Assignment 6

Issue Date: December 18, 2018
Due Date: January 14, 2018, 10:00 A.M.
$\sum$ 40+20 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

External Sorting

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 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.query.evaluator`.
The following listing will give you an overview of the packages contained in the distribution. A sample solution for the B^+-Tree exercise is included as well.

```
minibase-sorting/src/main/java/
└── minibase
    ├── access
    │   ├── btree        // Sample solution from assignment 05
    │   └── file         // Run file implementation
    ├── catalog          // 'DataType' used in schema definitions
    ├── query
    │   ├── evaluator    // ExternalSort implementation
    │   │   ├── compare  // Comparator used for sorting
    │   │   └── predicate
    │   └── schema       // 'Schema' used for working with tuples
    ...
```

The important packages for your current task are package `minibase.query.evaluator`, where the implementation of your external sort operator will go, package `minibase.access.file`, where you will put the classes for your run file, and `minibase.query.schema`, where classes for working with tuples are located. Some concepts from the lecture are implemented in a specific way inside Minibase, mostly for reasons of *consistency* and *performance.* You are expected to adhere to them inside your implementation.

- *Operators* of an *evaluation plan* are represented by classes implementing the `Operator` interface. They represent the abstract representation of the query and encode static properties such as the *predicate* of a join/selection or the *ordering* of a sort operator.
- In order to evaluate a plan composed of operators, a `TupleIterator` can be requested by calling the `Operator#open()` method. All state regarding the current evaluation must be stored in the iterator and not the operator.
  A `TupleIterator` can be closed, at which point all resources and precomputed results are discarded. It can also be reset to the first record it emitted, potentially avoiding repeated work that would have to be done when closing the current iterator and opening a new one.
- A `Schema` is used to represent the schema of a collection of tuples. However, as Minibase does not use objects to represent tuples[1], an instance of the class `Schema` is used to work with tuple data (`byte[]`). The class `Schema` has a myriad of methods which can, for example, read and write attributes in the tuple at specified offsets directly converting them to the desired type. The caller of each method is responsible that the data contained in the `byte[]` actually conforms to the schema, otherwise garbage data might be read. In other words, do not confuse two schema instances having different schema definitions.
  Since a `Schema` is *immutable*, there exists a class `SchemaBuilder` that provides functionality to construct an instance of said class. The builder can not only start with an empty schema, but also *copy* (parts of) a given schema. The method `SchemaBuilder addField(String name, DataType type, int length)` takes as arguments a column name, a column data type and a length (in bytes) for a new column. It returns the schema builder itself such that calls to `addField()` can be *chained*.
  For example, the following snippet will create an empty schema, add two columns to it and finally make it immutable.

  ```
  Schema students = new SchemaBuilder()
                      .addField("name", DataType.VARCHAR, 127)
                      .addField("student_id", DataType.INT,
                                  DataType.INT.getSize())
                      .build();
  ```

  Notice that once `build()` is called, the schema cannot be changed any more as only the builder instance has methods for adding columns. Typically, you will not need a reference to the builder itself as it is only used to ensure immutability of the schema while simultaneously providing an easy to use interface to add an arbitrary number of arbitrarily defined columns to a schema.

A major difference of this assignment compared to previous assignments is the fact that you will start your implementation from scratch, i.e., with little scaffolding and without any 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 (bonus points are available along the way). Second, you will also get guidance (this extensive exercise description) and credit for these tasks.

Hint: The first two exercises are *completely* independent of each other and can be solved individually to increase productivity!

[1] Many tuple objects would be instantiated during query evaluation, resulting in high garbage collection pressure.

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.

a) In the `src/test/java` source tree, create a new JUnit test class named `ExternalSortTest` within the `minibase.query.evaluator` package. The class should extend `BaseTest` as that class provides functionality to set up a Minibase instance which you can use in your tests and which is created and deleted for each individual test method annotated with `@Test`. This way, you get access to an instance of `BufferManager` and `Random` that you can use right away.

b) In your tests, create a temporary heap file using the method `HeapFile#createTemporary()` and wrap the call in a *try-with-resource* block to make sure the file is automatically discarded at the end of the test. 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 class `Schema`. As you can see from class `Convert` 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 of 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 valid primary 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. To make your test data more interesting, you *can* also come up with a more elaborate name generator, possibly by sampling from predefined lookup tables.
- Values for the *rating* attributes are generated as random integers in the range $[0, \ldots, 10]$.
- Values for the *age* attribute are generated as random floats in the range $[14.0, \ldots, 99.9]$.

c) 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 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 `FileScan` in order to check that everything is as it should be. Note that a 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 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.[2] Instead, you will implement a very simple heap file that manages its pages as a linked list of pages as shown in Figure 1. In the lecture this file is referred to as a *run file*, however, we will split the concept into separate classes that together implement the behavior of a run file.
Since this run file will only be written to and read from sequentially, it suffices to store a forward pointer (*NextID*) in each page. The numbers in grey denote the number of bytes that are required to store this information. Additionally, a run file is either being written to *or* read from, which means that once a run file is *finished* it cannot be written to again and can only be read from. To enforce this, we split a run file into three separate classes: a class describing the data structure (a *run*), a class creating the data structure (a *builder*), and a class reading the data in the data structure (a *scan*).

a) The three skeleton classes, `RunPage`, `RunBuilder`, and `RunScan`, which you have to implement are located in the `minibase.access.file` package, the full `Run` class is also provided. You can have a 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.

[2] Since records are never deleted from runs, the lack of this guarantee would not adversely affect the correctness of the external sort.

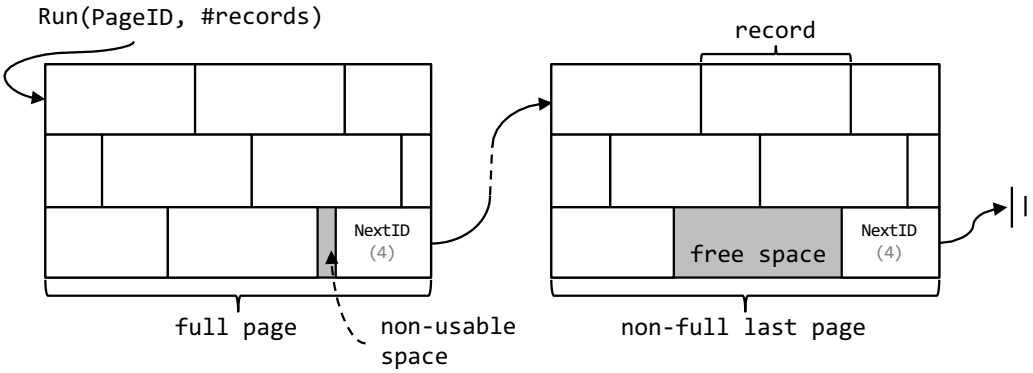

Figure 1: Page Layout of a Run

b) Class `Run` already provides the following functionality.
   - A run is a sequence of records stored on a linked list of `RunPages`. Only the last page is non-full.
   - Since all pages are singly-linked, it suffices to store the page id of the first page.
   - As runs are neither opened, nor closed, the number of records stored in a run is not persisted to disk.

c) Class `RunPage` implements the tagging interface `PageType` and will provide 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 on the page. Do not forget to provide the possibility to get a correctly initialized page.
   - Additionally, you will need methods to *append* records to a run and *read* records from a run, given a current run page and the overall position of the record in the run.

d) Class `RunScan` provides the following functionality.
   - The constructor of the class is initialized with the run over which the scan iterates. It pins the first page as the current page to read from. As the capacity of a run page does not change, it is a good idea to retrieve this information only once.
   - Method `hasNext()` returns `true` if there are more records in the run and `false`, otherwise. There are more records in the file, if the scan has not yet reached the end of the run, i.e., returned the number of records that are stored in the run.
   - Method `next()` returns the next record in the file. The method throws an exception if it is called in a state where `hasNext()` is `false` or if the scan has already been closed.
   - Method `reset()` resets the scan to be used again, i.e., from the start of the run. If a scan has already been closed, it should not be possible to reset it.
   - Finally, the `close()` method unpins any pinned pages and invalidates all internal state, such that attempts to reuse the scan will result in an exception.

e) Class `RunBuilder` will have functionality to create a run of records.
   - The constructor will receive the buffer manager, such that it is able to actually request new pages, and the length of a fixed-size record. It will also initialize the first page of the run.
   - Method `void appendRecord(byte[] record)` will insert a record into the run the builder is currently constructing and when necessary extend the run by a page to accommodate the new record.
   - Method `Run finish()` will return an instance of `Run` (effectively returning an immutable run that can now only be read). Additionally, the method should throw an exception if the builder was already finished.

Again, it is recommended that you write a couple of records to your run file (mind the sort order!) and then print them to the console using 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 partition the incoming tuples from the sort operator's *input* into *runs* that are sorted according to the `RecordComparator` that describes the sorting order of the current sort operation. You only have to implement the most basic variant of this pass, which generates runs that are exactly one `Page` in size. If you want to earn bonus points, you can instead skip the simple implementation and write a more advanced variant producing longer runs which is described below.

a) Create a new class `ExternalSort` in package `minibase.query.evaluator` inside the `src/main/java` source tree that is a subclass of `AbstractOperator`.

b) The constructor of this class is initialized with references to the buffer pool and to its input expression (represented by its root operator). Additionally, the external sort operator needs to know the *sort ordering* to be applied. We represent this information as a `RecordComparator` object used to compare records to each other, which leads to the following constructor.

```
public ExternalSort(final BufferManager bufferManager,
      final Operator input, final RecordComparator comparator)
```

The super-constructor of `AbstractOperator` needs the schema of the operator's result. Since sorting does not change the schema, you can just pass it the schema of the input operator.

c) The only other method required in the `ExternalSort` class is `TupleIterator open()`. What happens inside this method is divided into two phases.

i) All processing that has to be done before tuples can be returned is done directly inside `open()`.

ii) After that a `TupleIterator` is returned that does the rest of the work and emits the result tuples.

Since sorting is a fully blocking operation (i.e. we have to read the whole input before emitting the first output tuple), all sorting is done in the first part. After implementing Exercice 4 you should end up with a single sorted `Run` that you can iterate over in the second one. Note however that all pages you use have to be freed when the result iterator is closed.
As described above, you can choose between two ways of implementing the sort phase.

- **Basic Variant:** (+0 Points)
  After opening an iterator of the input operator, you have to fill one `Page<RunPage>` at a time with tuples from the input until the input iterator is drained. The records in each initial run (i.e. on each page) have to be sorted, which you can achieve either while filling the page (e.g. with *Insertion Sort*) or afterwards using e.g. *Quick Sort*.
- **Advanced Variant:** (+10 Points)
  A more efficient way of creating initial runs from the input tuples is to both more memory and exploit partial sortedness already present in the incoming data. This can be achieved by the *Replacement Sort* algorithm presented in the lecture.
  Your implementation must be configurable in the size of the internal buffer. After initially filling the buffer, it must perform the following steps until the input iterator and the buffer is empty.
  - Remove the smallest record (according to the record comparator) from the buffer.
  - If it is not part of the current run, finish that run and start a new one.
  - Append the record to the current run.
  - If the input is not drained, add the next record from it to the buffer.

  Try to find a good way to organize the order of records inside the buffer so that repeatedly finding the smallest one and inserting new ones can be done efficiently.

d) All initial runs generated in this exercise are appended to a *run of runs*, i.e., a `Run` of records containing the `PageID` of each run (as an `int`) and the number of records in it as a `long`.

Exercise 4: Pass $1, \ldots, \lceil \log_k N \rceil$: Merge Phase (10 Points)

After having partitioned the input tuples into sorted runs, the merge phase iteratively combines these until only one run remains. Again there are two ways of solving this exercise, one of which is awarded with bonus points.

a) You start with the run of initial runs generated by Pass 0 from Exercise 3.

b) While there is more than one run left after the previous phase, you generate a new run of runs by merging multiple runs of records together. Use `RunScans` to iterate over the input runs and a `RunBuilder` to create the output run. Also make sure that the input runs are freed after they have been merged.

- **Basic Variant:** (+0 Points)
  For *Binary Merge Sort* you always merge two input runs into one output run. Make sure that you retain duplicates and that the resulting run is again sorted according to the `RecordComparator` given to your `ExternalSort`.
- **Advanced Variant:** (+10 Points)
  A more realistic implementation of the *sort* operator should merge more than two runs at a time. Implement this using the *Tree of Losers* described in the lecture to efficiently select the tuple to emit. If k is the number of runs to merge at once, then the Tree of Losers needs an array of byte arrays (`byte[][]`) of length k to store the tree. The first element is the currently smallest element of all k input runs, for each index $1 \leqslant i < n$ the array contains the loser (i.e. the greater record) of its two sub-trees. The children of the node with index i in the array are at indexes $2i$ and $2i + 1$. If an index j is not in the array ($k \leqslant j < 2k$), then it points to the input run with index $j - k$.

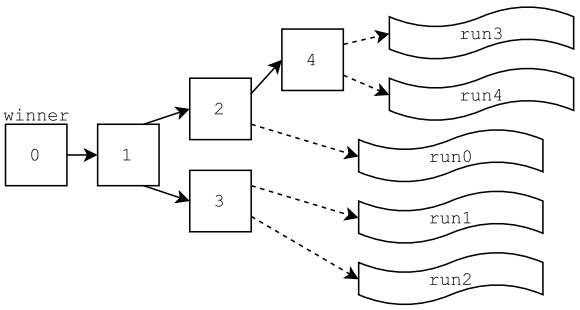

Figure 2: Structure of a Tree of Losers with $k = 5$

Additionally you need some data structure to remember which child each of the $k - 1$ losers in the tree came from. The algorithm then proceeds as follows.

i) The tree is initially filled with one tuple from each input by propagating them up the tree. The winner at each level is moved to the parent and we record if the left or right child won. At the end of this process the first element of the array contains the smallest tuple.

ii) Now we can remove the winner and write it to the output run.

iii) After that we have to refill the tree from below. Since all input runs are sorted, the tree currently contains the smallest tuple of all inputs except for the one the removed tuple came from. We can find that input by tracing the markers of who won/lost down the tree. Then we refill the tree with the next tuple from that input.
If an input is drained, `null` can be used as the next tuple instead to mark an empty slot. We just have to make sure that `null` is compared as greater than any other tuple.
When propagating the new tuple up the tree, we again adapt the winner/loser markers.

iv) Repeat steps ii) and iii) until the *winner* becomes `null`, at which point all input runs are drained and the merge is completed.

c) After only one run remains, return a `TupleIterator` over it from `ExternalSort#open()` that frees all pages when the iterator is closed. You can either integrate this functionality into the `RunScan` or write another iterator.

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 record comparator as the external sort 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 (i.e., test your implementation of Exercise 3).
- Create a JUnit test that verifies the final result (your implementation in Exercise 4). If you were not able to solve Exercise 3, you can still test the final result by manually creating sorted runs and then passing them to the merge phase. Afterwards you can assert that the resulting output is sorted correctly and that you did not lose, duplicate or invent tuples.

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 data and data that is 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 field types used in the **Sailors** schema. You can test this by sorting once on each field of the schema.
- Test that your algorithm sorts records correctly even if a multi-field record comparator is specified. Again, it is interesting to include the *sid* column in this key as it is already sorted.

i Submission

Solutions are submitted electronically via 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.

- `RunPage`, `RunBuilder`, and `RunScan`
- `ExternalSort` and additional classes you created yourself.
- `ExternalSortTest` and any implementations of `RecordComparator` and other classes 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.