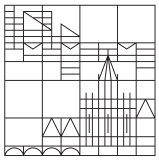
Assignment 8

Issue Date: January 15, 2018
Due Date: January 28, 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

Relational Operators

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.

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`.
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.query.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 `AbstractOperator` that implements the `Operator` interface. For a very simple example of a join operator, you can have a look at class `NestedLoopsEquiJoin`, which is part of the source code provided for this exercise. As you can see, a join operator is constructed using two input operators that compute 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 operator interface. To keep things simple and consistent over all choices of join algorithms, we will limit ourselves to equality joins of a *single pair of columns* (represented by offsets to columns in the relations' schemas). Therefore, a join operator is initialized with four parameters: an operator for the outer relation, an offset of a column in the outer relation's schema, an operator for the inner relation, and an offset of a column in the inner relation's schema, resulting in the constructor `...Join(Operator outer,`

`int outerOffset, Operator inner, int innerOffset)`. Your join operator will implement *one* of the following join algorithms that were presented in the lecture, sorted in ascending order of expected 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 use index implementations that are included in the source code provided for this assignment. This time, the provided index does not only work on integer keys, but on all other data types that are supported by Minibase by using a `SearchKeyType`. An index search key can be *atomic*, consisting of a single attribute, or *composite*, consisting of multiple attributes. Since we only need to support single-column equi-joins in this exercise, an atomic key type suffices. The abstract class `AtomicKeyType` already provides instances (as static members) that describe single-attribute key types for all supported data types. The static method `SearchKeyType.get(DataTye type, int length)` maps to the `AtomicKeyType` for the given data type (having the given length in bytes).
Since the inner relation is represented by an index, you will not be able to implement this join variant using the above mentioned constructor. Therefore, the constructor has to expect the inner relation's `File` (e.g., `HeapFile`) and an `Index` (e.g., `BTreeIndex`) instead of the inner relation's operator. Since the `BTreeIndex` is unclustered, the additional file is needed to retrieve record data. The resulting constructor is `IndexNestedLoopsEquiJoin(Operator outer, int outerOffset, File innerFile, Index innerIndex, int innerOffset)`. The inner relation's index has to be initialized with the correct `SearchKeyType` for the inner relation's column that participates in the equi-join predicate.
In order to find matching tuples in the inner relation, the index has to be probed using a `SearchKey`. Such a search key can be obtained for a given *outer record* by the schema instance that the record conforms to, e.g, by calling `schema.searchKey(outerRecord, new int[]{ outerOffset })`. Note, that you should compute the array parameter in this call only once in the constructor to avoid unnecessary allocations; it is made explicit here for demonstration purposes only. Once you get a search key for an outer record, you can use this key to scan the index, retrieve matching record ids and read the records' data from the file.
Although an index scan or lookup can also be implemented as an operator, the current operator interface of Minibase does not make this approach non-obvious. If you are feeling adventurous, you can also try to solve this exercise using the more general constructor.

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 choose to implement this join variant, you can build on the external sort operator from Assignment 7 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., `IndexNestedLoopsEquiJoin00`, `SortMergeEquiJoin00`, or `HashEquiJoin00`, where `00` should correspond to your group number.

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 `EvaluatorBaseTest00` and `JoinTest00`, again substituting your group number. `EvaluatorBaseTest00` is a common base class for all test cases in the `minibase.query.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 (small) 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*: `date`, *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 chose to populate these byte arrays manually or by using the class `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 date.
- 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 `JoinTest00` (subject to substituted group number).

- Test that your algorithm works correctly if you join the two relations on the *sid* attribute.
- 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.

Be careful: Make sure that you run your tests with a sufficiently large data set, i.e., make sure that you have enough records having duplicate values in the join columns, such that all code paths are tested. Usually, it does not suffice to use 150 (or a similarly small number) tuples to test a join comprehensively!

i Submission

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

- `IndexNestedLoopsJoin00`, `SortMergeJoin00`, or `HashJoin00`, with `00` corresponding to your group number
- `EvaluatorBaseTest00` and `JoinTest00`, again replace `00` for 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!

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.