CSI 410. Database Systems – Fall 2021

Programming Assignment III

The total grade for this assignment is 100 points. The deadline for this assignment is **11:59 PM, December 13, 2021**. *Submissions after this deadline will not be accepted.* Students are required to enter the UAlbany Blackboard system and then upload a .zip file (in the form of [first name]_[last name].zip) that contains the Eclipse project directory (please *exclude* all of the temporary `.run` files produced by the program) and a *short document* (5 grade points) describing:

- any missing or incomplete elements of the code
- any changes made to the original API
- the amount of time spent for this assignment
- suggestions or comments if any

In this programming assignment, you need to implement an *external sort* solution that sorts a large number of data elements using disk in ascending order according to the `Comparable` *natural ordering* of the elements. To sort large amounts of data that cannot fit into the main memory, external sort (1) repeatedly loads a portion of the data into the memory, sorts the data in memory, and saves the sorted data in a new file (called a run) until it saves all of the data in runs and then (2) keeps merging a number of runs into a sorted run until all runs are eventually merged (and thus the data can be accessed in order). For further details of external sort, refer to Section 15.4 of the textbook.

For this assignment, you first need to run Eclipse on your machine and import the "`external_sort`" project (see Appendix A). Please generate an API document (see Appendix B) and then take a look at that document as well as the source code to familiarize yourself with this assignment. This assignment provides you with a set of incomplete classes (see `InputBuffer`, `RunWriter`, `RunReader`, `OrderedMergeIterator`, `ExternalSort`, and `OptimizedExternalSort`). You will need to write code for these classes. Your code will be graded by running a set of unit tests and then examining your code (see `InputBufferTest`, `RunWriterTest`, `RunReaderTest`, `OrderedMergeIteratorTest`, `ExternalSortTest`, `ExternalSortConfigurationTest`, and `OptimizedExternalSortTest` which use JUnit[1]). Note that passing the unit tests does NOT necessarily guarantee that your implementation is correct and efficient. Please make sure that your code is correct and efficient so that it will not cause any problem in many other cases not covered by the unit tests. If you have questions, please contact the TA(s) or the instructor. The remainder of this document describes the components that you need to implement.

Part 1. Buffering (30 points)

External sort typically employs buffering (i.e., reading/writing multiple disk blocks each time) to reduce the number of disk seeks. In this assignment, each `InputBuffer` instance maintains a fixed size byte array and enables easy retrieval of the data elements stored (i.e., serialized) in that byte array. In this part, you need to complete only the `iterator()` method in the `InputBuffer` class. When this method is invoked on an `InputBuffer`, this method must return an `Iterator` over the data elements stored in the byte array of the `InputBuffer`. Given the byte array `buffer`, this

[1] `http://junit.org`

`iterator()` method needs to construct an `ObjectInputStream` to read (i.e., deserialize) the data elements stored in `buffer`. To read each data element, this method needs to call the `readObject()` on the `ObjectInputStream`. After implementing the `iterator()` method, please verify your code by running `InputBufferTest`.

Part 2. Writing Runs (20 points)

In this assignment, each run is written by a separate `RunWriter`. Given an `Iterator` over data items, a `RunWriter` opens a file (i.e., a run) and then saves the data items in that file. In this part, you need to implement the constructor of `RunWriter`. This constructor must *repeatedly* call the `write(Iterator<T> iterator, int bufferSize)` method which (i) constructs an `OutputBuffer` whose size is specified by `bufferSize`, (ii) writes data elements from `iterator` to the `OutputBuffer` until the `OutputBuffer` is full (i.e., runs out of space), and then (iii) returns the byte array of the `OutputBuffer`. The return value of `write(Iterator<T> iterator, int bufferSize)` (i.e., the byte array containing data elements) must be saved in the file specified by `fileName`. To save a byte array `buffer` in the file, create a `FileOutputStream` for that file (specified by `fileName`) and then call `write(buffer)` on that `FileOutputStream`. Note that it is in general necessary to call `write(iterator, bufferSize)` and `write(buffer)` *many times* to save all of the data items in the run (e.g., when all of the data items amount to 10GB and the buffer size is 1MB, 10,000 (=10GB/1MB) times). For each invocation of the `write(buffer)` method, call `increaseBufferWriteCount()` on `externalSort` to inform `externalSort` of the buffer write operation on the run (`externalSort` keeps track of statistics such as the number of buffer reads, the number of buffer writes, the bytes written so far, etc.)

After completing this part, make sure that your code passes the unit test in `RunWriterTest`.

Part 3. Reading Runs (20 points)

A `RunReader` can read a run written by a `RunWriter`. A `RunReader` can also be viewed/used as an `Iterator` over the data elements stored in a run. The constructor `RunReader(String fileName, int bufferSize, ExternalSort<?> externalSort)` of `RunReader` opens a file specified by `fileName` and then creates a `FileInputStream` to read data elements from that file. In this part, you need to implement the `hasNext()` and `next()` methods of `RunReader`. To be compatible with `RunWriter`, the `RunReader` implementation needs to repeatedly constructs an `InputBuffer` and then fills that `InputBuffer` with the data elements from the run. To obtain such an `InputBuffer` filled with data, call the `read()` method in `RunReader`. To access the data elements stored in an `InputBuffer`, call `iterator()` on that `InputBuffer`. `RunReader`'s `hasNext()` and `next()` methods must support iteration over all of the data elements stored in the run specified by `fileName` (i.e., the data elements that can be accessed via `InputBuffer` instances as explained above).

After completing this part, please verify your code using `RunReaderTest`.

Part 4. Ordering Data Elements from Multiple Iterators (10 points)

External sort requires the capability of merging multiple sorted runs into a single sorted run. The `OrderedMergeIterator` class provides this capability. Given multiple `Iterator`s each of which iterates over elements in ascending order, an `OrderedMergeIterator` can iterator over all of these elements in ascending order.

In this part, you need to implement the `next()` method of `OrderedMergeIterator` (`hasNext()` is already implemented). To remember the most recent data element from each input `Iterator` and quickly get the *smallest* among these most recent data elements (i.e., the data item that must precede all others), `OrderedMergeIterator` uses a priority queue `queue` storing (last data element, iterator) pairs. The `next()` method needs to return the smallest over the most recent data elements from the input `Iterator`s. That smallest data element can be obtained by calling `poll()` on `queue` and then calling `getKey()` on the return value of `poll()`. For example, assume

2 iterators i_1 to iterate over $2, 4, 6, \cdots$ and i_2 to iterate over $1, 3, 5, \cdots$. Initially, `queue` needs to contain two entries $(2, i_1)$ and $(1, i_2)$ since the first elements from i_1 and i_2 are 2 and 1, respectively. Then, `queue.poll()` returns $(1, i_2)$ since $1 < 2$ and thus `next()` needs to return 1. Furthermore, $(3, i_2)$ needs to be inserted into `queue` because i_2 still has more elements and 3 is the next item from i_2. After this step, between $(2, i_1)$ and $(3, i_2)$, `queue.poll()` returns $(2, i_1)$, meaning that `next()` needs to return 2. Then, $(4, i_1)$ needs to be inserted into `queue`.

When you complete `OrderedMergeIterator`, please run `OrderedMergeIteratorTest` to verify your implementation.

Part 5. The Basic External Sort Implementation (10 points)

In this part, you need to implement a basic form of external sort in `ExternalSort.java`. An `ExternalSort` obtains data elements through an input `Iterator` and then sorts data elements in ascending order according to the `Comparable` natural ordering of the elements in Java. Each `ExternalSort` can also be viewed as an iterator over the data elements sorted by that `ExternalSort` (i.e., to retrieve all of the data elements sorted by an `ExternalSort`, repeatedly call `hasNext()` and `next()` on that `ExternalSort`). The constructor of `ExternalSort` creates initial runs (files containing a partial collection of data elements in ascending order) and then carries out multiple merge passes that merge each group of runs into a single sorted run. For example, an `ExternalSort` may create initial runs r_0, r_1, r_2, and r_3. Then, merge pass 1 may merge r_0 and r_1 into a new run (denoted by $r_0 \oplus r_1$) and merge r_2 and r_3 into a new run (denoted by $r_2 \oplus r_3$). In this case, merge pass 2 can merge $r_0 \oplus r_1$ and $r_2 \oplus r_3$ into $r_0 \oplus r_1 \oplus r_2 \oplus r_3$. For further details of external sort, refer to Section 15.4 of the textbook. The common size of the groups of runs that are merged together into a run is called the *degree* (e.g., when the degree is 2, each merge pass forms groups of 2 runs and merges each group of 2 runs into a single sorted run).

In this part, you need to complete only the `merge(List<String> runNames)` method (the rest of `ExternalSort` is already implemented). This method must (i) group the current runs (whose file names are stored in `runNames`) according to the user-specified degree (specified by the member variable `degree` in `ExternalSort`), (ii) construct a new run for each group of runs, and (iii) return a list of `String`s that contain the file names of the newly created runs. After grouping run names according to the member variable *degree*, for each group `g` of run names (e.g., "`test0.run`" and "`test1.run`"), `merge(List<String> runNames)` can construct a new run and obtain the file name of that new run (e.g., "`test3.run`") by calling `createRun(g)`.

For example, assume that the method is given runs that are named "`test0.run`", "`test1.run`", "`test2.run`", and "`test3.run`" and the degree is 2. Then, the method needs to merge "`test0.run`" and "`test1.run`" into a new run (say, "`test4.run`") and merge "`test2.run`" and "`test3.run`" into a new run (say, "`test5.run`") and then return a list containing "`test4.run`" and "`test5.run`".

In the above example, if the degree is changed to 3, "`test0.run`", "`test1.run`", and "`test2.run`" need to be merged into a new run (say, "`test4.run`"). Furthermore, since there is only one run remaining (i.e., "`test3.run`"), no additional merging can be done. In this case, `merge(List<String> runNames)` needs to return a list containing "`test3.run`" and "`test4.run`".

If the degree is 3 and runs "`test0.run`", "`test1.run`", "`test2.run`", "`test3.run`", and "`test4.run`" are given, then `merge(List<String> runNames)` needs to merge three runs "`test0.run`", "`test1.run`", and "`test2.run`" into a new run (say, "`test5.run`") and merge the remaining runs "`test3.run`" and "`test4.run`" into a new run (say, "`test6.run`"). It then needs to return a list containing "`test5.run`" and "`test6.run`".

When you complete the `merge(List<String> runNames)` method, please verify your code using `ExternalSortTest`. It would also be interesting/important to understand the impact of parameters (`degree`, as well as `bufferSize` which specifies the size of the buffers used by `RunReader`s and `RunWriter`) on the performance of `ExternalSort`. For this understanding, run `ExternalSortConfigurationTest`

which will produce results similar to the following:

```
degree: 7
bufferSize: 65536
initial pass: 5000 runs (11.00199 seconds)
merge pass 1: 715 run(s) (10.86025 seconds)
merge pass 2: 103 run(s) (7.29554 seconds)
merge pass 3: 15 run(s) (9.97600 seconds)
merge pass 4: 3 run(s) (7.22050 seconds)
merge pass 5: 1 output iterator
retrieved 10000000 elements (2.68313 seconds)
number of buffer reads: 13482
number of buffer writes: 13482
bytes read: 883556352
bytes written: 883556352
elapsed time:  49.04074 seconds

degree: 63
bufferSize: 8192
initial pass: 5000 runs (7.77232 seconds)
merge pass 1: 80 run(s) (11.89483 seconds)
merge pass 2: 2 run(s) (9.54373 seconds)
merge pass 3: 1 output iterator
retrieved 10000000 elements (2.91450 seconds)
number of buffer reads: 54545
number of buffer writes: 54545
bytes read: 446832640
bytes written: 446832640
elapsed time:  32.12621 seconds

degree: 511
bufferSize: 1024
initial pass: 5000 runs (12.23826 seconds)
merge pass 1: 10 run(s) (23.08204 seconds)
merge pass 2: 1 output iterator
retrieved 10000000 elements (5.99998 seconds)
number of buffer reads: 297065
number of buffer writes: 297065
bytes read: 304194560
bytes written: 304194560
elapsed time:  41.32151 seconds
```

The above results are obtained from three different experiments that commonly performed external sort using only .5MB of main memory for buffering. The first experiment set the degree to 7 (i.e., 7 input buffers and one output buffer for merging each group of 7 runs into one run) and thus set the buffer size to 64KB (=0.5MB/8). In this experiment, the small degree (7) leads to much more merge passes (i.e., much more bytes read and written) compared to the other two experiments. On the other hand, the third experiment set the degree to 511 (i.e., 511 input buffers and one output buffer for merging each group of 511 runs into one run) and thus set the buffer size to 1KB (=0.5MB/512). In this third experiment, merging too many runs each time incurs relatively high overhead (particularly causing many disk seeks) compared to the other two experiments. The second experiment did not suffer the problems mentioned above and its elapsed time was shorter than those of the other two experiments.

Part 6. Optimization (5 points)

The `ExternalSort` class implemented in Part 5 may incur unnecessarily high disk I/O overhead. For example, assume that an `ExternalSort` instance has constructed 5 initial runs r_0, r_1, r_2, r_3, and r_4. During merge pass 1, it merges r_0 and r_1 into $r_0 \oplus r_1$ (i.e., a sorted run containing the data

elements from r_0 and r_1), and then r_2 and r_3 into $r_2 \oplus r_3$ while keeping r_4. During merge pass 2, it merges $r_0 \oplus r_1$ and $r_2 \oplus r_3$ into $r_0 \oplus r_1 \oplus r_2 \oplus r_3$ while keeping r_4. Finally, it merges $r_0 \oplus r_1 \oplus r_2 \oplus r_3$ and r_4 into $r_0 \oplus r_1 \oplus r_2 \oplus r_3 \oplus r_4$. Throughout these three merge passes, the data in the 5 initial runs are accessed as follows:

run name	r_0	r_1	r_2	r_3	r_4
number of writes	3	3	3	3	1
number of reads	3	3	3	3	1

However, it is possible to more efficiently carry out external sorting, for example, by merging only r_0 and r_1 into $r_0 \oplus r_1$ (merge pass 1), merging $r_0 \oplus r_1$ and r_2 into $r_0 \oplus r_1 \oplus r_2$, as well as r_3 and r_4 into $r_3 \oplus r_4$ (merge pass 2), and then merging $r_0 \oplus r_1 \oplus r_2$ and $r_3 \oplus r_4$ into $r_0 \oplus r_1 \oplus r_2 \oplus r_3 \oplus r_4$ (merge pass 3). In this scenario, the data in the 5 initial runs are accessed as follows (when the size of each run is 1GB, this and the above scenarios read/write 12GB of data and 13GB of data, respectively):

run name	r_0	r_1	r_2	r_3	r_4
number of writes	3	3	2	2	2
number of reads	3	3	2	2	2

In this part, you need to implement the `OptimizedExternalSort` class so that it runs faster than `ExternalSort` by performing a smaller number of read and write operations just like the example mentioned above. Please feel free to develop a merging strategy by yourself and then describe your strategy in the document that you submit. When `OptimizedExternalSort` is implemented adequately, `OptimizedExternalSortTest` will produce some output as follows (the exact result would vary depending on your merging strategy and the machine that you use, but `OptimizedExternalSort` should be more efficient than `ExternalSort` in terms of `number of buffer reads`, `number of buffer writes`, `bytes read`, `bytes written`, and `elapsed time`):

```
degree: 2
bufferSize: 4096
initial pass: 5 runs (6.95196 seconds)
merge pass 1: 3 run(s) (7.20901 seconds)
merge pass 2: 2 run(s) (6.59692 seconds)
merge pass 3: 1 output iterator
retrieved 10000000 elements (3.18204 seconds)
number of buffer reads: 64683
number of buffer writes: 64683
bytes read: 264941568
bytes written: 264941568
elapsed time:  23.94977 seconds

degree: 2
bufferSize: 4096
initial pass: 5 runs (5.33121 seconds)
merge pass 1: 4 run(s) (3.38210 seconds)
merge pass 2: 2 run(s) (8.33198 seconds)
merge pass 3: 1 output iterator
retrieved 10000000 elements (3.03896 seconds)
number of buffer reads: 59708
number of buffer writes: 59708
bytes read: 244563968
bytes written: 244563968
elapsed time:  20.08529 seconds
```

Appendix A. Importing a Java Project

1. Start Eclipse. When Eclipse runs for the first time, it asks the user to choose the workspace location. You may use the default location.

2. In the menu bar, choose "File" and then "Import". Next, select "General" and "Existing Projects into Workspace". Then, click the "Browse" button and select the "`external_sort.zip`" file contained in this assignment package.

3. Once the project is imported, you can choose one among `InputBufferTest`, `RunWriterTest`, `RunReaderTest`, `OrderedMergeIteratorTest`, `ExternalSortTest`, `ExternalSortConfigurationTest`, and `OptimizedExternalSortTest` in the `external_sorting.test` package and then run it.

Appendix B. Creating API documents using `javadoc`

One nice feature of Java is its support for "documentation comments", or "`javadoc`" comments, which you can use to automatically produce documentation for your code. Javadoc comments start with "`/**`". Inside a javadoc comment, there are some special symbols, like `@param` and `@return`.

You can create HTML-based API documents from the source as follows:

1. Click the "`external_sorting`" project icon in the Navigator or Project Explorer window.

2. Select "Generate Javadoc" from the "Project" menu.

3. In the "Generate Javadoc" dialog box, press the "Finish" button.

As it runs, it tells you that it's generating various things. When it is finished, a few new folders should appear in your project: `doc`, `doc.javadoc`, and doc.resources. See what got generated (to open the newly created HTML documentation files in a web browser window, just double-click them; you can start with "`index.html`").