# Advanced Databases :: First practical assignment

Stratis Viglas
sviglas@inf.ed.ac.uk

In this assignment, you will implement two algorithms in the context of the *attica* RDBMS. The first algorithm is external merge-sort and the second algorithm is hash-based grouping. In what follows we will go through the steps you need to carry out in order to complete the practical work.

## Download the code-base

The first thing to do is to download the code-base from *attica*'s location:

`http://www.inf.ed.ac.uk/teaching/courses/adbs/attica`

The system is a feature-complete relational query processor with added "hooks" in the code so you can modify the system's behaviour and implement a few extra algorithms that have not been implemented as of now.

After downloading *attica*, and if you are familiar with the `ant` build system you can use an `ant` build file to compile *attica*. The build system should be available for your platform, but in case it is not you can obtain `ant` from:

`http://ant.apache.org/`

The *attica* distribution contains a build file that you can use to compile the system. If you decide to use `ant` as your build system, these are (some of) the targets of the build file:

- `ant build` builds the system,
- `ant dist` builds the `jar` file and the `javadoc`'s for distribution,
- `ant clean` cleans up the installation,
- `ant cleanAll` cleans up everything.

This functionality is also covered in the notes from the lab sessions so you can refer to them if necessary.

An alternative is to import the source files into your favourite IDE and use that to control the environment. Regardless of your choice of environment and/or build system, however, your submitted work needs to run on DICE.

## Implementation

As a rough guide, the steps you have to follow are the following:

- Look into the files `ExternalSort.java` and `HashGroup.java`.
- Initialise all the temporary files you are going to need.
- Implement the external sort algorithm, and hash-based grouping.
- In the context of your implementation you will store the sorted /partitioned results in temporary files that you will need to access in order to return results to the caller.
- Modify the optimiser (`PlanBuilder.java`) so that it will pick up your code.

More details about your implementations follow.

---

### External merge-sort

Open `ExternalSort.java` under `org/dejave/attica/engine/operators` of your source installation. This is the main class that you will have to modify. The class's constructor has the following signature:

```
/**
  * Construct a new external sort operator.
  *
  * @param operator the input operator.
  * @param slots the slots acting as sort keys.
  * @param buffers the number of buffers (i.e.,
  * output files) to be used for the sort.
  * @throws EngineException thrown whenever the sort operator
  * cannot be properly initialized.
  */
public ExternalSort (Operator operator, StorageManager sm,
                     int [] slots, int buffers)
    throws EngineException { ...
```

The proper values of the first two arguments will be taken care of by the caller of the constructor—in this case, *attica*'s heuristics-based optimiser.

The third argument is an array of integers. These integers correspond to slots in the input relation of the operator and designate the sort keys. For instance, if the array values are `[1, 0, 2]` that means that the primary sort key is the value in the 1st slot of the tuple. The secondary sort key is the value in the 0th slot of the tuple, while the third sort key is the value in the 2nd slot of the tuple. The optimiser will set these values for you (you do not have to do any range-checking) but you will have to use these values to access the fields of the tuple that you are sorting by.

The fourth argument is the number of buffer pool pages allocated for the sort. This number should be less than the number of available buffer pool pages and the optimiser will ensure this is the case. There is a limit to the number of open files you can have, but that is actually determined by the operating system and depends on your local installation. As a guide, a 64-bit program using standard I/O can use up to 2 billion descriptors. For all sensible uses of your code, you will not reach this upper bound.

Moving on into `ExternalSort.java`, there is a method called `initTempFiles()`. You should insert your own code here if you would like to generate any temporary files that will be used for sorting and that you know of at this point in time. Notice that this is only for the initialisation phase. In reality, you will have to generate more than one files, i.e., during the intermediate passes of the algorithm.

The following code, initialises a temporary file, monitored by attica's storage manager:

```
String filename = FileUtil.createTempFileName();
sm.createFile(filename);
```

where `sm` is a `StorageManager` instance.

If you want to have a look at how these files can be created inside an operator, look at the constructor of `NestedLoopsJoin.java` under the directory `attica/engine/operators` of your source installation (lines 74 through 77.)

The next step is implementing the external sort algorithm. For this, you will have to modify the `setup()` method of class `ExternalSort`. Of course, not your entire implementation has to be in this single method! But when the method exits, the sorted result should be stored in the designated output file.

To re-iterate from lectures, here is a rough sketch of the algorithm to sort a file using *B* buffer pool pages:

1. Read in the input file in batches of *B* pages, sort them in main memory and write them out to a temporary file. If the file had *X* pages originally, this will generate *X*/*B* temporary sorted files on disk.
2. Read in *B*-1 temporary files and merge them into a new temporary file. Use one page for output (i.e., keep merging from the *B*-1 files into the output page, and as soon as the output page fills up, flush it to disk.)
3. Read in the next *B*-1 files and merge them. Continue until the *X*/*B* files are exhausted.

4. Apply steps 2 and 3 for the new set of temporary files and iterate until you are left with one big sorted file.

An alternative to this is to use replacement selection with two heaps, the current heap and the next heap. That is, instead of batching the input into chunks of *B* pages maintain two heaps of a combined size equal to the number of tuples that can fit into these *B* pages and then:

1. Read in *B* pages and turn them into a heap. This is the current heap; the next heap is empty.
2. While the input is not exhausted, apply the following steps.
   A. Output the minimum value from the current heap into the file for the current run.
   B. Read the next value from the input and decide if it belongs to the current heap or the next heap. Place it in the appropriate heap and resize the heaps if necessary.
   C. If the current heap becomes empty, close the run, start a new run, and swap the heaps (that is, the next heap becomes the current heap and the new next heap is empty).

To implement the algorithm you will have to do relation-level I/O, using *attica*'s storage manager. This I/O can be initialised with the following code:

```
Relation rel = getInputOperator().getOutputRelation();
RelationIOManager man =
    new RelationIOManager(getStorageManager(), rel, inputFile);
```

which initialises a relation I/O manager; input is read from `inputFile`, which is simply a file name (NB: this should be a file monitored by the storage manager, e.g., the file you created previously through a call to `StorageManager.createFile()`) and the tuples in the file are expected to conform to the relational schema referred to by `rel`. You can use a `RelationIOManager` instance to read/write tuples, through the `nextTuple()`/`insertTuple()` methods it provides.

You may also need to use the page-level functionality of `RelationIOManager`. The interface is very similar to the tuple-level interface. The following code reads all the pages in a relation:

```
RelationIOManager manager = ...

...
for (Page page : manager.pages()) {
    // manipulate the tuples in the page
}
```

You will have to determine yourselves when you need page-level and when tuple-level I/O functionality.

If you need to write a page out to disk you need to make a simple call to the `StorageManager` through the method `writePage(page)`where `page` is the `Page` instance you need to write. You will be able to manipulate the contents of the page by using the `setTuple()` and `retrieveTuple()` methods of the `Page` class.

When implementing the merge phase, you do not need to manipulate the buffer pool. Instead, you can continuously write tuples to the output file designated in your call, by making continuous calls to the `insertTuple()` method of the output `RelationIOManager` backing your output file. The storage manager of *attica* will take care of the rest (i.e., pagination, outputting a page as soon as it becomes full and so on).

After you have implemented both the sort and the merge phases of external sort, the final, sorted result, should be stored in the output file pointed to by the `outputFile` field of `ExternalSort` (which `outputManager` writes to.) This file will then be scanned during the retrieval of the sorted relation. Again, you might want to take a look at `NestedLoopsJoin.java` to see how this is done in the context of a different operator.

The final step you should take: open `PlanBuilder.java` under `org/dejave/attica/engine/optimiser` of your source installation. Go to line `1232` under method `imposeSorts()`. Notice that a few lines are commented out—the system right now does no sorting at all. You should uncomment that block of code and comment out lines `1265` and `1266`, which implement what the system is doing right now when the user

requests a sort—which is, simply, nothing. After you have made these substitutions and re-compiled, then the next time you start the server and you issue an `order by` query, the optimiser will pick up your code and use it to sort the output. If you do not make this change your code will never run!

Notice that in the example call, external sort is implemented using half the buffer pool pages. However, you may want to experiment with various values for the number of buffer pool pages to ensure your code works in various boundary conditions.

---

## Hash-based grouping

For the second part of the assignment, you will need to implement hash-based grouping. For this to be the case you will need to access `HashGroup.java` under `org/dejave/attica/engine/operators`. The signature of the constructor of the operator is as follows:

```
/**
 * Constructs a new hash grouping operator.
 *
 * @param operator the input operator.
 * @param sm the storage manager.
 * @param slots the indexes of the grouping keys.
 * @param partitions the number of partitions (i.e., files) to be
 * used for grouping.
 * @throws EngineException thrown whenever the grouping operator
 * cannot be properly initialized.
 */
public HashGroup(Operator operator, StorageManager sm,
                 int [] slots, int buffers)
    throws EngineException { ...
```

Where, as was the case for ExternalSort the first two parameters will be set for you by the optimiser, and the last two parameters you will need to use in your implementation.

The third parameters, slots, is an array containing the grouping attributes that appear in the group by clause of the query. The semantics is similar to the ones for sorting: these are the indexes of the tuple slots you are grouping by. So if the array values is `[1, 0, 2]` then you are grouping by the attributes in the 1st, 0th, and 2nd slot of the tuple.

The fourth parameters is the number of buffers that you have available for hash grouping in the buffer pool. This means that the size of any partition you create should not exceed this memory budget.

Your implementation should work as follows:

- Scan the input and store it locally so you know how many pages there are in it.
- Given your budget, decide on the number of partitions that you need to create. If the input is *N* pages and you only have *B* pages available, then you should generate at least *N*/*B* partitions. To keep things safe, aim to double that number.
- Generate as many partition files as needed (one per partition).
- Scan the local input and apply a hash function on each tuple. Your hash function should access the designated slots of the tuple and decide which partition the tuple belongs to. A simple way to compute a hash value would be to take the results of calling `hashCode()` on each slot, combine them, and then take the results of the combined hash value *modulo K* where *K* is the number of partitions.
- You will now have as many files on disk as there are partitions. But your job is not done just yet! The reason is that the number of partitions may well be smaller than the cross product of the key cardinalities of the grouping attributes. That means that the results within each partition will be completely random and not grouped in any way.
- To rectify this, you will need to access each partition again and regroup it. To do that you have two options.
  - You can either load the partition into main memory, and sort it using a main memory algorithm. You should access the partition page-by-page and sort the contents across the pages—similarly to sorting the contents of a run of pages. You already know how to do that from external sorting.

  - Or, you can scan the partition file a tuple at a time and build a hash table for the partition that uses all grouping attributes as a key. You can then scan the hash table to generate the grouped version of the partition.
- Rehardless of which option you choose above, you will need to write out the new partition and delete the (ungrouped) partition.
- The rest of the implementation will iterate over all partition files to propagate the results.

In terms of your implementation in the context of the existing code in `HashGroup.java`, the first thing to note is a field called `partitionFiles` that is a list of `RelationIOManagers` for storing partitions. These should be the final partitions (i.e., the fully grouped results). You may need to allocate more files for the intermediate partitions.

Then, note the `initTempFiles()` method. This method currently generates a file prefix for all partition file names. You will need to manage the files you create.

Finally, in the `innerGetNext()` method, the current code assumes that the partitions are contained in the `partitionFiles` list of `RelationIOManagers`. You may need to modify this depending on your implementation (the code currently loops over the partitions in the list, so if you put more or fewer partitions there that may cause a problem).

Once you are finished with your implementation you need to connect it with the rest of the system. To do that, you will need to modify `PlanBuilder.java` again. Open the file and go to line `1288` and method `imposeGroups()`. Starting in line 1308 there is a block of commented out statements that you will need to uncomment and then you will need to comment out lines `1327` and `1328` that are the current grouping implementation of the system (which does nothing for the time—as was the case for sorting this is the null operator).

Note here that hashing algorithms are not enabled by default. What you need to do is issue the following instruction to the command line interpreter in order to use hash based-grouping:

```
aSQL> enable hash;
```

Doing that ensures that hashing is enabled for your grouping implementation. Otherwise, sorting will be used as the default grouping mechanism. If you want to disable hash-based algorithms, then issue:

```
aSQL> disable hash;
```

---

## Testing your implementation

For that purpose, please go to

http://www.inf.ed.ac.uk/teaching/courses/adbs/attica/WBGen.java

and download the linked data generator. The generator builds arbitrarily large synthetic datasets that conform (for the most part) to the logical/physical layout of the relations in the Wisconsin Benchmark. Compile the source file, and you will have the corresponding class file which can be executed as follows:

```
java WBGen <table-name> <number-of-tuples>
```

This command line will output into standard output the SQL commands to create a table by the name of `<table-name>` containing `<number-of-tuples>` tuples. The schema of the generated file is shown in Table 1 below.

You can redirect the output of the generator to a text file and use that text file to feed *attica* with tuples. For instance, the following sequence of commands generates the SQL statements for a table of 1,000 tuples named `sort_data`, stores the statements in a text file called `sort_data.sql` and then uses the text file as input to *attica* in order to actually store the table.

```
$ java WBGen sort_data 1000 > sort_data.sql
$ java org.dejave.attica.server.Database attica.properties < sort_data.sql
...
[a whole bunch of successful insertion messages]
```

You can then use the generated table for your tests.

| Column | Type |
|---|---|
| unique1 | long |
| unique2 | long |
| two | long |
| four | long |
| ten | long |
| twenty | long |
| onepercent | long |
| tenpercent | long |
| twentypercent | long |
| fiftypercent | long |
| unique3 | long |
| even | long |
| odd | long |
| stringu1 | string |
| stringu2 | string |
| stringu4 | string |

**Table 1. The schema of the relation generated by the test data generator.**

## Marking guidelines

The assignment is marked out of a possible 100 marks. Each of the two implementations is worth 50 marks for the assignment. Of these 50 marks per implementation:

25 marks are for a faithful implementation of the algorithm.
10 marks are for code cleanliness
15 marks are for code efficiency of the standard algorithm.

## What you need to hand in

The minimum set of files you will have to hand in are:

- the compiled version of your entire source tree as a single `jar` file, and
- the source code for your implementation of `ExternalSort.java` and `HashGroup.java` and any other files you have modified apart from the optimiser (`PlanBuilder.java`).

The submission is electronic only and the deadline is

**Friday, 13 February, 12:00 pm.**

Use the `submit` program to make the submission. For instance, to submit the compiled version of the source tree, use:

```
submit adbs 1 attica.jar
```

You get the idea for the source files. If you have modified any other source files of the code base, please submit them as well. You might also think of handing in a description (a text file will do) of what you did if you think something is worth mentioning. It is not compulsory, but it might make marking the assignment easier. You can write whatever you want in that file, ranging from implementation issues and problems you faced (hopefully, along with the solutions you provided!) to comments about the code in general.