Assignment 5

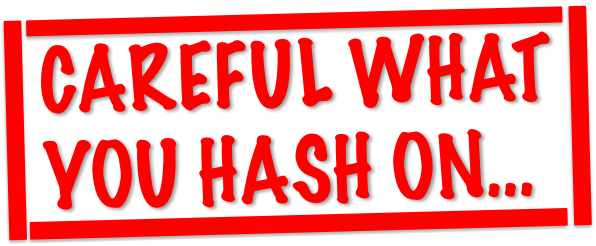
Issue Date: December 4, 2018
Due Date: December 17, 2018, 10:00 A.M.
$\sum$ 40 Points

Database System Architecture and Implementation
INF-20210
WS 2018/19

Universität Konstanz

University of Konstanz
Database and Information Systems
Prof. Dr. Michael Grossniklaus
Leonard Wörteler

Disk-based Hash Indexes



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/inf-20210/`.

- Submit your solutions through Ilias **before the deadline** published on the website and the assignment.
- 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 Section Submission 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.

i 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 `https://www.oracle.com/technetwork/java/javase/downloads/index.html` (version 8 or greater).
- **Apache Maven**, available at `http://maven.apache.org/` (version 3.5.2 or greater).
- **Eclipse IDE for Java Developers**, available at `http://www.eclipse.org/downloads/`.
- **Eclipse Checkstyle plug-in**, installation instructions at `https://checkstyle.org/eclipse-cs/` (version 8.12 or greater).

i 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.
- **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. See `COPYRIGHT.txt` in the `doc` folder of the Minibase distribution for terms and conditions. Also do not push the code to publicly accessible repositories, e.g. GitHub.

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.
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 line break.
- 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 `boolean` 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 course slides (Module 4) and the text book (Sections 11.1, 11.2, and 11.3) about different types of hash-based indexing. Following the reference architecture presented in the course, Minibase is structured in layers. Each layer corresponds to a Java package. The Minibase indexes are part of the `minibase.access` 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 your Eclipse project is configured and imported correctly, all of these directories should be used as source folders.

In this assignment, you will be given the access layer (`minibase.access`), which contains the implementations of all access paths supported by Minibase. The provided code consists of a fully-functional *static* hash index as well as skeleton classes for the extendible and linear hash index. Before you start implementing you should study this code and make sure that you have an in-depth understanding of how it works. Your task will be to implement a second hash-based index that either uses extendible **or** linear hashing.

Exercise 1: Static Hash Index (10 Points) ✍

The existing static hash index is located in package `minibase.access.hash` and builds on the general index functionality contained in package `minibase.access.index`. Below is a summarized overview of these packages and the classes they provide.

- `minibase.access.index`
  - `Index`: General interface for all indexes provided by Minibase. This interface defines *search*, *insert*, and *remove* methods as well as methods to open scans and to delete the index.
  - `IndexScan`: Marker interface for index scans, which combines the functionality of `Iterator` and `Closeable`. This interface needs to be implemented by all index scans as it is returned by `Index` whenever a scan is opened.
  - `DataEntry`: An *index entry* (!) of the form (k, rid), *i.e.*, Variant 2 from the course and the text book.
- `minibase.access.hash`
  - `HashIndex`: Common interface for all hash indexes (currently, being redesigned).
  - `StaticHashIndex`: Fully-functional implementation of a static hash index as described in Section 11.1 of the text book.
  - `ExtendibleHashIndex`: Skeleton class for an extendible hash index as described in Section 11.2 of the text book. The fields of the class, its constructor, and some utility methods are given.
  - `LinearHashIndex`: Skeleton class for a linear hash index as described in Section 11.3 of the text book. The fields of the class, its constructor, and some utility methods are given.
  - `HashDirectoryHeader`: Page type used to manage the first page of the hash directory of a hash index. Apart from the header information, a page of this type also contains hash directory entries in its data section. As the directory grows, additional hash directory entries are stored using pages of type `HashDirectoryPage` (see below).
  - `HashDirectoryPage`: Page type used for additional directory pages of a hash index. Note that the hash buckets containing the data are represented by `BucketPage` (see below).
  - `HashDirectoryIterator`: Iterator over the entries in the hash directory. Directory entries are represented a the id of the first page of the corresponding hash bucket.
  - `BucketPage`: Page type used to store a hash bucket containing the entries of the hash index. Bucket pages can be linked to form overflow chains.
  - `BucketChainIterator`: Iterator over all index entries of a hash bucket, which is represented as a (list of) bucket page(s).
  - `HashIndexScan`: Scan that iterates over all index entries contained in the index by looping over all index buckets.
  - `KeyScan`: Scan that iterates over all index entries that have the same `SearchKey` as the one given to the constructor of this class.

As an introduction to the actual programming exercise, study this code and answer the following questions.

a) Since the static hash index cannot adapt its hash directory as index entries are inserted and deleted, it has to rely on overflow chains. Examine the methods that implement the insert and remove operations.
   i) Draw a diagram of how an overflow chain is represented in terms of bucket pages.
   ii) On which bucket page are new entries inserted?
   iii) How is an entry removed depending on whether it is located on the first or any other page?
   iv) State the invariant that holds for a bucket page chain due to this insertion and removal algorithm.
   v) Describe the complexity of search, insert, and remove functionality.

b) Outline an alternative design to manage chains of bucket pages. How does your design compare to the implemented one? Give three advantages or disadvantages.

c) Can you imagine why it makes sense to implement a scan that returns all index entries of an index, *e.g.*, `HashIndexScan`?

Exercise 2: Extendible *or* Linear Hash Index (20 Points)

Choose *one* of the following two programming exercises, or both for bonus points.

a) **Extendible Hash Index**: In package `minibase.access.index`, you will find a skeleton class `ExtendibleHashIndex` that implements the `HashIndex` interface described above. In order to implement the extendible hashing algorithm described on the course slides or in the text book (Section 11.2), you need to manage directory page and hash bucket pages.
Your implementation should reuse the existing infrastructure, *i.e.*, classes `HashDirectoryHeader` and `HashDirectoryPage`. In contrast to the static hash index, the extendible hash index needs to manage the global depth of the directory and local depths of the hash buckets. The global depth should be stored on the header page of the hash directory, whereas the local depths of the hash buckets should be stored together with the directory entries pointing to the buckets. Since this changes the layout of directory entries, you will need to write a new directory iterator, say `ExtendibleHashDirectoryIterator`, for the extendible hash index.
You may have noticed that the static hash index does not allocate empty bucket pages (the same optimization is also possible in the case of the linear hash index). In the case of the extendible hash index, however, we recommend that you do allocate empty buckets as it is otherwise difficult to efficiently determine which buckets need to be created once a split occurs.

b) **Linear Hash Index**: In package `minibase.access.index`, you will find a skeleton class `LinearHashIndex` that implements the `HashIndex` interface described above. In order to implement the linear hashing algorithm described on the course slides and in the text book (Section 11.3), you need to manage directory and bucket pages.
Your implementation should reuse the existing infrastructure, *i.e.*, classes `HashDirectoryHeader` and `HashDirectoryPage`. In contrast to the extendible hash index that doubles or halves the size of the directory to grow or shrink it, the directory of the linear hash index grows and shrinks one entry at a time. Therefore, you need to manage a pointer to the next bucket that will be split or merged. As you can see in the constructor, the skeleton that we provide does not store this pointer directly. Rather, it computes it from the current level and number of buckets.
Recall that linear hashing requires a family of hash functions $h_0, h_1, \ldots$ and selects the appropriate function according to the current level. This functionality is already provided by the method `SearchKey#getHash(int)` that accepts the level (named `depth` in the code) as an input parameter.
Finally, the split and merge criterion that you will use in your implementation is different from the one used in the examples of the lecture. Rather than using the allocation and deallocation of overflow pages to trigger splits and merges, your implementation should split and merge based on the current load factor of the index, which is computed as

$$\frac{|\text{Entry}|}{|\text{Bucket}| \times \textit{sizeOf}(\texttt{BucketPage})/\textit{sizeOf}(\text{Entry})}.$$

If this load factor increases above the threshold set for the maximum load factor, a split should occur. Analogously, if it drops below the threshold for the minimum load factor, a merge should occur.

Regardless of which index you choose to implement, you should also think about how you can deal with highly skewed data. For example, what happens if all values inserted are multiples of 256 (`0b100000000`) and, therefore, do not differ in the last eight bits?

Exercise 3: Testing (10 Points)

For your convenience, we provide a set of general JUnit tests that you can use to check whether your implementation works. In this last exercise, you will implement additional JUnit tests to demonstrate the correctness of your implementation under special circumstances and to compare its performance to the existing static hash index.

a) Look through the code of class `IndexTest` and study the different test cases that are provided.

b) Method `IndexTest#fuzzyTest()` performs a configurable number of insert, lookup, and deletion operations on the index. However, it uses a random number generator to create the integer keys are worked with. Add another test case that performs the same number of operations, but uses skewed data. How is the performance of your index impacted by this change?

c) Another assumption made by `IndexTest#fuzzyTest()` is that insert, lookup, and deletion operations have the same frequency. Add another test case that performs the same number of operations, but can be configured to simulate different work loads, *e.g.*, "read-only", "read-mostly", "read-write", "no-delete", *etc.* How is the performance of your index impacted by this change?

i Submission

Solutions are submitted electronically by Ilias. 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` (or end with the corresponding suffix of the compression program you used), where # is your group and the assignment number, respectively. You will also include a file called `README.txt` that should contain the following information.

Answers to Written Questions Include your answers to questions 1a) and 1b).

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 For all files you have created, list their names and a short description.

Benchmark Results Summarize the run-time results you have measured with your benchmarking tests. Do not forget to include a description how Minibase was configured (size of buffer pool, page size, etc.) and what computer (CPU type, amount of RAM, type and size of hard disk) you used.

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.

Are you still reading or already programming?