Advanced Databases :: Second practical assignment

Stratis Viglas
sviglas@inf.ed.ac.uk

In this assignment you will implement two join algorithms in the context of the *attica* RDBMS. The first algorithm is merge-join and the second algorithm is Grace hash join. In what follows we will go through the steps necessary to successfully complete the assignment. Throughout, we assume that you have a working version of *attica* and you also have some experience with it through your first assignment.

Merge-join

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

- look into the `MergeJoin.java` file of your source distribution,
- initialise all the temporary files you might need,
- implement the merge-join algorithm, storing the join result in the pre-defined output relation, and
- modify the optimiser (`PlanBuilder.java`) so that it will pick up your code.

In more detail:

`MergeJoin.java` is 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 mergejoin operator.
  *
  * @param left the left input operator.
  * @param right the right input operator.
  * @param sm the storage manager.
  * @param predicate the predicate evaluated by this join operator.
  * @throws EngineException thrown whenever the operator cannot be
  * properly constructed.
  */
public MergeJoin (Operator left,
                  Operator right,
                  StorageManager sm,
                  int leftSlot,
                  int rightSlot,
                  Predicate predicate)
    throws EngineException { ...
```

The values of the first three arguments will be set by the caller of the constructor—as was the case in the first assignment, *attica*'s optimiser. The fourth and fifth argument should concern you, but only minimally. They are the indexes of the sort-attributes for either input relation (`leftSlot` for the left input, `rightSlot` for the right input.) You will need to access these attributes of the corresponding tuple to "advance" your relation pointers when implementing the merging phase. The last argument is the predicate itself. The predicate evaluator will compute the value of the predicate (details following), while the argument value to the constructor will, again, be taken care of by the optimiser.

Moving on into `MergeJoin.java`, there is a method called `initTempFiles()`. You should write your own code here in order to generate any temporary files that may be used for join evaluation. You already know how to do so from your previous assignment.

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

implement the merging phase, you will need to access the attributes on which the input relations are sorted (indexed by `leftSlot` for the left input and `rightSlot` for the right input.)

To look at another join implementation, consult `NestedLoopsJoin.java`. A crucial point is making the right call to the predicate evaluator. Assuming that the two tuples the join predicates is being evaluated on are `leftTuple` and `rightTuple`, then the call you need to make is the following:

```
Tuple rightTuple = rightMan.nextTuple();
PredicateTupleInserter.insertTuples(leftTuple, rightTuple, getPredicate());
boolean value = PredicateEvaluator.evaluate(getPredicate());
if (value) {
    // the predicate is true -- store the new tuple
    Tuple newTuple = combineTuples(leftTuple, rightTuple);
    outputMan.insertTuple(newTuple);
}
```

After you have implemented the merge phase of merge join, the final result should be stored in the output file pointed to by the `outputFile` field of `MergeJoin` (which `outputManager` writes to.) This file will then be scanned during the retrieval of the output relation. Again, you might want to take a look at `NestedLoopsJoin.java` to see how this is done.

Grace hash join

The second algorithm you will implement is Grace hash join. To do that you will need to modify the source of `GraceHashJoin.java` under `org/dejave/attica/engine/operators` of your source installation.

Your implementation should roughly work as follows:

- look into the `GraceHashJoin.java` file of your source distribution,
- initialise all the temporary files you might need,
- implement the Grace hash-join algorithm, storing the join result in the pre-defined output relation, and
- modify the optimiser (`PlanBuilder.java`) so that it will pick up your code.

In more detail, the signature of the class implementing the operator is:

```
/**
 * Constructs a new grace-hash join operator.
 *
 * @param left the left input operator.
 * @param right the right input operator.
 * @param sm the storage manager.
 * @param leftSlot pointer to the left sort attribute.
 * @param rightSlot pointer to the right sort attribute.
 * @param buffers the number of buffers to be used for the hash tables.
 * @param predicate the predicate evaluated by this join operator.
 * @throws EngineException thrown whenever the operator cannot be
 * properly constructed.
 */
public GraceHashJoin(Operator left,
                     Operator right,
                     StorageManager sm,
                     int leftSlot,
                     int rightSlot,
                     int buffers,
                     Predicate predicate)
    throws EngineException { ...
```

As was the case in merge join, the first three arguments will be set by the optimiser. The fourth and firth arguments are the two join attributes from the lest and right input respectively. The sixth argument is the number of buffers you have available for building the hash table for one of the inputs. And the last argument is the predicate being evaluated.

- Assume for the time being that the left input is the smaller one (i.e., the one you should have all of its hash tables fit into main memory).
- Scan the left input and store it in a temporary file, so you know how many pages there are in that file.
- Given the number of buffers you have available, pick the right number of partitions.
- 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 for the left input (one per partition).
- Scan the stored left input and partition it by applying a hash function on the value of the join slot to figure out what partition it belongs to.
- Generate as many partition files as needed for the right input (one per partition).
- Scan the right input and partition it by applying a hash function on the value of the join slot to figure out what partition it belongs to.
- Iterate over the number of partitions and for each partition:
  - Open the left partition file on disk, read it tuple by tuple and place its contents into an in-memory hash table based on the join key of the input.
  - Open the right partition file, ready it tuple-by-tuple and for its tuple extract the value of its join attribute and look it up in the hash table for the left partition.
  - If there are any matches, output the result out to disk in the designated output file. The output file has already been created for you as was the case in merge join.

One thing to note here is that the left join attribute may not be a candidate key, i.e., each value may appear more than one times in the input. So you will need to build a hash tble structure capable of storing multiple records per value.

Connecting your code to the rest of the system

The final step you should take: open `PlanBuilder.java` under `org/dejave/attica/engine/optimiser` of your source installation. Go to line `985` and uncomment all lines until line `1044`.

Then substitute the calls to `ExternalSort` with your own implementation of external sort—remember to change the name if you have named it differently (if you have not submitted the first assignment, a solution will be provided.). Then comment out lines `1045` and `1046`. That way, the next time a join is specified in a query, your code will be executed if it is an equality join. In all other cases the system will choose nested loops.

As was the case in the first assignment, to enable hash-based algorithms and, subsequently, Grace hash join, you need to issue the appropriate command on *attica*'s prompt:

```
aSQL> enable hash;
```

To disable hashing you will need to execute:

```
aSQL> disable hash;
```

Overall, the rules are that:

- By default (i.e., without any changes to `PlanBuilder.java`) nested loops join is the join processing algorithm.
- If you change `PlanBuilder.java` by following the instructions, you enable fast join processing algorithms for equality join predicates.
- Merge-join is the default algorithm and, if hash-based algorithms have been enabled through the command prompt, join processing defaults to using Grace hash join.

Testing your implementation

For that purpose you can use the same data generator as before, after generating multiple test tables. You can have key to key joins (e.g., joining on any of the `unique*` attributes) or joins with different key cardinalities. Have a look into `WBGen.java` to see the key cardinalities but your best bet for testing is a one-to-many join.

Your implementation will only be tested with a single join per query, so there is no need to try anything more complicated than that (not to mention that it might have chasing down bugs you are most likely not responsible for).

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

You will have to hand in the compiled version of your source tree, along with the source for your implementations of `MergeJoin.java` and `GraceHashJoin.java`. Please also submit any other files you have modified (apart from `PlanBuilder.java`).

The submission is electronic only and the deadline is

Friday, 13 March, 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 2 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.