CSE177/EECS277 – DATABASE SYSTEMS IMPLEMENTATION
Project 5: Join Operators
Due date: May 2 (in the lab)

This project requires the implementation of **three join operators: nested-loop join (NLJ), hash join (HJ), and symmetric hash join (SHJ)**. These algorithms have been discussed extensively in class and are also well-explained in the textbook. We implement only in-memory versions of these operators, i.e., we assume that there is enough memory to store both children of the join operator in in-memory data structures. With the join operator, it is possible to run all the SQL queries supported by our reduced query language, as long as each operator in the plan can be evaluated in memory. Essentially, the output of this phase is an in-memory database server.

Nested-Loop Join (NLJ)

Implement the `GetNext` method from the `Join` relational operator. You have to implement the nested-loops algorithm, with one of the relations fitting entirely in memory. There are two phases. In the *build* phase, one of the child relations – preferably the smaller – is read entirely in memory and stored in a list data structure, e.g., `TwoWayList`. In the *probe* phase, the other child relation is read one tuple-at-a-time. For every tuple, an iteration is executed over the stored list and the matching records are produced, one-at-a-time. This method is the most general form of join since it supports any join predicate—not only equi-join. In order to find matching records, method `Run` from `CNF` has to be invoked with the two records, one from each relation. Since join records are produced one-at-a-time, you have to be careful how you manage the iteration for every probing record.

Hash Join (HJ)

All the hash algorithms work only for equality conditions. In the case of the join operator, this means equi-join. The HJ algorithm follows closely NLJ. The main difference is that a multi-map is used instead of a list. For this purpose, both `EfficientMap` and `InefficientMap` work. In the *build* phase, one of the child relations – preferably the smaller – is read entirely in memory and stored in the multi-map, i.e., call `Insert` for every record. The key of the multi-map is the `Record`, while the data can be anything—you can use a dummy `SwapInt`, for example. In order to have `Insert` work properly, you have to set the `OrderMaker` in the `Record` before the operation. You have to create an `OrderMaker` for each child of the join. The `OrderMaker` contains the attributes that participate in join predicates. You have to identify them from the `CNF` condition. Method `GetSortOrders` from `CNF` does exactly that for you. Remember that you want to create the `OrderMaker` objects only once, when the `Join` operator is created, not in `GetNext`. In the *probe* phase, the other child relation is read one record-at-a-time. For every record, method `IsThere` is invoked on the multi-map—remember to set the `OrderMaker` in the `Record` first. In the case of a positive return value, you `Remove` the `Record` from the multi-map and store it in a list, e.g., `TwoWayList`. You repeat this process until no more matching records are found, i.e., `IsThere` returns `false` or `0`. Since join records are produced one-at-a-time, you have to be careful how you manage the iteration for every probing record. A join record is obtained by appending the two records that match, i.e., method `AppendRecords` from class `Record`. You iterate over the list and return a record each time. When the iteration is at the end of the list, you re-insert the records from the list into the multi-map and retrieve the next record from the child operator with `GetNext`.

Symmetric Hash Join (SHJ)

This operator is a non-blocking version of HJ. `GetNext` is called alternatively for the two children operators of the join. This can be done for a single record on each side, or for a few, e.g., 10 or 100. I suggest you do it

for 10 records. This means you get 10 records from the left child, then you get 10 from the right child, then from the left, and so on. In this case, you have two multi-maps—one for each child. Whenever you retrieve a `Record` from a child, you perform exactly the same procedure as in the probe phase of HJ. When you have finished returning all the joined records, you `Insert` the retrieved record into its corresponding multi-map. Then, you invoke `GetNext` for the corresponding child. You see a difference between HJ and SHJ in the order in which join records are produced and in the delay at which they are produced.

Implementation

The implementation is confined to the `GetNext` method in the `Join` operator—the only method remaining to implement. The `Join` operator works as follows. It has to determine which of the three algorithms to perform—NLJ, HJ, or SHJ? The decision to execute NLJ is simple: if there is any inequality condition in the join predicate, i.e., the `CNF`, then NLJ is the only alternative. For this, you have to inspect the `CNF` comparisons. If there are only equi-joins, you use HJ or SHJ. By default, you use HJ. When the number of records in both children of the join operator is larger than 1000, you use SHJ. Of course, you cannot know exactly how many records a child produces. However, you estimated this number in your query optimizer. You use this estimate to take the decision between HJ and SHJ.

Requirements

1. Implement the `GetNext` method for the `Join` operator to include the three join algorithms and the logic to choose between them. You may want to implement each join algorithm in a separate class with your choice of API. This allows a nice separation of the code.

2. Execute the queries provided with this stage of the project over the `TPC-H` data you generated and loaded in Phase 3 of the project.

3. For correctness and performance analysis, compare the results you obtain with the results generated by some other database server, e.g., `SQLite`.