

Deadline: 24th November 2020, 23:59 (GMT+2)

3.1 Reading Suggestions

This is a list of suggested steps before getting into the coding of the page replacement algorithms.

3.1.1 Before Starting

Review in the book and in the slides the parts about Disks, Files and Buffer Manager. Read the Javadoc Minibase documentation and the online material (<http://research.cs.wisc.edu/coral/minibase/project.html>) to understand how Java Minibase is put together and the APIs across its layers. Do not miss the description of the DB class and the diskmgr package, which are called by the buffer manager. You can look over the code under `diskmgr/` to learn more details.

3.1.2 Understanding the Data Structures

Before you start with your coding, it is strongly recommended to study `BufMgr.java` until you understand how the basic data structures and operations are implemented. The `BufMgr` class allocates pages (frames) for the buffer pool in main memory and conducts the basic operations, including the managing of the replacement policy (specified by `replacerArg`).

The buffer pool is a collection of frames (each frame is a page-sized sequence of main memory bytes) that is managed by the Buffer Manager. It is stored as an array `bufPool[numbuf]` of `Page` objects, defined as an array of bytes. We deal with buffer pool at the byte array level. Therefore, the buffer pool is declared as `bufpool[numbuf][page_size]`. The size of the pages is defined in the interface `GlobalConst` of the global package.

In addition, an array `FrameDesc[numbuf]` of descriptors is maintained, one per frame. Each descriptor is a record with the following fields: `page_number`, `pin_count`, and `dirtybit`. This describes the page that is stored in the corresponding frame. A page is identified by a `page_number` that is generated by the DB class when the page is allocated, and that is unique over all pages in the database. The `PageId` type is defined as an integer type. When a pin request is received, the buffer manager needs an efficient way to determine whether the page is already in the buffer pool. To handle this, a hash table is defined and used.

Before jumping into the basic operations, please be certain to understand how pages and descriptors are defined and manipulated within Minibase.

3.1.3 Understanding the Basic Operations

Understand the code for the following basic operations (look up on slides and book). You will find them in `BufMgr.java`:

```
public void pinPage(PageId pin_pgid, Page page, boolean emptyPage) {};  
  
public void unpinPage(PageId PageId_in_a_DB, boolean dirty) {};  
  
public PageId newPage(Page firstpage, int howmany) {};  
  
public void freePage(PageId globalPageId) {};  
  
public void flushPage(PageId pageid) {};  
  
public void flushAllPages() {};
```

3.1.4 Looking into the Test Code

Minibase declares a diskspace manager (DB), a buffer pool manager, etc., when initialized. Follow the code in the test file `BMTTest.java` to understand the flow and how the buffer manager is executed. This will lead you to look at other packages, such as `SystemDefs`. Look at the code and at the JavaDoc to see more details.

3.1.5 Clock

At this point you should be ready to understand the code for the “Clock” replacement policy, which is provided in `BufMgr.java`. This will lead you to study `Replacer.java`.

In the Clock policy, all the frames in the buffer pool are (conceptually) arranged in a circle around the face of a clock. Each frame is associated to a state “bit” which may be in one of three states: referenced, available or pinned.

Whenever the pin count of buffer frame `i` is greater than zero, then `state[i] = pinned` should be true. Each time a page in the buffer pool is unpinned, the state of the corresponding frame is set to Referenced (Intuition: postpone availability, as a page that was used recently has a high probability of being needed again soon). Whenever we need to choose a frame for replacement, the current frame pointer, or the “clock hand” (an integer whose value is between 0 and `numbuf-1`), is advanced, using modular arithmetic so that it does not go past `numbuf-1`. This corresponds to the clock hand moving around the face of the clock in a clockwise fashion.

For each unpinned frame that the clock hand encounters, the state bit is examined and eventually is changed from Referenced to Available. In fact, if the bit had been set to Referenced, it means that the corresponding frame has been referenced “recently” and so it is not replaced. On the other hand, if the bit is set to Available, the page is selected for replacement. Of course, if the selected

buffer frame is dirty (i.e. it has been modified), the page currently occupying the frame will have to be written to disk. While this is not as discerning as LRU, it offers some of the same advantages in this way.

Notice that, if clock has checked `2*NumBuffers`, this means that there were no available frames, even on the second pass through the buffer pool. When this happens, a check is used to prevent the program from entering an infinite loop: if this situation occurs, the code issues an error stating that there are no free frames in the buffer pool.

It is also useful to look at the more replacement strategies provided in the code: `LRU.java` and `MRU.java`, but it is not strictly needed at this point.

3.1.6 Test Clock

1. Look into the content of `BufMgr.java` class and make sure to understand its methods. Buffer Manager allocates new pages for the buffer pool and it does tasks like, pinning and unpinning the frames. It also uses the replacement algorithm to replace the page.
2. Check the class `Replacer.java`. It is an abstract class which provides basic methods for replacement algorithms.
3. In `BufMgr.java` class, find the class `Clock` which inherits from `Replacer`. It is important to understand the method `pick_victim`. This method is responsible to return the frame which should be replaced (based on the replacement policy).
4. Look into the class `BMTest.java`. This class includes methods to test different functionalities of the Buffer Manager. The method `runTests` in `BMTest.java` runs some predefined tests for replacement policies.
5. By defining the object `SystemDefs` we can pass some information about dbname, number of pages, buffer pool size and replacement policy. Make sure that `Clock` is defined as replacement policy.
6. `JavabaseBM` is a Buffer Manager object which is defined in `SystemDefs` and it provides Buffer Manager basic functions (creating new pages, pin, unpin and freeing pages).
7. Run `BMTest.java` to run the first 3 tests for Clock Replacer and make sure you understand what is happening at each step of these tests. By reading the comments you can see what is the purpose of each test. You can insert breakpoints next to each test and then step inside each one of them. To execute the test, follow the same procedure we used to run the join test in the first lab (i.e., modify the `makefile` script). Modify the test file as needed to print more information on screen.

3.1.7 Buffer Manager FAQ

1. Do we control the physical location of the buffer in main memory?

The 2d array `bufpool[numbuf][page_size]` is the buffer pool itself in Minibase: space in main memory controlled by the buffer pool manager to store (main-memory copies of) the database pages from disk.

Array `bufpool[numbuf]` is the array of “frames”. The buffer pool owns ‘numbuf’ number of frames in this case. Each entry here is a reference to an (implicit) array of ‘page_size’ number of bytes. Each such chunk of memory, a contiguous array of ‘page_size’ number of bytes, is precisely the size of memory needed to store a page. In the case of the primitive datatype byte (which is shorter than a pointer / reference), Java allocates directly an array of bytes, an exception to the general array-allocation rule of Java (based on references). So each array of bytes really is a contiguous chunk of memory.

However, we cannot control the full buffer pool to be one entire, contiguous span of main memory. A real database system relies on a diskspace manager writing into main memory directly, without the CPU being involved. We are not guaranteed this here. Since ‘`bufpool[numbuf]`’ is an array of references, the arrays of bytes (our frames) can be scattered throughout the heap. Java does not allow us closer control over memory allocation. However, each of our arrays of bytes (our frames) is contiguous.

This also explains why we make limited use of objects in these classes. Ideally, our frames for page placement would be contiguous in memory. But, if we made a frame object with the descriptor information and the page-space together, we are guaranteeing that this is not the case.

2. Can you provide more details on the error protocol in minibase?

Although the `Throwable` class in Java contains a snapshot of the execution stack of its thread at the time it was created and also a message string that gives more information about the error (exception), Minibase maintains a copy of its own stack, in order to have more control over the error handling.

The `chainexception` package handles the Minibase exception stack. Every exception created in the `bufmgr` package should extend the `ChainException` class. The exceptions are thrown according to the following rules:

- Error caused by an exception caused by another layer:

For example: (when try to pin page from diskmgr layer)

```
try {
    SystemDefs.JavabaseBM.pinPage(pageId, page, true);
} catch (Exception e) {
    throw new DiskMgrException(e, "DB.java: pinPage() failed");
}
```

- Error not caused by exceptions of others, but needs to be acknowledged.

For example: (when try to unpin a page that is not pinned)

```

    if (pin_count == 0) {
        throw new PageUnpinnedException (null, "BUFMGR: PAGE_NOT_PINNED.");
    }
}

```

Basically, the `ChainException` class keeps track of all the exceptions thrown across the different layers. Any new exception that you want to throw in your `bufmgr` package should extend the `ChainException` class.

3.2 First In First Out

In this section, you are required to code the FIFO (First In First Out) algorithm. To do so, you are first asked to understand LRU by reading the following subsection. After understanding the code for LRU, you should be ready to write the code for FIFO.

3.3 LRU

The implementation of the buffer manager maintains some bookkeeping information on each page in the buffer pool: “`pin_count`” and “`dirty`” variables. Every page in the buffer pool has a value for these variables. Whenever a page is requested by a user, the “`pin_count`” variable is incremented. Also, if any modification is made to a page after it enters the main memory, then the “`dirty`” variable is set to true, which indicates that the given page has to be written to disk before being removed from the buffer pool.

Any page replacement policy uses a set of methods to support its functionality. The functions are modified with respect to each of the page replacement policy. Have a look at `LRU.java` in the package `bufmgr` while reading the following.

update(int)

The buffer manager maintains a data structure called “`frames`” to identify the order in which frames are to be considered for swapping out of buffer-pool when needed. The order does not necessarily mean that the frames are swapped out in this order, instead depends on conditions such as whether or not the considered page is pinned. The `update(int)` method is used to keep the order updated. The input to this function is the frame number whose position in the frames is to be updated. In case of Least Recently Used (LRU) and Most Recently Used (MRU), this function is invoked every time a frame is pinned (allocation of a page) and during page replacement. In case of LRU policy, the frame number passed as an input is shifted to the last position in the frame to indicate that a) it is recently used and b) it should be considered only after considering all other pages already present in “`frames`”. In the case of MRU policy, this frame number is shifted to the first position in the frame (`frames[0]`), indicating that a) this is the most recently used frame and b) it

is to be considered for swapping out before any other frame in bufferpool.

setBufferManager(BufMgr)

The functions of this method initialize

1. The Buffer Manager object from the input parameter.
2. The number of buffers (number of frames the bufferpool can hold) from the Buffer Manager object.
3. State of all the frames to “Available”.

pin(int)

This function takes as input a frame number that needs to be pinned. It

1. increments the pin count of the given frame, and
2. sets the state of the frame number as “Pinned”.

In policies like LRU and MRU this function also requires us to update the position of the frames by making a call to the update function.

pick_victim()

This function returns to the invoker which frame to use in the bufferpool. It either picks a free frame and returns it or it applies any of the page replacement algorithms to determine which page is to be replaced. In LRU and MRU, the data structure “**frames**” maintains the order in which the replacement is to be considered. A scan is performed on “**frames**” and the first frame with 0 pin count is considered as victim for replacement. If there are no frames with pin count 0, it throws an exception indicating that buffer pool was exceeded.

3.3.1 FIFO

With this approach, the first page that was read to the memory buffer is considered for page replacement if its `pin_count=0`. This can be implemented by

1. maintaining a queue,
2. placing frames in it according to their order of entry to the buffer pool, and
3. picking the frame in front of the queue for flushing.

If the head of the queue is currently pinned with pin count greater than 0, then the next element in the queue is considered for flushing. The process goes on until a victim is found for replacement. If there are no frames in the queue with `pin_count==0`, then the replacement algorithm throws Buffer Pool Exceeded Exception.

When a frame gets into the buffer pool, it should be placed in such a way that order is preserved. E.g., `frames[0]` should contain the first page that

was read into the buffer pool, `frames[1]` the second page that was read and so on. When the first frame is chosen for replacement, the frame order has to be updated. When a frame already in buffer pool is requested for pinning, it is not necessary to call this function, and the order should not change based on its usage. If we do, then its functionality changes to that of a LRU policy. The difference between LRU and FIFO is that the position of frames in queue is updated for every pinning in case of LRU while it is updated only on first pinning in case of FIFO. The update function should be invoked only when a replacement is needed, not when pinning a page.

3.4 LRU-K

In this section, you are required to implement the LRU-K algorithm. The pseudo-code for the algorithm can be found in Fig. 2.1 of **this paper**.

Below are some remarks that would help you in the implementation:

- **LRU-K and time:** Some ambiguity comes from the fact that LRU-K can be implemented with two notions of "back", by pages or by time. It is recommended to implement the version in the paper, therefore by time. Using `System.currentTimeMillis()` is fine.
- **SystemDefs.java:** To accommodate the change to the constructor in the buffer manager, you are allowed to modify the global/`SystemDefs.java` which invokes the buffer manager constructor
- **Time comparisons:** If you have problems comparing times, try using `>=` instead of `>`, even if strict `>` is written in the original paper. This issue is due to the extremely small buffer we are testing.
- **"Correlated reference period" value:** For your first implementation of LRU-K, it is fine to use 0 as the constant for the correlated reference period. This makes the coding easier and can be revisited once you have a first working version.
- **Pages in LRU-K:** Assuming you are dealing only with pages in the current frames, you can implement it as it is done in LRU, by using only data structures referring to frames (basically treating them as pages). For example, you could use `HIST[frameNo][i]` and `LAST[frameNo]`

For testing **LRU-K**, you should download the file "`BMTTest2.java`" from Moodle. It contains 4 tests. Test 4 should be modified according to your code. This test reads all the pages in the bufferpool for the pages allocated and read to bufferpool using LRU-K algorithm.

Remark: If you are implementing your classes starting from LRU (or MRU) class, you may have a problem with how exceptions are handled. In particular, you may have a test failing because it is expecting to receive **Buffer Exceeded**

Exception but it is getting instead a **Replacer Exception** (as it is done in the Clock implementation).

The solution is to modify `public int pick_victim()` as follows:

1. replace `return -1` by `throw new
BufferPoolExceededException (null, "BUFMGR: BUFFER_EXCEEDED.");`.
2. add `throws BufferPoolExceededException` after the `public int pick_victim()`

DELIVERABLES

This is a **group** submission.

- Write a simple Notes.pdf (.txt is also ok) file, it should report comments about your code, such as explanations you find useful to understand how it works, issues or problems that you have not been able to solve. This document is intended to be very short and concise, low-level comments should be directly in the code.
- Create a directory, “Code” and copy BufMgr.java, FIFO.java, LIFO.java and LRU.java (plus any other java files that you modified and/or created), properly commented.
- Create a directory called “Output” and copy typescript (output of the BMTest2020.java execution) and “Notes.pdf” in it.
- Make sure to have a “README”-like file or section in the report with instruction to run your code.

Please ZIP your submission, name it as *Group_X*, where X is your group number, and upload it in Moodle under the assignment link (LRU-K).