

# Assignment 1

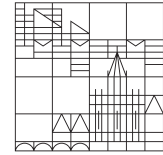
Issue Date: October 21, 2013

Due Date: October 28, 2013

Σ 20 Points

**Database System Architecture and Implementation**  
**INF-12950**  
**WS 2013/14**

Universität  
Konstanz



University of Konstanz  
Database and Information Systems  
Jun.-Prof. Dr. Michael Grossniklaus  
Andreas Weiler

## Disk Storage

### General Notes

Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises. Also take note of the general policies that apply to this course as listed on the course website <http://www.informatik.uni-konstanz.de/grossniklaus/education/dbmsinternals/>.

- Register for this course in **LSF** (to receive course-related e-mail) and in **StudIS** (to be admitted to the exam).
- Assignments are to be solved in pairs of **two** students.
- Send all of your submissions by e-mail to `andreas.weiler@uni-konstanz.de` with the subject line **[arch13] Asg# LastName1 LastName2**.
- All of your solutions must be handed in electronically. For written assignments, only **PDF documents** are accepted. Solutions provided in other formats, such as plain text, hand-written, and Word will not be graded. For programming assignments, submit an archive (ZIP file or similar) that only contains the **relevant** files.
- The use of external libraries is **not** permitted, except if they are explicitly provided by us.
- We use software to scan for duplicate submissions and plagiarism, so do **not** do it.

### Exercise 1: Hard Disks

(5 Points)



Suppose we have a hard disk with sectors of 512 bytes. Each track has 512 sectors, each platter surface has 4096 tracks, and the disk has 5 double-sided platters. The average seek time is 10 ms.

- a) How many bytes can be stored on the hard disk?
- b) What is the total number of cylinders?
- c) If the hard disk rotates with 7200 rpm (revolutions per minute), what is the maximum and the average rotation delay (in ms)?

### Exercise 2: Parities

(3 Points)



Assume that we are using a RAID-3 system with four hard disks. The following eight bits are stored on the first three data disks.

Disk 1: 10101010, Disk 2: 11001100, Disk 3: 11111111

- a) How will the parity sequence look like on the fourth disk?
- b) If Disk 2 gets damaged, how can the lost eight bits be restored?

### Exercise 3: Technical Terms

(5 Points)



Define the following terms in one or two sentences: a) Zone Bit Recording, b) S.M.A.R.T., and c) Wear Leveling.

### Exercise 4: Creating and Reading Files with Java

(7 Points)



Please download and unzip the file `asg01.zip` from the course website.

- Implement the methods `IO#create()` and `IO#read()`. Write the most efficient code you can think of!
- Did you expect the resulting performance? If yes/no, why (not)? Write a couple of sentences to explain the (sub-)optimal performance of your implementation.

# Assignment 2

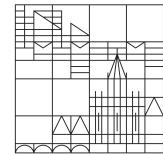
Issue Date: October 28, 2013

Due Date: November 04, 2013, 10:00 a.m.

Σ 20 Points

**Database System Architecture and Implementation**  
**INF-12950**  
**WS 2013/14**

Universität  
Konstanz



University of Konstanz  
Database and Information Systems  
Jun.-Prof. Dr. Michael Grossniklaus  
Andreas Weiler

## Disk Management

### General Notes

Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises. Also take note of the general policies that apply to this course as listed on the course website <http://www.informatik.uni-konstanz.de/grossniklaus/education/dbmsinternals/>.

- If you haven't done so yet, register yourself in the **LSF** and **StudIS**.
- From now on, please include **[arch13]** and your **group number** in the e-mail subject.

### Exercise 1: Free Blocks

(8 Points)



As you may know, most disk space managers use either a block list or a block bitmap to reference free and full database blocks on disk.

- Please name both an advantage and drawback for each concept.
- Let's say your database occupies 8192MB; all blocks are sized 16KB, and we use a block bitmap: how many KB are needed to reference all blocks?
- As block bitmaps are pretty small (see 1b), we can usually keep them in main memory. A single byte has 8 bits, so we store all bits in a byte array. Please write some efficient Java code that checks if a particular block is free:

```
boolean blockFree(byte[] bitmap, int blockNr) {  
    /** your code */  
}
```

### Exercise 2: Buffer Management: OS vs DB

(6 Points)



Databases and Operating Systems share a number of fancy concepts when it comes to the efficient access of data on hard disks.

- Invest some time for browsing books and web pages, and compile similarities between:
  - the virtual memory concept of operating systems and
  - the buffer management of databases
- Invest some more time and get creative: if you would write your own database...
  - would you rely on the existing OS features, or rather write your own buffer management?
  - what are the reasons for your choice?

### Exercise 3: Measuring Performance

(6 Points)



Please run the Test class in *asg02.zip*, and play around with the loop size.

- Which loop is evaluated faster in the sequential mode? Can you guess why?
- Implement the *parallel* method for executing the loops in parallel mode.
- Which loop is evaluated faster in the parallel mode? Can you guess why?
- Which mode is evaluated faster? Sequential or parallel? Can you guess why?

# Assignment 3

Issue Date: November 4, 2013

Due Date: November 18, 2013, 10:00 a.m.

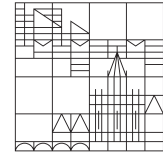
Σ 40 Points

Database System Architecture and Implementation

INF-12950

WS 2013/14

Universität  
Konstanz



University of Konstanz  
Database and Information Systems  
Jun.-Prof. Dr. Michael Grossniklaus  
Andreas Weiler

## Buffer Management



### General Notes

Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises. Also take note of the general policies that apply to this course as listed on the course website <http://www.informatik.uni-konstanz.de/grossniklaus/education/dbmsinternals/>.

- Please include **[arch13] Group #**, where # is your **group number**, in the subject line of the e-mail with your submission.
- Since this is a programming assignment, you will submit an archive (ZIP file or similar) that only contains the **relevant** source code files and a `README.txt` file (see below).
- The use of external libraries is **not** permitted, except if they are explicitly provided by us.
- Submissions with compile-time errors due to syntax or Checkstyle problems will **immediately** be disqualified. Whereas solutions with run-time errors will be considered, they will not result in a good grade.

### Prerequisites

In order to successfully complete this project, you will need to install the following software on the computer that you use for software development.

- **Java Development Kit**, available at <http://www.oracle.com/technetwork/java/javase/downloads/index.html> (version 6 or greater).
- **Apache Maven**, available at <http://maven.apache.org> (version 3.0.0 or greater).
- **Eclipse IDE for Java Developers**, available at <http://www.eclipse.org/downloads/> (we recommend the Kepler release, i.e., version 4.3.1).
- **Eclipse Checkstyle plug-in**, installation instructions at <http://eclipse-cs.sourceforge.net/downloads.html> (use the latest version).
- **Source Code Repository** (optional), contact us if you would like to use a repository to manage your code.

### Minibase

Minibase is a miniature relational DBMS, originally developed at the University of Wisconsin by Raghu Ramakrishnan to support the practical exercises of the text book that we use in our course. In total, there will be four assignments that are based on Minibase. In all of these assignments, you will implement simplified versions of different layers of a typical DBMS, without support for concurrency control or recovery.

We use an updated version of the code written by Chris Mayfield and Professor Walif Aref of Purdue University as well as Professor Leonard Shapiro of Portland State University. More recently, members of the Database and Information Systems group at the University of Konstanz have refactored and extended the Minibase source code for use in this course. The assignments themselves are partly based on assignments provided by Professor Kristin

Tufte of Portland State University, Christian Grün of the University of Konstanz, Chris Mayfield of Purdue University and Raghu Ramakrishnan of the University of Wisconsin. We thank everybody for the development of these exercises and their work on the Minibase code!

After you have installed and configured the required software (see above), you can set up a local copy of the Minibase source code distribution by following the steps below.

- **Download** the source code distribution archive from the course website and **unzip** it in a directory of your choice.
- Open a **command line** or **terminal** window and change into the directory (`cd`), where you have unzipped the source code. Make sure you are in the **root directory** of the source code distribution, which contains the file `pom.xml`.
- In order to **initialize the Maven project** and **create configuration files** for Eclipse, execute `mvn clean eclipse:clean eclipse:eclipse` from the command line prompt. Note that you need to have an Internet connection the first time you execute this command as Maven will not be able to download plug-ins and libraries otherwise. Should you be unfamiliar with Apache Maven, please refer to the documentation on the website listed above.
- Now you are ready to **launch Eclipse** and **import the Minibase project**. In order to do so, follow the steps of the import project wizard (File → Import... → General → Existing Projects into Workspace). In the wizard select the directory where you have unzipped the source code distribution under **Select root directory** and make sure that the option **Copy projects into workspace** is *not* selected.
- Once Eclipse has finished importing and compiling the project, there should be *no* errors, but there might be a couple of warnings. Make sure that **Checkstyle is activated** for the project by right-clicking on the project and selecting **Checkstyle → Activate Checkstyle**.
- **Congratulations!** If you have made it to this point without errors, you are ready to begin your programming project.

**Disclaimer:** Please note that Minibase is neither open-source, freeware, nor shareware. Refer to the file `COPYRIGHT.txt` in the `doc` folder of the Minibase distribution for terms and conditions.

## **i** Programming Practices

Checkstyle will already check the formatting of the code and enforce a certain coding convention. The Checkstyle configuration that we use is a slightly more relaxed version of the default Java coding convention. Note that you can configure Eclipse to ensure that your code is formatted accordingly as you write it (**Preferences...** → **Java** → **Code Style**). The source code distribution that you downloaded from the course website contains configuration files that can be imported to correctly configure the **Clean Up**, **Code Templates**, **Formatter**, and **Organize Imports** options.

Apart from the rules enforced by Checkstyle your code also needs to adhere to the following (good) programming practices.

- Use **meaningful names** for classes, interfaces, methods, fields, and parameters. For example, use `BufferManager` instead of `BMgr` or `page` instead of `p`.
- **Organize** your code by grouping it into (conceptual) blocks that are separated by a new line.
- Write a (Javadoc) **comment** before each method and paragraph of code.
- Provide a **comment** before each non-obvious declaration.
- Your code must be **understandable** by reading the comments only.
- Comments should not simply paraphrase what the code does, but rather **explain** it. For example, your comment for the line of code `clock = 0;` should not read “Assign 0 to variable *clock*”. A good comment would be “Initialize the clock to point to the first page in the buffer pool”.
- **Efficiency** is considered, e.g., do not use an `int` if a `bool` is appropriate, do not do a sequential search or a disk I/O unless necessary, etc.

## **i** Assignment Overview

You will be given parts of the Minibase code, in Java, and asked to fill in other parts of it. You may spend 90% of your time understanding the given code and 10% of your time writing new code.

The best place to start understanding this assignment are the lecture slides and the text book (Sections 9.3 and 9.4) about disk space manager and the buffer manager. Following the reference architecture presented in the lecture,

Minibase is structured in layers. Each layer corresponds to a Java package. For example, the disk manager is located in the `minibase.storage.file` package. All of these packages are managed as subdirectories of `src/main/java`, where directory `src` is directly located in the root directory of the Minibase distribution. For each package of the main Minibase system, there is a corresponding package of the same name that contains JUnit tests, which you will use to check the correctness of your implementation. The JUnit tests are located in subdirectories of `src/test/java`. If you have successfully used Apache Maven to create Eclipse project files, all of these directory will be added as source folders to your project when you import it into Eclipse.

In this assignment, you will be given the disk space manager layer (`minibase.storage.file`), which manages pages in a file on the disk. Before you start implementing you should study the existing code and make sure that you have an in-depth understanding of how it works. Your first task is to write the next higher layer, i.e., the buffer manager layer (`minibase.storage.buffer`), which builds on functionality from the disk space manager layer. Note that everything is not perfect in the disk space manager layer as there is a method in class `DiskManager` that calls a method in class `BufferManager`, which violates the layering approach. Your second task is to implement alternative buffer replacement policies to be used in the buffer manager.

### Exercise 1: Buffer Manager

(25 Points)



As mentioned above, your first task is to implement the Minibase buffer manager. In the `minibase.storage.buffer` package, you will find a skeleton class called `BufferManager` that contains all method you will need to implement. Currently, all methods have been implemented to throw an `UnsupportedOperationException`, i.e., they will simply fail when invoked. Look through all the methods signatures and read the comments provided for each method.

Once you understand the structure of the buffer manager and the functionality that it will provide, you will begin by implementing the constructor of the class `BufferManager`. The main task of this constructor is to initialize the buffer pool as an array of `Page` objects, which will be referred to as *frames*.

Each frame has certain states associated with it. These state indicate whether a frame is dirty, whether it includes valid data, i.e., data that reflects the data in a disk page, and, if it includes valid data, what is the disk page number of the data, how many callers have pins on the data, i.e., the pin count, and any other information you wish to store, for example information relevant to the replacement algorithm. Be sure to store this information as efficiently as possible while preserving readability. We recommend that you store these states in a structure represented by a separate class called `FrameDescriptor` that are managed in a *frame table* (`frameTable`), which has an entry for each frame.

Your `BufferManager` class will need to determine, very efficiently, what frame a given disk page occupies. You may wish to use Java's `HashMap` class to map a disk page number to a frame descriptor. This *mapping* will also tell you, if a disk page is not in the buffer pool. As you write methods such as `pinPage()` it is very important, and a prime source of bugs, to keep the frame table and your mapping current.

Your primary goal in this exercise is to fill in the stubs in `BufferManager.java` so that the tests in `BufferManagerTest.java` run successfully, and so that the specifications above and in the documentation are met. In your solution, you may add any other classes or files to the `minibase.storage.buffer` package. For example, you will need to create a file `FrameDescriptor.java` for the class that represents the state of a buffer frame as described above. Study the code for `DiskManagerTest.java` in the tests package. This may be useful to see how to call the methods in `DiskManager`, and to verify that `DiskManager` actually works. Look even more carefully at `BufferManagerTest.java`. Note that in `TestDriver.java` `DB_SIZE` is set to 20000 and `BUF_SIZE` is set to 100. You may wish to set these artificially lower for debugging purposes.

### Exercise 2: Buffer Replacement Policies

(15 Points)



You can find an existing buffer replacement policy in the `minibase.storage.buffer` package that selects a random victim (`RandomPolicy`). The buffer replacement strategy can be configured in `TestDriver.java`. We recommend you use this replacement policy to test your implementation of the first exercise. In order for the buffer replacement policy to work correctly, you will need to call its methods `freePage()`, `pinPage()`, and `unpinPage()` at the correct positions in the code of `BufferManager`. These methods are intended to inform the policy about the state of the buffer pool and to enable it to update its internal data structures.

Once you have made sure that the buffer manager and the policy are synchronized, you can use the following line of code in your implementation of the `pinPage()` method to figure out which page is best to replace.

```
final int frameNo = this.replacementPolicy.pickVictim();
```

As you can see from looking at the code of `ReplacementPolicyFactory`, Minibase has been designed to support a number of different buffer replacement strategies. As soon as you feel confident that your buffer manager works correctly with the random policy, your second exercise is to implement the three missing strategies, i.e., LRU, MRU, and Clock. The different algorithms for these policies have been discussed in the lecture, but are also repeated below. For each algorithm, you will create a new subclass of `AbstractReplacementPolicy` that implements the methods defined in interface `ReplacementPolicy` according to these algorithms. The names of the new classes will be `LRUPolicy`, `MRUPolicy`, and `ClockPolicy`.

### LRU: Least Recently Used

*Initialization:* The LRU policy maintains a queue that is initialized with the frame descriptors  $1 \dots N$ .

*Update:* If the `pinCount` of a page in a frame becomes 0, the frame descriptor is moved to the end of the queue.

*Pick Victim:* Return the index of the first frame in the queue containing a page with `pinCount == 0`, remove that frame from the queue, and append it to the end.

### MRU: Most Recently Used

*Initialization:* The MRU policy maintains a stack that is initialized with the frame descriptors  $N \dots 1$ . *Update:* If the `pinCount` of a page in a frame becomes 0, the frame descriptor is moved to the top of the stack.

*Pick Victim:* Return the index of the first frame in the stack containing a page with `pinCount == 0`, remove that frame from the stack, and move it to the top of the stack.

### Clock: Second Chance

*Initialization:* For every page, the Clock policy maintains a flag whether the page is *referenced*. Additionally, the clock hand `current` needs to be initialized to 0.

*Update:* If the `pinCount` of a page in a frame becomes 0, mark the page in this frame descriptor as referenced.

*Pick Victim*

```
for(int counter = current; counter < 2 * N; counter++) {
    set current to counter, mod N
    if (data in bufferPool[current] is not valid) choose current;
    if (frameTable[current].pinCount == 0) {
        if frameTable[current].referenced == true {
            set frameTable[current].referenced = false
        } else {
            choose current
        }
    }
}
could not find an available frame, return an error
```

## Submission

Solutions are submitted electronically by e-mail. Your submission must consist of a single zipped archive that contains all the relevant Java files, i.e., all files that you have modified or created, structured in the appropriate packages. The name of the file containing your submission should be `grp#-asg#.zip`, where # is your group and the assignment number, respectively. You will also include a file called `README.txt` that should contain the following information.

**Overall Status** A one to two paragraph overview of how you implemented the major components. If you were unable to finish the project, please give details about what is and is not complete. Be short and to the point!

**File Descriptions** If you have created any new files, list their names and a short description.

**Time Spent** Please include how many hours you spent on this project. Note that the time spent has no impact on your grade. This information will only be used in planning future projects.

# Assignment 4

Issue Date: November 18, 2013

Due Date: November 25, 2013, 10:00 a.m.

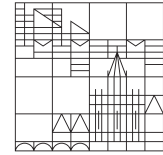
Σ 20+5 Points

Database System Architecture and Implementation

INF-12950

WS 2013/14

Universität  
Konstanz



University of Konstanz  
Database and Information Systems  
Jun.-Prof. Dr. Michael Grossniklaus  
Andreas Weiler, Leo Wörteler

## B+ Trees

### i General Notes

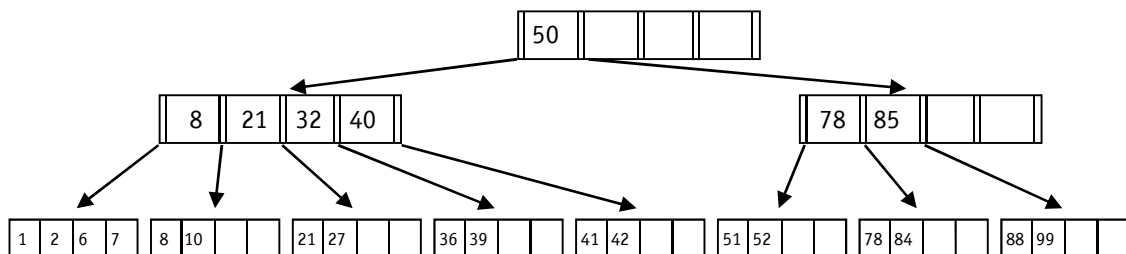
Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises. Also take note of the general policies that apply to this course as listed on the course website <http://www.informatik.uni-konstanz.de/grossniklaus/education/dbmsinternals/>.

- From now on, please include [arch13] and your **group number** in the e-mail subject.
- Write your **names** and **group number** in all files that you submit.
- Solutions must be submitted **before the deadline** published on the website and the assignment.
- Submissions with compile-time errors due to syntax or Checkstyle problems will **immediately** be disqualified. Whereas solutions with run-time errors will be considered, they will not result in a good grade.

### Exercise 1: Understanding B+ Trees

(5 Points)

Have a look at the following B+ Tree with order  $d = 2$ :



For the following three operations, sketch the (relevant part of the) resulting B+ Tree and count how many nodes have to be read, written, created and deleted. For this exercise you can ignore any effects of the BUFFER MANAGER.

- Value 4 is inserted, and neighbors are checked for redistribution.
- Value 5 is inserted in the original tree without checking the neighbors.
- Value 88 is deleted in the original tree with the neighbors checked.

## Exercise 2: Implementing a B+ Tree

(15 Points)



Your task is to implement a main memory variant of a B+ Tree for integer values. The data structure will support the following features.

- `AbstractBTree#contains(int value)` looks up if a value exists
- `AbstractBTree#insert(int value)` adds new values to the tree structure
- `AbstractBTree#delete(int value)` removes existing values structure
- `AbstractBTree#size()` returns the number of values stored

For this task, please download the file `asg04.zip` from the course website, which contains the framework that you need to use.

- Your submission needs to be a single **public class** `BTree##` **extends** `AbstractBTree` (where `##` is your group number), which correctly implements all abstract methods of `AbstractBTree`.
- The framework deliberately avoids the use of objects in favor of primitive data types to simulate paged disk access. All nodes are stored in a central map and referenced by their unique ID. When inserting and deleting values, take care that each node contains at least  $d$  and at most  $2 \cdot d$  nodes, where  $d$  is the degree of the B+ Tree as specified in the constructor.
- Implement the algorithms without using existing Java data structures, such as `Collections`, and do not modify any of the classes other than the one you will submit. Make sure to comment your code sufficiently and to remove any warnings and errors (of `CheckStyle` as well as `Eclipse`), as they will cost you points. Only compiling solutions will be graded.

## Exercise 3: Optimizing your B+ Tree (Bonus)

(+5 Points)



Implement additional optimizations that reduce the number of necessary splits and merges of nodes.

- In `insertValue`, before splitting the node, the neighbors can be checked. If one of them is not full, entries are shifted over to it and the new entry can be inserted without overflow.
- In `deleteValue`, it is more efficient to check both neighbors before deciding to merge two nodes.
- While it is sufficient to move one entry from or to the neighbor in the above optimizations, it is better to distribute the entries evenly so that (optimally) both nodes are neither full nor empty. This enables subsequent insertions as well as deletions without rebalancing.



## i B+ Tree – Details

In the following, you find three algorithms that might help you to implement the B+ Tree. Note that some additional operations have to be performed, so it will not suffice to convert the pseudo code to Java:

```
function containsValue (int nodeID, int value) : boolean
    node ← get the node for nodeID;
    if node is a leaf node then
        if value is found in node then
            | return true
        else
            | return false
    else // node is a branch node, continue with child
        childID ← child ID with greatest key  $k$  so that  $k \leq \text{value}$  in node;
        return containsValue (childID, value)
```

**Algorithm 1:** Checking if a value is contained in a B+ Tree.

```
function insertValue (int nodeID, int value) : long
    node ← get the node for nodeID;
    if node is a leaf node then
        if node already contains value then
            | return NO_CHANGES
        else // value has to be inserted
            increment the number of values stored in the tree;
            if some space is left then
                insert the value into node;
                return NO_CHANGES
            else // node has to be split
                rightID ← create a new leaf node;
                right ← the node with ID rightID;
                distribute values between node and right;
                return keyIDPair (firstValue (right), rightID)
    else // node is a branch node
        childID ← child ID with greatest key  $k$  so that  $k \leq \text{value}$  in node;
        result ← insertValue (childID, value);
        if result = NO_CHANGES then // nothing to do
            | return NO_CHANGES
        else // child was split
            midKey ← getMidKey (result);
            rightID ← getChildID (result);
            if some space is left in node then
                insert the rightID into node's child ID list;
                insert the midKey into node's value list;
                return NO_CHANGES
            else // node is full and has to be split
                rightID ← create a new branch node;
                right ← the node with ID rightID;
                distribute values and child pointers in node and right;
                midKey' ← middle value between those in node and right;
                return keyIDPair (midKey', rightID)
```

**Algorithm 2:** Inserting a value into a B+ Tree.

```

function deleteValue (int nodeID, int value) : boolean
    node ← get the node for nodeID;
    if node is a leaf node then
        if value is not found in node then
            | return true
        else // value is found
            delete value from node;
            decrement the number of values stored in the tree;
            if node still has enough values stored then
                | return true
            else // node is under-full
                | return false
    else // node is a branch node
        childID ← child ID with greatest key  $k$  so that  $k \leq \text{value}$  in node;
        result ← deleteValue (childID, value);
        if result = true then // nothing to do
            | return true
        else // the child has become under-full
            child ← get node with ID childID;
            if child is the only child of node then // node must be the root
                | delete the old root node;
                | set child as the new root node;
                | return true
            else // child has a neighbor below node
                neighbor ← get a neighbor of child;
                if neighbor has more than  $d$  values then
                    | re-fill child by borrowing from neighbor;
                    | return true
                else // neighbor is minimally full
                    | merge child with neighbor;
                    | adjust the pointers and values of node;
                    | delete child;
                    | if node still has enough values stored then
                        | | return true
                    | else // node is under-full
                        | | return false

```

**Algorithm 3:** Deleting a value from a B+ Tree.

### **i** Array Handling in Java:

Some useful utility methods (e.g., for searching and filling) can be found in the class `java.util.Arrays`. For efficiently copying/moving entries in arrays, the method

`System.arraycopy(Object src, int srcOffset, Object dest, int destOffset, int length)` is the one to be used:

- Insert a value at position `pos`:

```

System.arraycopy(array, pos, array, pos + 1, nrValues);
array[pos] = value;

```

- Double the size of an array to make room for more entries:

```

int[] temp = new int[array.length * 2];
System.arraycopy(array, 0, temp, 0, array.length);
array = temp;

```

This can also be done in the following way:

```

array = Arrays.copyOf(array, array.length * 2);

```

# Assignment 5

Issue Date: November 25, 2013

Due Date: December 09, 2013, 10:00 a.m.

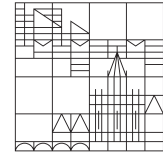
Σ 40+5 Points

Database System Architecture and Implementation

INF-12950

WS 2013/14

Universität  
Konstanz



University of Konstanz  
Database and Information Systems  
Jun.-Prof. Dr. Michael Grossniklaus  
Andreas Weiler, Leo Wörteler

## Disk-based B<sup>+</sup> Trees



### General Notes

Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises. Also take note of the general policies that apply to this course as listed on the course website <http://www.informatik.uni-konstanz.de/grossniklaus/education/dbmsinternals/>.

- Please include **[arch13] Group #**, where # is your **group number**, in the subject line of the e-mail with your submission.
- Since this is a programming assignment, you will submit an archive (ZIP file or similar) that only contains the **relevant** source code files and a `README.txt` file (see below).
- The use of external libraries is **not** permitted, except if they are explicitly provided by us.
- Submissions with compile-time errors due to syntax or Checkstyle problems will **immediately** be disqualified. Whereas solutions with run-time errors will be considered, they will not result in a good grade.

### Prerequisites

In order to successfully complete this project, you will need to install the following software on the computer that you use for software development.

- **Java Development Kit**, available at <http://www.oracle.com/technetwork/java/javase/downloads/index.html> (version 6 or greater).
- **Apache Maven**, available at <http://maven.apache.org> (version 3.0.0 or greater).
- **Eclipse IDE for Java Developers**, available at <http://www.eclipse.org/downloads/> (we recommend the Kepler release, i.e., version 4.3.1).
- **Eclipse Checkstyle plug-in**, installation instructions at <http://eclipse-cs.sourceforge.net/downloads.html> (use the latest version).
- **Source Code Repository** (optional), contact us if you would like to use a repository to manage your code.

### Minibase

Minibase is a miniature relational DBMS, originally developed at the University of Wisconsin by Raghu Ramakrishnan to support the practical exercises of the text book that we use in our course. In total, there will be four assignments that are based on Minibase. In all of these assignments, you will implement simplified versions of different layers of a typical DBMS, without support for concurrency control or recovery.

We use an updated version of the code written by Chris Mayfield and Professor Walif Aref of Purdue University as well as Professor Leonard Shapiro of Portland State University. More recently, members of the Database and Information Systems group at the University of Konstanz have refactored and extended the Minibase source code for use in this course. The assignments themselves are partly based on assignments provided by Professor Kristin

Tufte of Portland State University, Christian Grün of the University of Konstanz, Chris Mayfield of Purdue University and Raghu Ramakrishnan of the University of Wisconsin. We thank everybody for the development of these exercises and their work on the Minibase code!

After you have installed and configured the required software (see above), you can set up a local copy of the Minibase source code distribution by following the steps below.

- **Download** the source code distribution archive from the course website and **unzip** it in a directory of your choice.
- Open a **command line** or **terminal** window and change into the directory (`cd`), where you have unzipped the source code. Make sure you are in the **root directory** of the source code distribution, which contains the file `pom.xml`.
- In order to **initialize the Maven project** and **create configuration files** for Eclipse, execute `mvn clean eclipse:clean eclipse:eclipse` from the command line prompt. Note that you need to have an Internet connection the first time you execute this command as Maven will not be able to download plug-ins and libraries otherwise. Should you be unfamiliar with Apache Maven, please refer to the documentation on the website listed above.
- Now you are ready to **launch Eclipse** and **import the Minibase project**. In order to do so, follow the steps of the import project wizard (File → Import... → General → Existing Projects into Workspace). In the wizard select the directory where you have unzipped the source code distribution under **Select root directory** and make sure that the option **Copy projects into workspace** is *not* selected.
- Once Eclipse has finished importing and compiling the project, there should be *no* errors, but there might be a couple of warnings. Make sure that **Checkstyle is activated** for the project by right-clicking on the project and selecting **Checkstyle → Activate Checkstyle**.
- **Congratulations!** If you have made it to this point without errors, you are ready to begin your programming project.

**Disclaimer:** Please note that Minibase is neither open-source, freeware, nor shareware. Refer to the file `COPYRIGHT.txt` in the `doc` folder of the Minibase distribution for terms and conditions.

## **i** Programming Practices

Checkstyle will already check the formatting of the code and enforce a certain coding convention. The Checkstyle configuration that we use is a slightly more relaxed version of the default Java coding convention. Note that you can configure Eclipse to ensure that your code is formatted accordingly as you write it (**Preferences...** → **Java** → **Code Style**). The source code distribution that you downloaded from the course website contains configuration files that can be imported to correctly configure the **Clean Up**, **Code Templates**, **Formatter**, and **Organize Imports** options.

Apart from the rules enforced by Checkstyle your code also needs to adhere to the following (good) programming practices.

- Use **meaningful names** for classes, interfaces, methods, fields, and parameters. For example, use `BufferManager` instead of `BMgr` or `page` instead of `p`.
- **Organize** your code by grouping it into (conceptual) blocks that are separated by a new line.
- Write a (Javadoc) **comment** before each method and paragraph of code.
- Provide a **comment** before each non-obvious declaration.
- Your code must be **understandable** by reading the comments only.
- Comments should not simply paraphrase what the code does, but rather **explain** it. For example, your comment for the line of code `clock = 0;` should not read “Assign 0 to variable *clock*”. A good comment would be “Initialize the clock to point to the first page in the buffer pool”.
- **Efficiency** is considered, e.g., do not use an `int` if a `bool` is appropriate, do not do a sequential search or a disk I/O unless necessary, etc.

## **i** Assignment Overview

You will be given parts of the Minibase code, in Java, and asked to fill in other parts of it. You may spend 90% of your time understanding the given code and 10% of your time writing new code.

Minibase is structured in layers that follow the reference architecture presented in the lecture. Each layer corresponds to a Java package. All of these packages are managed as subdirectories of `src/main/java`, where

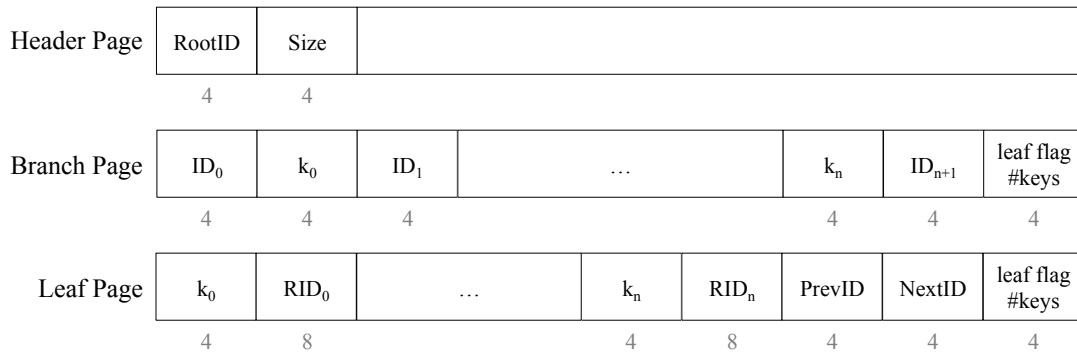


Figure 1: B<sup>+</sup> page layouts (sizes in bytes are given below each field)

directory `src` is directly located in the root directory of the Minibase distribution. For each package of the main Minibase system, there is a corresponding package of the same name that contains JUnit tests, which you will use to check the correctness of your implementation. The JUnit tests are located in subdirectories of `src/test/java`. If you have successfully used Apache Maven to create Eclipse project files, all of these directories will be added as source folders to your project when you import it into Eclipse.

The best place to start understanding this assignment are the lecture slides and the text book (Sections 10.3–10.8) about B<sup>+</sup> trees. In this assignment, you will work on the access layer, which contains all file organizations supported in Minibase and is located in package `minibase.access`. For example, Minibase’s heap file implementation can be found in `minibase.access.heap` and a rudimentary hash-based index is located in `minibase.access.hash`. Finally, package `minibase.access.index` contains general classes to support index access structures. Your task is to implement B<sup>+</sup> tree index for Minibase in package `minibase.access.btree`. Specifically, you will complete the unimplemented methods in the skeleton class `BTreeXX`, where XX is your group number. The package already contains the classes listed below that you will use in your implementation. Do *not* alter these classes and do *not* create any new classes.

**BTreeIndex** The (abstract) main class of the B<sup>+</sup> tree implementation. The class `BTreeXX` that you will implement is a subclass of this class.

**BTreeHeader** Use the methods of this class to interpret the header page of the B<sup>+</sup> tree. The layout of the header page is shown in Figure 1. It currently only contains the id of the root node and the size of the B<sup>+</sup> tree.

**BTreeBranch** Use the methods of this class to interpret non-leaf nodes (i.e., branch pages) of the B<sup>+</sup> tree. The layout of branch pages is shown in Figure 1. The ID fields point to the child nodes of a given node and the k fields contain the separator keys. The last field contains a flag that indicates whether the current node is a leaf or not (leaf flag) as well as the total number of keys stored in the node (#keys).

**BTreeLeaf** Use the methods of this class to interpret leaf nodes of the B<sup>+</sup> tree. The layout of leaf pages is shown in Figure 1. The k fields contain the stored keys and the RID fields contain pointers to the corresponding record ids. Since leaf nodes form a doubly-linked list, a leaf node also contains pointers to the previous (prevID) and next (nextID) leaf node in that list. The last field contains a flag that indicates whether the current node is a leaf or not (leaf flag) as well as the total number of keys stored in the node (#keys).

**BTreePage** This class contains methods to interpret B<sup>+</sup> tree pages that are common to `BTreeBranch` and `BTreeLeaf`.

Before you start implementing you should study the existing code and make sure that you have an in-depth understanding of how it works. In order to keep matters reasonably simple, the Minibase B<sup>+</sup> tree implementation *only* supports `int` keys. For a maximum grade, your implementation should support the following B<sup>+</sup> tree characteristics.

- **Sequence Set:** Your implementation should organize leaf nodes in a *doubly-linked list*.
- **Split and Merge:** To deal with overflow and underflow, your implementation should support both splitting and merging of non-leaf and leaf nodes.
- **Redistribution:** Minimally, your implementation should support redistribution of a single entry at the leaf level. For bonus points, your implementation could additionally support 50/50 rebalancing of nodes through redistribution and/or redistribution for non-leaf nodes.

**Hint:** In contrast to the previous assignment, i.e., the in-memory B<sup>+</sup> tree implementation, one major challenge of this assignment is to *correctly* pin and unpin the memory pages that contain the data of the B<sup>+</sup> tree. We recommend that you inspect the other access structures supported by Minibase, in particular the heap file, to understand how to pin and unpin pages when working with a complex disk-based data structure.

### Exercise 1: B<sup>+</sup> Tree Search

(5 Points)



To complete the search functionality of your B<sup>+</sup> tree, you need to implement the following two methods.

- `int findKey(Page page, int key, boolean leaf)`: This method implements the *binary search* for the given key within a single node stored in page. The Boolean flag `leaf` indicates whether the node follows the leaf or branch node layout. If the key is found, the method returns the position of the key within the node and `-(insPos + 1)`, otherwise. **Hint:** The behavior of `findKey` follows the standard behavior of `java.util.Arrays#binarySearch()`.
- `Page search(PageID pageID, int key)`: This method implements the recursive search for the given key, starting at the node stored in the page with id `pageID`. It returns the leaf node, where the key belongs. Note that the actual check whether the key is contained in this leaf node is performed by method `BTreeIndex#lookup()`, which also initializes the recursion to begin at the root page, identified by `rootID`.

### Exercise 2: B<sup>+</sup> Tree Insert

(15 Points)



To complete the insert functionality of your B<sup>+</sup> tree, you need to implement the following method.

- `Entry insert(final PageID pageID, final int key, final RecordID value)`: This method implements the recursive insert of the given index entry consisting of an integer key and a record id value, starting at the page with id `pageID`. In case of a split, the method returns an `Entry` that has to be inserted into the parent. If an entry with the given key already exists in the B<sup>+</sup> tree, the corresponding record id should be updated. Note that again the recursion is initialized to start at the root page, identified by `rootID`, from an existing method of class `BTreeIndex`.

### Exercise 3: B<sup>+</sup> Tree Delete

(20 Points)



To complete the delete functionality of your B<sup>+</sup> tree, you need to implement the following method.

- `boolean delete(final int key, final PageID pageID)`: This method implements the recursive delete of a given entry, identified by its key, from the B<sup>+</sup> tree, starting at the page with id `pageID`. The method returns `true` if the page is still full enough, i.e., does not have an underflow, after the deletion of the entry. Otherwise, a redistribution or merge needs to be performed. Note that again the recursion is initialized to start at the root page, identified by `rootID`, from an existing method of class `BTreeIndex`.

### Exercise 4: B<sup>+</sup> Tree Iterator (Bonus)

(5 Points)



As a bonus exercise you can implement an iterator that scans through all record ids indexed in the B<sup>+</sup> tree in sorted order. To help you with the completion of this task, we have provided a skeleton implementation of this iterator in class `BTreeIteratorXX`.

### Submission

Solutions are submitted electronically by e-mail. Your submission must consist of a single zipped archive that contains all the relevant Java files, i.e., class `BTreeXX` and `BTreeIteratorXX` (if you chose to implement the bonus exercise), where XX is your group number. The name of the file containing your submission should be `grpXX-asgYY.zip`, where XX and YY are your group and the assignment number, respectively. You will also include a file called `README.txt` that should contain the following information.

**Overall Status** A one paragraph overview of the status of your implementation. If you were unable to finish the project, please give details about what is and is not complete. Be short and to the point!

**B<sup>+</sup> Characteristics** A one paragraph summary detailing the B<sup>+</sup> tree characteristics (see above) you support in your implementation.

**Time Spent** Please include how many hours you spent on this project. Note that the time spent has no impact on your grade. This information will only be used in planning future projects.

### **i Assignment 3**

Looking back at Assignment 3, you can get an indication of the time you spent in comparison to the time spent by other groups from Figure 2.

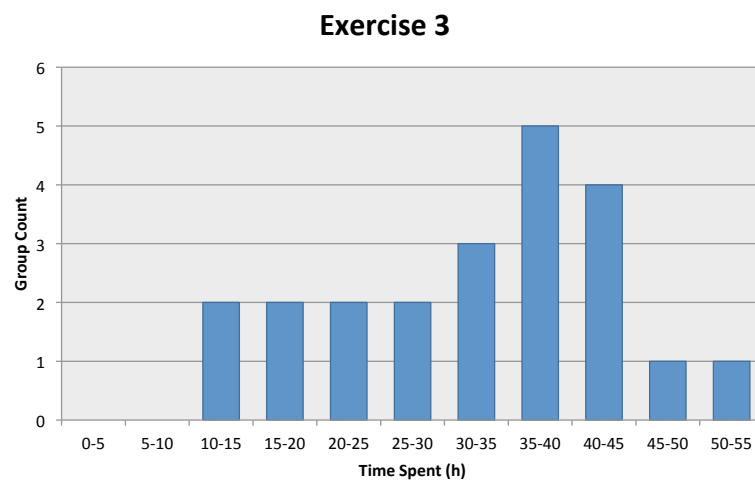


Figure 2: Overview of time spent on Assignment 3

# Assignment 6

Issue Date: December 09, 2013

Due Date: December 16, 2013, 10:00 a.m.

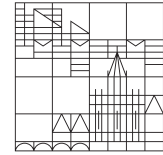
Σ 20 Points

Database System Architecture and Implementation

INF-12950

WS 2013/14

Universität  
Konstanz



University of Konstanz  
Database and Information Systems  
Jun.-Prof. Dr. Michael Grossniklaus  
Andreas Weiler, Leo Wörteler

## Hashing

### i General Notes

Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises. Also take note of the general policies that apply to this course as listed on the course website <http://www.informatik.uni-konstanz.de/grossniklaus/education/dbmsinternals/>.

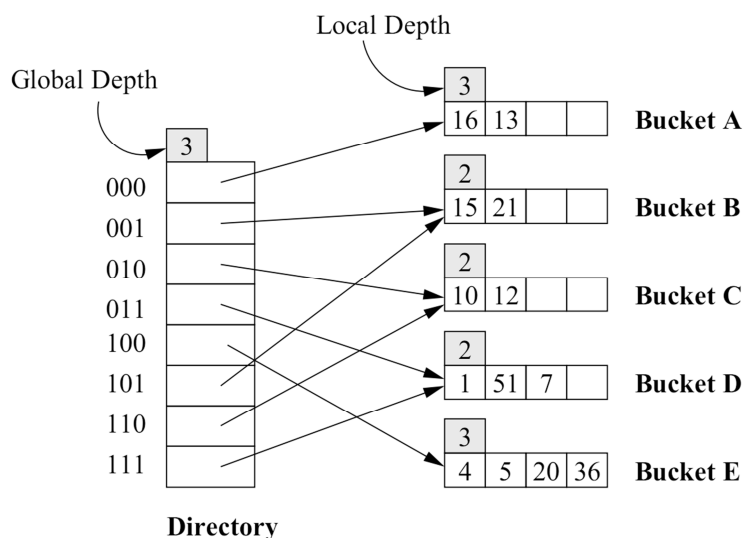
- From now on, please include [arch13] and your **group number** in the e-mail subject.
- Write your **names** and **group number** in all files that you submit.
- Solutions must be submitted **before the deadline** published on the website and the assignment.

### Exercise 1: Extendible Hashing

(8 Points)




The extendible hash index given below has some flaws w.r.t. the mapping of the data values to the hash buckets:



- Fix the flaws in the extendible hash index.
- Sketch the index of a) after inserting an entry with hash value 68.
- Sketch the index of a) after inserting an entry with hash value 17.
- Sketch the index of a) after inserting an entry with hash value 16.




**Exercise 2: Linear Hashing**(8 Points) 


Consider the linear hashing index shown below. Assume that we split the bucket located at the table position *Next* whenever an overflow page is created:

<b>h(1)</b>	<b>h(0)</b>	<b>Primary Pages</b>	<b>Overflow Pages</b>				
000	00	<table border="1"><tr><td>16</td><td>40</td><td>8</td><td></td></tr></table>	16	40	8		
16	40	8					
001	01	<table border="1"><tr><td>49</td><td>25</td><td>1</td><td>17</td></tr></table>	49	25	1	17	
49	25	1	17				
010	10	<table border="1"><tr><td>18</td><td>14</td><td>10</td><td>30</td></tr></table>	18	14	10	30	
18	14	10	30				
011	11	<table border="1"><tr><td>31</td><td>35</td><td>11</td><td>7</td></tr></table>	31	35	11	7	
31	35	11	7				
100	00	<table border="1"><tr><td>44</td><td>36</td><td></td><td></td></tr></table>	44	36			
44	36						

Next=1



- Let's say there haven't been any deletions so far. What can you say about the last entry whose insertion into the index caused a split?
- Sketch the index after inserting an entry with hash value 22.
- Now sketch the resulting index after additionally inserting an entry with hash value 15.
- Sketch the original index after deleting the entries with hash values 36 and 44.

**Exercise 3: Extendible vs. Linear Hashing**(4 Points) 

Can you build a Linear Hashing and an Extendible Hashing index with the same data entries, such that the Linear Hashing index has more pages than the Extendible Hashing index and vice versa? If so, give an example. For simplicity, assume that the directory page of the Extendible Hashing index spans just one page, independent of the number of entries.

# Assignment 7

Issue Date: December 16, 2013

Due Date: January 13, 2013, 10:00 a.m.

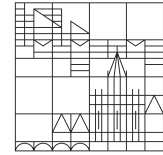
Σ 40+15 Points

Database System Architecture and Implementation

INF-12950

WS 2013/14

Universität  
Konstanz



University of Konstanz  
Database and Information Systems  
Jun.-Prof. Dr. Michael Grossniklaus  
Andreas Weiler, Leo Wörteler

## External Sorting



### General Notes

Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises. Also take note of the general policies that apply to this course as listed on the course website <http://www.informatik.uni-konstanz.de/grossniklaus/education/dbmsinternals/>.

- Please include **[arch13] Group #**, where # is your **group number**, in the subject line of the e-mail with your submission.
- Since this is a programming assignment, you will submit an archive (ZIP file or similar) that only contains the **relevant** source code files and a `README.txt` file (see below).
- The use of external libraries is **not** permitted, except if they are explicitly provided by us.
- Submissions with compile-time errors due to syntax or Checkstyle problems will **immediately** be disqualified. Whereas solutions with run-time errors will be considered, they will not result in a good grade.

### Prerequisites

In order to successfully complete this project, you will need to install the following software on the computer that you use for software development.

- **Java Development Kit**, available at <http://www.oracle.com/technetwork/java/javase/downloads/index.html> (version 6 or greater).
- **Apache Maven**, available at <http://maven.apache.org> (version 3.0.0 or greater).
- **Eclipse IDE for Java Developers**, available at <http://www.eclipse.org/downloads/> (we recommend the Kepler release, i.e., version 4.3.1).
- **Eclipse Checkstyle plug-in**, installation instructions at <http://eclipse-cs.sourceforge.net/downloads.html> (use the latest version).
- **Source Code Repository** (optional), contact us if you would like to use a repository to manage your code.

### Minibase

Minibase is a miniature relational DBMS, originally developed at the University of Wisconsin by Raghu Ramakrishnan to support the practical exercises of the text book that we use in our course. In total, there will be four assignments that are based on Minibase. In all of these assignments, you will implement simplified versions of different layers of a typical DBMS, without support for concurrency control or recovery.

We use an updated version of the code written by Chris Mayfield and Professor Walif Aref of Purdue University as well as Professor Leonard Shapiro of Portland State University. More recently, members of the Database and Information Systems group at the University of Konstanz have refactored and extended the Minibase source code for use in this course. The assignments themselves are partly based on assignments provided by Professor Kristin

Tufte of Portland State University, Christian Grün of the University of Konstanz, Chris Mayfield of Purdue University and Raghu Ramakrishnan of the University of Wisconsin. We thank everybody for the development of these exercises and their work on the Minibase code!

After you have installed and configured the required software (see above), you can set up a local copy of the Minibase source code distribution by following the steps below.

- **Download** the source code distribution archive from the course website and **unzip** it in a directory of your choice.
- Open a **command line** or **terminal** window and change into the directory (`cd`), where you have unzipped the source code. Make sure you are in the **root directory** of the source code distribution, which contains the file `pom.xml`.
- In order to **initialize the Maven project** and **create configuration files** for Eclipse, execute `mvn clean eclipse:clean eclipse:eclipse` from the command line prompt. Note that you need to have an Internet connection the first time you execute this command as Maven will not be able to download plug-ins and libraries otherwise. Should you be unfamiliar with Apache Maven, please refer to the documentation on the website listed above.
- Now you are ready to **launch Eclipse** and **import the Minibase project**. In order to do so, follow the steps of the import project wizard (File → Import... → General → Existing Projects into Workspace). In the wizard select the directory where you have unzipped the source code distribution under **Select root directory** and make sure that the option **Copy projects into workspace** is *not* selected.
- Once Eclipse has finished importing and compiling the project, there should be *no* errors, but there might be a couple of warnings. Make sure that **Checkstyle is activated** for the project by right-clicking on the project and selecting **Checkstyle → Activate Checkstyle**.
- **Congratulations!** If you have made it to this point without errors, you are ready to begin your programming project.

**Disclaimer:** Please note that Minibase is neither open-source, freeware, nor shareware. Refer to the file `COPYRIGHT.txt` in the `doc` folder of the Minibase distribution for terms and conditions.

## **i** Programming Practices

Checkstyle will already check the formatting of the code and enforce a certain coding convention. The Checkstyle configuration that we use is a slightly more relaxed version of the default Java coding convention. Note that you can configure Eclipse to ensure that your code is formatted accordingly as you write it (**Preferences...** → **Java** → **Code Style**). The source code distribution that you downloaded from the course website contains configuration files that can be imported to correctly configure the **Clean Up**, **Code Templates**, **Formatter**, and **Organize Imports** options.

Apart from the rules enforced by Checkstyle your code also needs to adhere to the following (good) programming practices.

- Use **meaningful names** for classes, interfaces, methods, fields, and parameters. For example, use `BufferManager` instead of `BMgr` or `page` instead of `p`.
- **Organize** your code by grouping it into (conceptual) blocks that are separated by a new line.
- Write a (Javadoc) **comment** before each method and paragraph of code.
- Provide a **comment** before each non-obvious declaration.
- Your code must be **understandable** by reading the comments only.
- Comments should not simply paraphrase what the code does, but rather **explain** it. For example, your comment for the line of code `clock = 0;` should not read “Assign 0 to variable *clock*”. A good comment would be “Initialize the clock to point to the first page in the buffer pool”.
- **Efficiency** is considered, e.g., do not use an `int` if a `bool` is appropriate, do not do a sequential search or a disk I/O unless necessary, etc.

## **i** Assignment Overview

You will be given parts of the Minibase code, in Java, and asked to fill in other parts of it. You may spend 90% of your time understanding the given code and 10% of your time writing new code.

Minibase is structured in layers that follow the reference architecture presented in the lecture. Each layer corresponds to a Java package. All of these packages are managed as subdirectories of `src/main/java`, where

directory `src` is directly located in the root directory of the Minibase distribution. For each package of the main Minibase system, there is a corresponding package of the same name that contains JUnit tests, which you will use to check the correctness of your implementation. The JUnit tests are located in subdirectories of `src/test/java`. If you have successfully used Apache Maven to create Eclipse project files, all of these directories will be added as source folders to your project when you import it into Eclipse.

The best place to start understanding this assignment are the lecture slides and the text book (Chapter 13) about external sorting. In this assignment, you will work on the operator evaluator layer, which is part of Minibase's query processor and is located in package `minibase.evaluator`. Your task is to populate this package with a sorting operator for Minibase that implements the simple *two-way merge sort* algorithm from the lecture. The `minibase.evaluator` package currently contains two helper classes, `Schema` and `Tuple`, which you may use at your convenience. A major difference of this assignment compared to previous assignments is that fact that you will start your implementation from scratch, i.e., without skeleton classes and JUnit tests. As a consequence, you will also need to develop the "ecosystem" that supports and validates the sorting operator. This additional effort is balanced in two ways. First, you only need to implement the most simple variant of external sorting to get full credit. Second, you will also get guidance and credit for these tasks. **Hint:** The first two exercises are *completely* independent of each other and can be solved individually to increase productivity!

### Exercise 1: Creating Input Data

(5 Points)



The first step towards building an external sort operator is to create a file that contains the data of the table that will eventually be sorted. You will use JUnit to create a test setup that writes random tuples to a heap file.

- In the `src/test/java` source tree, create a new JUnit test class named `ExternalSortTest` within the `minibase.evaluator` package.
- In this class, create a new method `public final void setUp()` that initializes the Minibase disk and buffer managers. Look at method `TestDriver#createDB()` to see how this can be done. Once the buffer manager has been initialized, you can create a new `HeapFile`. Annotate this method with `@Before`.
- Now you will fill this heap file with random tuples that follow the **Sailors** schema from the lecture.

**Sailors**(*sid*: integer, *sname*: string, *rating*: integer, *age*: float)

Records can be written to a heap file using the method `HeapFile#insertRecord()`, which takes a byte array as input. You can either choose to populate these byte arrays manually or by using the above-mentioned classes `Tuple` and `Schema`. As you can see from the `Convert` class in package `minibase.util`, integers and floats are 32-bit long. For values of type `string`, you will need to choose a length, say 50 characters, and also set it when you initialize the schema. Note that Minibase uses fixed-length strings and therefore the records you will sort will also be fixed-length.

- Values for the *sid* attribute are generated as an incrementing integer value, starting at 0, in order to guarantee that it is a key.
  - Values for the *sname* attribute are generated by concatenating the string "Sailor" with a random integer. You do not need to make sure that there are no duplicates as two sailors can have the same name.
  - Values for the *rating* attributes are generated as random integers in the range  $[0, \dots, 10]$ .
  - Values for the *age* attribute are generated as random floats in the range  $[18.0, \dots, 99.9]$ .
- d) The heap file you generate by inserting records should consist of at least 10 pages. Assuming that you chose 50 as the length of the *sname* attribute, each record will be 62 bytes long. Since Minibase's heap file implementation uses a slot directory that stores a record length (`short`) and a pointer to the record (`short`), each record grows by another 4 bytes. The header of each (data) page in a heap file is 16 bytes. Since the Minibase's page size is set to 1 kB, a page can only store  $\left\lfloor \frac{1024-16}{62+4} \right\rfloor = 15$  records. You will therefore need to create and insert at least 150 records.

Once you have generated the file, it might be a good idea to print its contents to the console using a `HeapScan` in order to check that everything is as it should be. Note that a heap scan can be opened using method `HeapFile#openScan()`.

### Exercise 2: File Format for Runs

(10 Points)



Before you can implement the actual sorting operator, you will need to create an additional file data structure to

save the runs that the two-way merge sort algorithm creates. We will not use the existing heap file implementation for this purpose as it does not manage its pages in a sequence and it does not guarantee that records are physically stored in the order they are inserted.<sup>1</sup> Instead, you will implement a very simple heap file that manages its pages as a linked list of pages as shown in Figure 1. We will refer to this file as a *run file*.

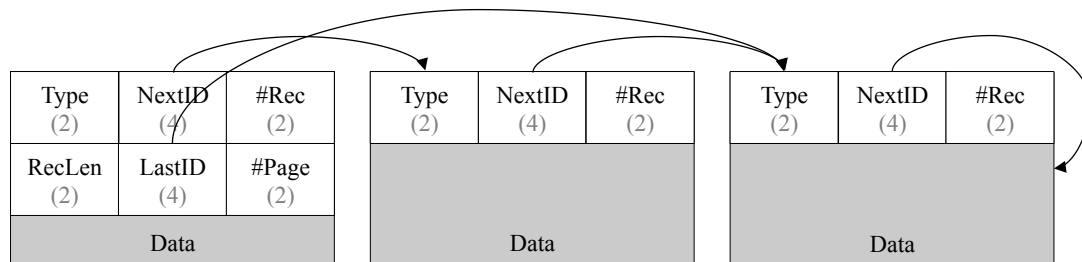


Figure 1: Page Layout of Run File

Since this run file will only be written and read sequentially, it suffices to store a forward pointer (*NextID*) in each page. Additionally, we store the type (*Type*) of the page (i.e., whether or not it contains header information) and the number of records on that page (*#Rec*). The first page, which also contains header information additionally stores the length (*RecLen*) of the (fixed-size) records in the file, a pointer (*LastID*) to the last page that contains data, and the total number of pages in the file (*#Page*). The numbers in grey denote the number of bytes that are required to store this information.

- a) In the `src/main/java` source tree, create three new classes `RunFile`, `RunFilePage`, and `RunFileScan` in the `minibase.access.heap` package. Look at the corresponding classes that implement the existing heap file to get an idea what functionality will go in each of these new classes. **Hint:** Most of the functionality that you need to implemented can be based on or adapted from the general heap file.
- b) Class `RunFile` will provide the following functionality.
  - The constructor of the class accepts a reference to the buffer manager, a name for this file (e.g., `run.nr`), the size of the records in this run, and a number of pages equal to the length of the current run. The constructor then uses the disk manager (accessible through the buffer manager) to look up (`DiskManager#getFileEntry()`) or create a new file entry (`DiskManager#addFileEntry()`). If a new file entry has to be created, the constructor also allocates a sequence of pages by invoking `DiskManager#allocatePages(int)` with the given number of pages. This will return the page ID of the first page, all other pages are guaranteed to be found at increasing page numbers. **Hint:** Regardless of whether the run file is being created or opened, it is a good idea to *cache* header information from the first page in global variables. The same is true for the page ID of the first page (header page) of the run file.
  - The most important method of this class is `appendRecord(byte[] record)` that appends the given record to the file. Assuming we have cached the ID of the last page that contains data, we can directly pin that page. By subtracting  $\#Rec \times RecLen$  (plus the size of the header) from the page size, we can check if there is space on the page left to place the record. If so, we append the record to the page (see below). If not, we increase the current page ID<sup>2</sup>, write it to *NextID* and unpin the current page (dirty). We then pin the page with page ID *NextID*, initialize it, and append the record (see below). Finally, we unpin the new page (dirty), pin the header page, and update the *LastID* information (as well as our cache).
  - In order to delete a file, this class also implements a method `deleteFile()`, which frees all pages of the file. Since pages have been allocated in a sequence of increasing page IDs, we can simply iterate over all pages, starting at the header page.
  - Finally, the class provides a method `openScan()`, which returns an instance of `RunFileScan` that can be used to iterate over all records in the file.
- c) Class `RunFilePage` implements the tagging interface `PageType` and provides a number of static methods that interpret the layout of run file pages as shown in Figure 1.
  - You will need to implement getter (and in some cases setter) methods for fields stored in the header of the page. For the fields *Type*, *NextID*, and *#Rec*, you can directly read and write the corresponding val-

<sup>1</sup>Since records are never deleted from runs, the lack of this guarantee would not adversely affect the correctness of the external sort.

<sup>2</sup>For this step, we need to make sure that we do not exceed the total number of pre-allocated pages, using *#Page*.

ues at the correct offset using appropriate methods from class `Page`, for example `Page#writeShort` to update the number of records. For the fields `RecLen`, `LastID`, and `#Page`, you need to check the value of `Type` to make sure that you are dealing with the header page. Otherwise, you risk reading nonsensical bits and pieces of the first record on that (non-header) page.

- The next method you implement is `appendRecord()`, which appends a record to a page. To do so, it needs to compute the correct offset as the size of the header plus  $\#Rec \times RecLen$ . You can then use `System#arrayCopy()` to write the record to the page. Finally, you need to update the `#Rec` information in the header.
- The last method you need to write is `getRecord(int n)`, which reads and returns the  $n$ -th record as a byte array.

d) Class `RunFileScan` provides the following functionality.

- The constructor of the class is initialized with the run file over which the scan iterates. It pins the first page, reads header information into global variables, and unpins the first page again.<sup>3</sup> It also initializes pointers to the currently pinned page (`Page< ? >`) and the last record (`int`) that was returned.
- Method `hasNext()` returns `true` if there are more records in the file and `false`, otherwise. There are more records in the file, if no page has been pinned yet, i.e., we are at the beginning of the file, or if the currently last returned record is not the last record on the last page, i.e., we are not yet at the end of the file. The method throws an exception, if it is called after the scan has been closed.
- Method `next()` returns the next record in the file. If no page is pinned, it pins the first page of the file, returns the first record and updates the pointer to the last returned record. Otherwise, it returns the next record on the currently pinned page. If there is no such record left, it unpins the current page, pins the next page, and returns the first record on that page. The method throws an exception if it is called in a state where `hasNext()` is `false` or if the scan has already been closed.
- Finally, the `close()` method unpins any pinned pages and resets all internal state, e.g., the pointer to the last returned record.

Again, it is recommended that you write a couple of records to your run file and then print them to the console using the your run file scan.

### Exercise 3: Pass 0: Sort Phase

(10 Points)



This exercise combines and builds on the previous two exercises. The goal is to implement the first pass of the simple two-way merge sort algorithm that uses one buffer page to create  $N$  sorted runs of one page each, where  $N$  is the number of pages in the file that is sorted. **Hint:** At the end of this assignment, there is a bonus exercise worth 5 points, which consists of implementing this step using *replacement sort* as presented in the lecture. If you plan to score these bonus points anyway, you can also take the short-cut of bypassing this simple implementation of the first phase!

- Create a new class `ExternalSort` in package `minibase.evaluator` in the `src/main/java` source tree.
- The constructor of this class is initialized with references to the buffer pool and to the heap file to be sorted. Additionally, the external sort operator needs to know the *sort key* and *sort ordering* to be applied. We represent this information as follows. We define a utility class named `SortKey`, which represents the sort key as an ordered subset of record fields, for example using an array. Based on this utility class, we create the interface `SortKeyExtractor` with only one method `SortKey extract(byte[] record)` that extracts a sort key from records. The sort ordering is then simply represented by an instance of `Comparator<SortKey>`, which leads to the following constructor.

```
public ExternalSort(BufferManager bufferManager, HeapFile heapFile,
    SortKeyExtractor extractor, Comparator<SortKey> comparator)
```

- Having initialized the class, you can now begin to implement Pass 0 of the simple two-way merge sort. To do so, you will create a method called `sort()`, which iterates through the heap file page by page. Unfortunately, this functionality is not yet provided by the heap and therefore, the `sort()` method needs to work directly on the pages of the heap file.
  - We begin by pinning the first directory page, which can be obtained by invoking `HeapFile#getHeadID()`.

<sup>3</sup>Note that we can aggressively cache information in global variables because Minibase does not support concurrent transactions.

- ii) For each directory page, we iterate over all data pages. The number of data pages can be read using `HeapFileDirectoryPage#getEntryCount()` and individual data pages can be accessed by retrieving their ID through `HeapFileDirectoryPage#getPageID()`.
- iii) For each data page, we read all records it contains using methods `getFirstRecord()`, `getNextRecord()`, and `selectRecord()` in class `HeapFilePage`.
- iv) For each record, we extract the search key using the search key extractor. We save that search key together with the ID of the record that is currently being processed in an in-memory array. To do so, you will need to create a helper class named `SortEntry`, which defines a field `SortKey key` and a field `RecordID rid` as well as corresponding getter methods.
- v) As soon as we have read all records on a data page, we use `Arrays#sort(T[] a, Comparator<? super T> c)` to sort the in-memory array of type `SortEntry[]`. Note that we cannot directly use the comparator passed to the constructor of the `ExternalSort` class, but we can use an internal class to wrap it into a comparator of type `Comparator<SortEntry>`. The described approach has two advantages. First, by using `Arrays#sort` you do not need to implement your custom in-memory sort algorithm. Second, by extracting `SortEntry` objects from the records on the heap file page, we can avoid sorting the entire records using `System#arrayCopy()` with corresponding offsets, which is likely to be error-prone.
- vi) Now, we create a new one-page run file, named as indicated in the pseudo-code from the lecture, and pin its first and only page. We then append the records from the heap file page one after another in the desired sort order. To do so, we iterate over the sorted sort keys and use the `rids` to read the corresponding records `HeapFilePage#selectRecord()`.
- vii) Once all records have been sorted, we unpin the heap file page and the run file page. We then use the directory page to get the page ID of the next data page, which we pin. If the directory page has been exhausted, we move to the next directory page using `HeapFilePage#getNextPage()` before getting the page ID of the next data page.
- viii) We then repeat steps iii) through vii) until we reach the end of the last directory page.

After invoking `run()`, there should be  $N$  files named `run_0..r` with  $0 < r \leq N$ , where  $N$  is the number of data pages in the heap file. At this point you should use your implementation of `RunFileScan` to examine some of the files that are generated to ensure that they are sorted according to the specified sort key and sort ordering.

#### Exercise 4: Pass $1, \dots, \lceil \log_2 N \rceil$ : Merge Phase

(10 Points)



After having sorted each individual page of the heap file into a one-page run file, the merge phase iteratively combines these runs until only one run remains. To implement this functionality, you will create a method called `merge()` that performs the following steps, according to the pseudo-code of the lecture. **Hint:** There are two bonus exercises worth 5 points each at the end of this assignment, which consist of implementing a multi-way merge and blocked I/O. If you would like to implement these refinements anyway, you can also skip the implementation of the simple two-way merge sort and directly implement the more realistic external merge sort as presented in the lecture.

- a) Since we know how many runs were created in the sort phase, say  $N$ , we know that the merge phase requires  $n = 1, \dots, \lceil \log_2 N \rceil$  passes.
- b) In each pass, we iterate over pairs of run files from the previous pass. We open these files and read them in page by page, by pinning and unpinning one page after another.
- c) For each pair of pinned pages, we iterate over the records of each page. We use the search key extractor together with the comparator to compare records and merge them into a new run.
- d) This new run is written to a run file page by page. The output file for the run is created whenever we open a pair of existing runs and named `run_n.r`, following the naming scheme of the pseudo-code of the lecture, where  $n$  is the pass, i.e.,  $1 \leq n \leq \lceil \log_2 N \rceil$  and  $r$  is the  $r$ -th run of pass  $n$ . The number of pages in this file will be the sum of the number of pages in the input files, which can be obtained by using `RunFilePage` to read `#Page` from the header page, i.e., first page, of each input file.
- e) Once we reach the end of both run files, we delete these files, and open the next pair of runs. If there are no more runs left, we start the next pass by incrementing  $n$ . If  $n > \lceil \log_2 N \rceil$ , we have completed the last pass and can stop the merge phase.

The result of the final run should be stored in a file called `run_⌈log2 N⌉_0`. Once again, use your implementation of `RunFileScan` to verify that this file is indeed sorted according to the specified sort key and sort ordering. If this is not the case, it might be worthwhile to include such a verification step after each pass of the merge phase.

### Exercise 5: Verifying Output Data

(5 Points)

So far, the correctness of the external sort operator has only been verified “manually”, i.e., by printing out the contents of the heap and run files to check whether they match the expected results. In this last exercise, you will implement the following JUnit tests that use the same sort key extractor and comparator as the external operator to automatically verify whether the various files that your algorithm produces are correct.

- Create a JUnit test that verifies the intermediate results after Pass 0 has been executed.
- Create a JUnit test that verifies the final result.

Using the random test data that you have created in Exercise 1 according to the **Sailors** schema, you should try to test your implementation as comprehensively as possible.

- Test that your algorithm works correctly on both unsorted sort keys and sort keys that are already sorted. For the latter, recall that we generate values for attribute *sid* in ascending order.
- Test that your algorithm produces the correct result regardless of the sort ordering. You can test this by supplying a comparator that implements ascending order and one that implements descending order.
- Test that your algorithm works correctly on all field types. Since the **Sailors** schema has fields of all types that are currently supported by Minibase, you can test this by sorting once on each field.
- Test that your algorithm sorts record correctly even if a multi-field sort key is specified. Again, it is interesting to include the *sid* column in this key as it is already sorted.

### Exercise 6: Bonus Round

(15 Points)



For all of the bonus exercises, you will need to extend the constructor of the `ExternalSort` class with a parameter  $B$  that denotes the total number of buffer pages that the sorting algorithm can use. In the case of the last bonus exercise, you will additionally need to pass the number of  $b$  blocks per input to the constructor.

**Replacement Sort** Instead of implementing Exercise 3 as described above, you can also directly implement the replacement sort approach described in the lecture. Since we are only dealing with fixed-length records, this is not as challenging as in the general case, where variable-length records might occur.

**External Merge Sort** Instead of merging two runs from a previous pass as described above in Exercise 4, you can also directly implement the approach where  $B - 1$  runs are merged in each pass. To get the five bonus points, you do not need to implement a selection tree to choose the next record to be output. However, if you do, you will of course be awarded additional bonus points. Note, however, that you need to map the selection tree structure to the  $B - 1$  buffer page as your implementation would otherwise use twice as much memory!

**Blocked I/O** Instead of reading input files one page at a time, you can also read  $b$  pages to profit from sequential disk access. As discussed in the lecture, this refinement increases the number of disk I/O operations, but decreases the average cost of a disk I/O operation.

### Submission

Solutions are submitted electronically by e-mail. Your submission must consist of a single zipped archive that contains the following Java files, located in the appropriate directory, i.e., package structure.

- `RunFile`, `RunFilePage`, and `RunFileScan`
- `SortKey`, `SortKeyExtractor`, and `SortEntry` (possibly an internal class)
- `ExternalSort`
- `ExternalSortTest` and any implementations of `SortKeyExtractor` and `Comparator<SortKey>` required to run the tests.

The name of the file containing your submission should be `grpXX-asgYY.zip`, where `XX` and `YY` are your group and the assignment number, respectively. You will also include a file called `README.txt` that should contain the following information.

**Overall Status** A one paragraph overview of the status of your implementation. If you were unable to finish the project, please give details about what is and is not complete. Be short and to the point!

**Refinements** A one paragraph summary detailing the additional refinements (see above) that you have implemented in your external sort operator.

**Time Spent** Please include how many hours you spent on this project. Note that the time spent has no impact on your grade. This information will only be used in planning future projects.



# Assignment 8

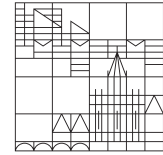
Issue Date: January 13, 2014

Due Date: January 27, 2014, 10:00 a.m.

Σ 40 Points

**Database System Architecture and Implementation**  
**INF-12950**  
**WS 2013/14**

Universität  
Konstanz



University of Konstanz  
Database and Information Systems  
Jun.-Prof. Dr. Michael Grossniklaus  
Andreas Weiler, Leo Wörteler

## Relational Operators



### General Notes

Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises. Also take note of the general policies that apply to this course as listed on the course website <http://www.informatik.uni-konstanz.de/grossniklaus/education/dbmsinternals/>.

- Please include **[arch13] Group #**, where # is your **group number**, in the subject line of the e-mail with your submission.
- Since this is a programming assignment, you will submit an archive (ZIP file or similar) that only contains the **relevant** source code files and a `README.txt` file (see below).
- The use of external libraries is **not** permitted, except if they are explicitly provided by us.
- Submissions with compile-time errors due to syntax or Checkstyle problems will **immediately** be disqualified. Whereas solutions with run-time errors will be considered, they will not result in a good grade.

### Prerequisites

In order to successfully complete this project, you will need to install the following software on the computer that you use for software development.

- **Java Development Kit**, available at <http://www.oracle.com/technetwork/java/javase/downloads/index.html> (version 6 or greater).
- **Apache Maven**, available at <http://maven.apache.org> (version 3.0.0 or greater).
- **Eclipse IDE for Java Developers**, available at <http://www.eclipse.org/downloads/> (we recommend the Kepler release, i.e., version 4.3.1).
- **Eclipse Checkstyle plug-in**, installation instructions at <http://eclipse-cs.sourceforge.net/downloads.html> (use the latest version).
- **Source Code Repository** (optional), contact us if you would like to use a repository to manage your code.

### Minibase

Minibase is a miniature relational DBMS, originally developed at the University of Wisconsin by Raghu Ramakrishnan to support the practical exercises of the text book that we use in our course. In total, there will be four assignments that are based on Minibase. In all of these assignments, you will implement simplified versions of different layers of a typical DBMS, without support for concurrency control or recovery.

We use an updated version of the code written by Chris Mayfield and Professor Walif Aref of Purdue University as well as Professor Leonard Shapiro of Portland State University. More recently, members of the Database and Information Systems group at the University of Konstanz have refactored and extended the Minibase source code for use in this course. The assignments themselves are partly based on assignments provided by Professor Kristin

Tufte of Portland State University, Christian Grün of the University of Konstanz, Chris Mayfield of Purdue University and Raghu Ramakrishnan of the University of Wisconsin. We thank everybody for the development of these exercises and their work on the Minibase code!

After you have installed and configured the required software (see above), you can set up a local copy of the Minibase source code distribution by following the steps below.

- **Download** the source code distribution archive from the course website and **unzip** it in a directory of your choice.
- Open a **command line** or **terminal** window and change into the directory (`cd`), where you have unzipped the source code. Make sure you are in the **root directory** of the source code distribution, which contains the file `pom.xml`.
- In order to **initialize the Maven project** and **create configuration files** for Eclipse, execute `mvn clean eclipse:clean eclipse:eclipse` from the command line prompt. Note that you need to have an Internet connection the first time you execute this command as Maven will not be able to download plug-ins and libraries otherwise. Should you be unfamiliar with Apache Maven, please refer to the documentation on the website listed above.
- Now you are ready to **launch Eclipse** and **import the Minibase project**. In order to do so, follow the steps of the import project wizard (File → Import... → General → Existing Projects into Workspace). In the wizard select the directory where you have unzipped the source code distribution under **Select root directory** and make sure that the option **Copy projects into workspace** is *not* selected.
- Once Eclipse has finished importing and compiling the project, there should be *no* errors, but there might be a couple of warnings. Make sure that **Checkstyle is activated** for the project by right-clicking on the project and selecting **Checkstyle → Activate Checkstyle**.
- **Congratulations!** If you have made it to this point without errors, you are ready to begin your programming project.

**Disclaimer:** Please note that Minibase is neither open-source, freeware, nor shareware. Refer to the file `COPYRIGHT.txt` in the `doc` folder of the Minibase distribution for terms and conditions.

## **i** Programming Practices

Checkstyle will already check the formatting of the code and enforce a certain coding convention. The Checkstyle configuration that we use is a slightly more relaxed version of the default Java coding convention. Note that you can configure Eclipse to ensure that your code is formatted accordingly as you write it (**Preferences...** → **Java** → **Code Style**). The source code distribution that you downloaded from the course website contains configuration files that can be imported to correctly configure the **Clean Up**, **Code Templates**, **Formatter**, and **Organize Imports** options.

Apart from the rules enforced by Checkstyle your code also needs to adhere to the following (good) programming practices.

- Use **meaningful names** for classes, interfaces, methods, fields, and parameters. For example, use `BufferManager` instead of `BMgr` or `page` instead of `p`.
- **Organize** your code by grouping it into (conceptual) blocks that are separated by a new line.
- Write a (Javadoc) **comment** before each method and paragraph of code.
- Provide a **comment** before each non-obvious declaration.
- Your code must be **understandable** by reading the comments only.
- Comments should not simply paraphrase what the code does, but rather **explain** it. For example, your comment for the line of code `clock = 0;` should not read “Assign 0 to variable *clock*”. A good comment would be “Initialize the clock to point to the first page in the buffer pool”.
- **Efficiency** is considered, e.g., do not use an `int` if a `bool` is appropriate, do not do a sequential search or a disk I/O unless necessary, etc.

## **i** Assignment Overview

You will be given parts of the Minibase code, in Java, and asked to fill in other parts of it. You may spend 90% of your time understanding the given code and 10% of your time writing new code.

Minibase is structured in layers that follow the reference architecture presented in the lecture. Each layer corresponds to a Java package. All of these packages are managed as subdirectories of `src/main/java`, where

directory `src` is directly located in the root directory of the Minibase distribution. For each package of the main Minibase system, there is a corresponding package of the same name that contains JUnit tests, which you will use to check the correctness of your implementation. The JUnit tests are located in subdirectories of `src/test/java`. If you have successfully used Apache Maven to create Eclipse project files, all of these directories will be added as source folders to your project when you import it into Eclipse.

The best place to start understanding this assignment are the lecture slides and the text book (Chapter 14) about the evaluation of relational operators. As in the previous assignment, you will work on the operator evaluator layer, which is part of Minibase's query processor and is located in package `minibase.evaluator`. Your task is to create a join operator for Minibase according to the Volcano iterator model. Your join operator will implement *one* of the following algorithms: index nested loops join, sort-merge join, or hash join. As in the previous exercise, you will also design test cases that thoroughly validate the correctness of your implementation.

### Exercise 1: Join Operator

(30 Points)



The first exercise of this assignment is to implement a join operator for Minibase. To do so, you will need to extend the abstract class `AbstractJoin`, which is a subclass of `AbstractOperator` that implements the `Iterator` interface. For a simple example of a join operator, you can have a look at class `NestedLoopsJoin`, which is part of the source code provided for this exercise. As you can see, a join operator is constructed using two input iterators over the outer and inner relation, respectively. If you want to directly input relations into your join operator, you can use class `TableScan`, which wraps a heap file to provide the required iterator interface. Additionally, a join operator is initialized with a (conjunctive) list of simple predicates. To keep things simple, however, we will limit ourselves to join predicates that consist of a *single* equality or inequality. Your join operator will implement *one* of the following join algorithms that were presented in the lecture, sorted in ascending order of difficulty.

**Index Nested Loops Join** For every tuple of the outer relation, the *index nested loops join* probes an index over the inner relation. While the algorithm for this join variant is easy to implement, you will have to build an index over the inner relation. Of course, you can reuse the  $B^+$  tree implementation from Assignment 5 that is included in the source code provided for this assignment. Since the inner relation is represented by an index, you will not be able to implement this join variant as a subclass of `AbstractJoin` as its constructors expects two iterators. Although an index scan or lookup can also be implemented as an iterator, the current iterator interface of Minibase does not support this approach.

**Sort-Merge Join** The *sort-merge join* sorts both relations in a first phase and then merges them into the result relation in the second phase. If you chose to implement this join variant, you can build on the external sort operator from Assignment 6 to sort (and materialize) the two input relations. The merge phase can then be implemented as discussed in the lecture. Note that instead of simply reusing the external sort operator, you can also completely integrate your join implementation with the external sort operator as described in the lecture.

**Hash Join** The hash join uses a hash function to split both relations into  $k$  partitions during the build phase. In the probe phase, it loads the outer relation into memory partition by partition, builds a hash table over each partition using a second hash function, and probes that hash table with tuples from the inner relation. Clearly, this join variant is the most challenging of the three as there is nothing, *absolutely nothing* that can be reused. There are several challenges that you will need to address. First, you have to define two hash functions that are as immune to skew in the data as possible. Second, you will have to figure out how to represent a hash table in memory using Minibase pages, i.e., byte-arrays. Finally, there is the possibility that the partitions created by your hash function do not fit into the available buffer pages. At this point, your implementation should recursively repartition the input relations. In the lecture, we have conveniently omitted this step!

Depending on the algorithm that you choose to implement, create a Java class that is named accordingly, e.g., `IndexNestedLoopsJoin`, `SortMergeJoin`, or `HashJoin`.

### Exercise 2: Tests

(10 Points)



As in the previous exercise, you will implement a JUnit test suite that validates the correctness of your implementation. To do so, you will complete the classes `EvaluatorBaseTest` and `JoinTest`. `EvaluatorBaseTest` is a common base class for all test cases in the `minibase.evaluator` package. It defines the schema for relations **Boats**, **Reserves**, and **Sailors**, as defined in the text book and used in the course. The class already contains functionality to generate sample data for the **Sailors** relation (from the previous exercise). Your task is to write a piece of code that generates similar data for the schema of **Reserves** relation as shown below.

**Reserves**(*sid*: integer, *bid*: integer, *day*: integer, *rname*: string)

Records can be written to a heap file using the method `HeapFile#insertRecord()`, which takes a byte array as input. You can either choose to populate these byte arrays manually or by using the above-mentioned classes `Tuple` and `Schema`. As you can see from the `Convert` class in package `minibase.util`, integers and floats are 32-bit long. For values of type `string`, you will need to choose a length, say 50 characters, and also set it when you initialize the schema. Note that Minibase uses fixed-length strings.

- Values for the *sid* attribute are generated as a random integer value between 0 and `MAX_TUPLES`, i.e., the number of **Sailors** tuples.
- Values for the *bid* attribute are generated as an incrementing integer value, starting at 0. Since we are not populating the **Boats** relation, it does not matter what values are created for *bid*.
- Values for the *day* attribute are generated as a random integer value. Note that Minibase does not provide a type to support dates and, therefore, we cannot generate realistic data for this attribute.
- Values for the *rname* attribute are generated selecting a random name from the `SNAMES` array.

Based on this random test data for the **Sailors** and **Reserves** schema, you should try to test your implementation as comprehensively as possible by creating test cases for the following scenarios in class `JoinTest`.

- Test that your algorithm works correctly if you join the two relations on the *sid* attribute. First, make sure that equality predicates work. If you have implemented the index nested loops join for a  $B^+$  tree, you should also verify that inequality predicates work.
- Test that your algorithm works correctly, independently of which table is the inner and the outer table.
- Implement additional data generators that enable you to test fringe cases. For example, test that your join will compute the cross-product if all tuples match and the empty set if no tuples match.
- For implementations of the sort-merge join or hash join, test that your operator also joins the two relations correctly if the predicate **Sailors**.*sname* = **Reserves**.*rname* is used.
- Experiment with join predicates over key attribute versus join predicates over non-key attributes. This is particularly important in the case of the sort-merge join.

## **i Submission**

Solutions are submitted electronically by e-mail. Your submission must consist of a single zipped archive that contains the following Java files, located in the appropriate directory, i.e., package structure.

- `IndexNestedLoopsJoin`, `SortMergeJoin`, or `HashJoin`
- `EvaluatorBaseTest` and `JoinTest`.

The name of the file containing your submission should be `grpXX-asgYY.zip`, where `XX` and `YY` are your group and the assignment number, respectively. You will also include a file called `README.txt` that should contain the following information.

**Overall Status** A one paragraph overview of the status of your implementation. If you were unable to finish the project, please give details about what is and is not complete. Be short and to the point!

**Bonus Points** A long list of reasons explaining why or for what you should be awarded bonus points, even though there has not been a single bonus exercises in this assignment.

**Time Spent** Please include how many hours you spent on this project. Note that the time spent has no impact on your grade. This information will only be used in planning future projects.

# Assignment 9

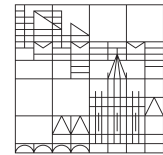
Issue Date: January 27, 2013

Due Date: February 03, 2013, 10:00 a.m.

Σ 20 Points

**Database System Architecture and Implementation**  
**INF-12950**  
**WS 2013/14**

Universität  
Konstanz



University of Konstanz  
Database and Information Systems  
Jun.-Prof. Dr. Michael Grossniklaus  
Andreas Weiler, Leo Wörteler

## Cost and Estimations

### **i** General Notes

Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises. Also take note of the general policies that apply to this course as listed on the course website <http://www.informatik.uni-konstanz.de/grossniklaus/education/dbmsinternals/>.

- From now on, please include [arch13] and your **group number** in the e-mail subject.
- Write your **names** and **group number** in all files that you submit.
- Solutions must be submitted **before the deadline** published on the website and the assignment.

### Exercise 1: Cost Estimations

(4 Points)

To improve cost estimations, cost indicators can be attached to the operators in a query plan.

- Can a single operator type (*Selection*, *Projection*, ...) have different costs?
- Name two factors that might influence the costs of an operator.

### Exercise 2: Cost Calculations

(5 Points)

The following costs and selectivities are given:

Operator	Selectivity	Costs
01	$S1 = 0.1$	100
02	$S2 = 0.2$	20
03	$S3 = 0.8$	10

Choose and name one of the three plans given in the lecture slides (*DNF*, *CNF*, *Bypass*), and calculate the costs of the two following operations: a)  $01 \vee (02 \wedge 03)$  b)  $01 \wedge (02 \vee 03)$

### Exercise 3: Join Conditions

(5 Points)

Which of the following join operators can be applied to non-equi joins ( $\neq$ ,  $<$ ,  $>$ , ...)? Please guess/explain why.

- a) *Sort-merge*   b) *Hash*   c) *Index-nested loop*   d) *Block-nested*

### Exercise 4: New Operators

(4 Points)

How would you implement a new aggregate operator called *SECOND\_LARGEST*, which is a variation of the *MAX* operator? You can either describe your idea, or write down pseudocode.

### Exercise 5: Histograms

(2 Points)

The histogram on the right shows the (approximate!) data distribution of a specific table attribute (few entries with value 0 exist, most entries have high values). Please formulate a sample query that might be sped up by the given histogram.

