CS448/648 Database Systems Implementation Assignment 2: Symmetric Hash Join

1 Objective

You are to implement a new query operator, symmetric hash join, to replace the current PostgreSQL hash join operator. Adding new operator will require modifications to both the query optimizer and the query executor in PostgreSQL .

2 The Basic Hash Join

A basic hash join requires one or more predicates of the form $T_1.attr_1 = T_2.attr_2$, where T_1 and T_2 are two relations to be joined and $attr_1$ and $attr_2$ are join attributes of the same data type. One of the relations is designated as the inner relation, while the other is designated as the outer relation. For the rest of this section, we will assume that T_1 is the inner relation and T_2 is the outer relation.

The basic hash join has two consecutive phases: the building phase and the probing phase.

1. **The building phase:** In this phase, the inner relation (T_1) is scanned. The hash value of each tuple is computed based on the join attribute ($attr_1$), using a hash function f, and is inserted into a hash table. No outputs are produced during this phase.
2. **The probing phase:** When the inner relation has been completely scanned and the hash table has been constructed, the outer relation T_2 is scanned and its tuples are matched to those from the inner relation. This matching is done in two steps: (1) Each tuple's join attribute $attr_2$ is hashed using the same hash function f and used to probe the hash table. (2) If matches are found, the actual values of the matches are compared to the actual value of $attr_2$. If the actual values match, then the corresponding tuples are joined and included in the join result.

The basic hash join is a blocking operation. That is, no outputs are produced until the tuples of the inner relation are completely processed and inserted into the inner hash table. This represents a bottleneck in a pipelined query plan.

3 Symmetric Hash Join

This section presents the symmetric hash join, which you will be implementing in this assignment. Unlike the the basic hash join, the symmetric hash join allows pipelined processing of tuples.

The symmetric hash join [WA91] differs from the basic hash join in that it does not wait until it acquires all of the tuples from one of its input relations before producing results. The Symmetric Hash Join operator maintains two hash tables, one for each input relation. It works by repeating the following procedure until the inner and outer input relations are both exhausted:

1. If the current tuple is null, retrieve a new tuple from either the inner relation or the outer relation. Normally, the symmetric hash join will alternate between the inner and outer relations when it retrieves a new current tuple until one of the two input relations is exhausted. From that point on, the current tuple is obtained from the input that is not exhausted until all tuples have been consumed from that input as well. When a new current tuple is obtained, it is inserted into the appropriate hash table, i.e., if it is an outer tuple it is inserted into the outer hash table, if it is an inner tuple it is inserted in to the innner hash table.

2. If the current tuple is obtained from the inner relation, use it to probe the outer hash table for matches. If the current tuple is obtained from the outer relation, use it to probe the inner hash table for matches. In either case, return any matches that are found.

Of course, in PostgreSQL , the symmetric hash join is implemented as an iterator, like all other query processing operators. This means that on each call to the operator it executes the procedure above until it has produced a single output tuple, which it can return to its caller. Before returning the result to its caller, the operator must save enough state information so that the next time it is called it can resume execution where it left off the last time. For example, the operator should remember the current tuple, whether that tuple came from the inner relation or the outer relation, and how far it had gotten in probing the appropriate hash table for matches. In PostgreSQL , such information is recorded in a structure called a *state node*. Each operator in a PostgreSQL query plan has a corresponding state node.

As an example of symmetric hash join execution, consider a join using the same join predicate, $T_1.attr_1 = T_2.attr_2$. The join operator will create a hash table for $attr_1$ and another hash table for $attr_2$ (we will call them H_1 and H_2, respectively). These hash tables, however, are not completely built in a separate blocking building phase as in the traditional hash join. Instead, they are expanded each time a new inner or outer tuple is retrieved. The symmetric hash join operator starts by getting a tuple t from T_1, hashing its $t.attr_1$ and inserting it into H_1. Then, it probes H_2 using $t.attr_1$. If matching tuples are found, it compares their actual $attr_2$ values, and returns any matching records. Similarly, it gets a tuple from T_2, inserts it into H_2, and probes H_1 with it. When no more matches can be found from the current inner and outer tuples, a new pair is retrieved. The previous steps are repeated until all tuples from T_1 and T_2 are consumed by the join operator.

Since the building and the probing phases are interleaved, the symmetric hash join is non-blocking. Results are produced as soon as there are available matching tuples from the input relations.

4 PostgreSQL Implementation of Hash Join

In this section, we present an introduction to two components of PostgreSQL that you will need to modify in this assignment, namely the optimizer and the executor. Then, we describe the current implementation of hash join in PostgreSQL .

4.1 Query Optimizer

The PostgreSQL query optimizer uses the output of the query parser to generate an optimal plan for the executor. During the optimization process, PostgreSQL builds `Path` trees representing the different ways of executing a query. It selects the cheapest Path and converts it into a `Plan` to pass to the executor. Each Path (and each Plan) is represented as a tree of nodes. There is a one-to-one correspondence between the nodes in the Plan and the nodes in its corresponding Path. Path nodes omit information that is not needed during planning, while Plan nodes omit planning information that is not needed by executor.

The optimizer builds a `RelOptInfo` structure for each base relation used in the query. `RelOptInfo` records information necessary for planning, such as the estimated number of tuples to be retrieved from that relation and their retrieval order. Base relations (`baserel`) are either primitive tables or subqueries that are planned via a separate recursive invocation of the planner. A `RelOptInfo` is also built for each join relation (`joinrel`) that is considered during planning. A `joinrel` is simply a combination of `baserels`. There is only one join `RelOptInfo` for any given set of `baserels` — for example, the join $\{A \bowtie B \bowtie C\}$ is represented by the same `RelOptInfo` no matter whether we build it by joining A and B first and then adding C, or joining B and C first and then adding A. These different means of building the `joinrel` are represented as different Paths. For each `RelOptInfo` we build a list of Paths that represent plausible ways to produce that relation. Once we have considered all the plausible Paths for producing a relation, we select the one that is cheapest according to the planner's cost estimates. The final plan is derived from the cheapest Path for the `RelOptInfo` that includes all the base relations of the query. A Path for a join relation is a tree structure, with the top Path node representing the join method. It has left and right subpaths that represent the scan or join methods used for producing the join's two input relations.

The join tree is constructed using a dynamic programming algorithm: in the first pass (already described) it considers ways to create `joinrel`s representing exactly two FROM items. The second pass considers ways to make `joinrel`s that represent exactly three FROM items; the next pass, four items, and so on. The last pass considers how to make the final join relation that includes all FROM items. For more details about construction of query Path and optimizer data structures, refer to `src/backend/optimizer/README`.

4.2 Query Executor

The executor processes a tree of `Plan` nodes. The plan tree is essentially a demand-pull pipeline of iterators. Each node, when called, will produce the next tuple of its output, or NULL if no more tuples are available. If the node is not a primitive relation-scanning node, it will have child node(s) that it calls recursively to obtain input tuples. The plan tree delivered by the planner contains a tree of `Plan` nodes (struct types derived from struct `Plan`). Each Plan node may have expression trees associated with it. These represent its target list and qualification conditions.

When the executor starts, it builds a parallel tree of *state nodes*. The state node tree has the same structure as the plan tree. Every plan and expression node type has a corresponding executor state node type. Each node in the state tree has a pointer to its corresponding node in the plan tree, in addition to executor state data that is needed to implement that node type. This arrangement allows the plan tree to be completely read-only as far as the executor is concerned; all data that is modified during execution is in the state tree. Read-only plan trees make life much simpler for plan caching and reuse.

Altogether there are four classes of nodes used in these trees: `Plan` nodes, their corresponding `PlanState` nodes, `Expr` nodes, and their corresponding `ExprState` nodes.

There are two main types of Plan node execution: *single tuple retrieval* and *multi-tuple retrieval*, which are implemented using the functions `ExecProcNode` and `MultiExecProcNode`, respectively. In single tuple retrieval, `ExecProcNode` is invoked each time a new tuple is needed. (`ExecProcNode` is the PostgreSQL name for the `GetNext` operation of the generic iterator that was discussed in class.) In multi-tuple retrieval, the function `MultiExecProcNode` is invoked only once to obtain all of the tuples, which are then saved the form of a hash table or a bitmap. For more details about executor structures, refer to `src/backend/executor/README`.

4.3 PostgreSQL Hash Join Operator

In PostgreSQL , hash join is implemented in the file `nodeHashjoin.c` and creation of a hash table is implemented in the file `nodeHash.c`. A hash join node in the query plan has two subplans that represents the outer and the inner relations to be joined. The inner subplan must be of type `HashNode`.

PostgreSQL implements an algorithm called *Hybrid Hash Join (HHJ)*. HHJ enhances the previously-described basic hash join to handle the case in which there is insufficient memory to hash the entire inner table (which the basic algorithm expects to be able to do). HHJ divides the tuples of each input relation into *batches*. For example, each batch can defined by a specific range of hash values. As the HHJ reads the tuples from the inner relation, it hashes only those tuples that belong to the first batch. Tuples belonging to other batches are spilled to temporary files on disk, with one file per batch. Similarly, as the HHJ reads the tuples from the outer relation, it probes the hash table only with tuples belonging to the first batch, spilling other tuples to another set of temporarly files. Once the first batch of tuples has been completely processed, HHJ processes the second batch. To do this, it clears the first-batch tuples from its hash table, reads inner second-batch tuples from a temporary file, and inserts them into the hash table. It then reads second-batch outer tuples from their temporary file and uses them to probe the hash table in the usual way. This process continues until all of the batches have been processed.

In this assignment, you will replace the HHJ in PostgreSQL from with symmetric hash join. To do this, you will want to start with the existing HHJ implementation and modify it. To make this task simpler, you may *assume that that both of the hash tables (inner and outer) needed by the symmetric hash join will fit in memory.* In other words, you are to assume that there is only one batch. As you modify HHJ, you will need to disable the handling of multiple batches in the current HHJ implementation so that your symmetric hash join will terminate after processing its single batch.

4.4 Important Files

These PostgreSQL source files are important for this assignment:

- In `src/backend/executor/`
  - `nodeHashJoin.c`: This file implements the hybrid hash join operator.
  - `nodeHash.c`: This file implements the hash operator, which builds a hash table. The existing hybrid hash join operator expects its inner input to be a hash operator.
- `src/backend/optimizer/plan/`
  - `createplan.c`: This file contains code the builds plans.
- `src/include/nodes/`
  - `execnodes.h`: Contains the structure `HashJoinState`, which maintains the state of the hash join during execution.

4.5 Important Functions

The implementation of the hash join operator in `nodeHashJoin.c` is divided into several functions. These are some ofq the most important ones:

- `ExecHashJoin` is the main function that is called each time a new tuple is required from the hash join node. Note that the first time this function is called it has to create the hash table or the inner node.
- `ExecInitHashJoin` is responsible for initializing the state of the join node as well as invoking initialization procedures for inner and outer input nodes.
- `ExecHashJoinOuterGetTuple` is invoked to get the next tuple from the outer node.

In addition, the following functions from `nodeHash.c` are particilarly relevant to this assignment:

- `MultiExecHash` retrieves *all* tuples from hash node's input and inserts them into a hash table.
- `ExecHash` is the function that would be called to retrieve the next tuple from a hash node. This function is not used and is not implemented in the current PostgreSQL code, because the hybrid hash join uses `MultiExecHash` instead. For this assignment, you will need to implement `ExecHash` so that the symmetric hash join can retrieve tuples one at a time from both its inner and outer inputs.
- `ExecHashGetBucketAndBatch` determines the hash bucket number (and the batch number) for a specific hash key.

5 Problem Statement

In this assignment, you are to implement the symmetric hash join to replace the existing hybrid hash join in PostgreSQL . The assignment can be broken into a subtasks as follows.

- Disable other non-hash join operators (i.e. merge join and nested loop join) to force the query optimizer to use the hash joins. This can be done by modifying the configuration file `postgresql.conf`, which can be found in the database directory (`pgdb`). This simplifies the testing of your code, since PostgreSQL will be forced to handle any equi-join using your hash join implementation.

  In the `postgresql.conf` file, you will find lines that look like this:

  ```
  #enable_mergejoin = on
  #enable_nestloop = on
  ```

These lines are simply comments indicating that the default value for these two parameters is "on". To disable merge join and nested loop join, you need to add two non-comment lines like this:

```
enable_mergejoin = off
enable_nestloop = off
```

- Change the optimizer so that *both* inputs for each hash join operator will be *hash nodes*. In the current implementation only the inner relation is hashed, as described in Section 2. This can be implemented by modifying the function that creates the hash join node in the planner, which can found in the file `createplan.c`.
- Modify the hash operator so that it supports pipelined execution (`ExecHash`) in addition to the blocking execution mode (`MultiExecHash`) that is currently implemented. This means that you will need to implement the `ExecHash` function, which, in the original code, is just a stub that returns an error. Your implementationn of `ExecHash` should obtain tuple from its input, insert the tuple in to the hash table, and return the tuple to its caller (the hash join).

  After completing these initial tasks, you will have a hybrid hash join that builds hash tables for both its inner and outer inputs. The inner hash table is built by the hash join's call to `MultiExecHash`. The outer hash table is built incrementally by calls to `ExecHash` as the hash join fetches tuples from its outer input. The outer hash table, though built, is not yet used, since you have not yet changed the actual hash join algorithm to implement the symmetric hash join. Nonetheless, you should be able to test what you have so far. In particular, you should confirm that PostgreSQL can still execute queries that involve hash joins, and you should confirm, using the PostgreSQL EXPLAIN utility, that hash join plans now include hash nodes for both their left and right inputs.
- Modify the structure of `HashJoinState` to support the symmetric hash join algorithm by altering the file `execnodes.h`. Since the symmetric hash join algorithm uses two hash tables, rather than one, you will need to record additional information to track the current state of the hash join.
- Replace the hash join algorithm in `nodeHashjoin.c` with the symmetric hash join. This is the most time consuming task. Remember that you are not required to handle multi-batch hash joins for this assignment. Thus, you are free to disable code in the hash join implementation that is intended to handle multiple batches.
- Your symmetric hash join implementation must produce two kinds of debugging information in the PostgreSQL server log. This information should be printed only when the server is started with debugging enabled, at debugging level 1 (i.e, when the server is started with the `-d 1` command line argument). This debugging information will be used for marking, so it is very important that your server produce it.

  Two kinds of debugging information are required: summary information and trace information. The summary information should be logged at the end of each hash join, and should include the following statistics about the hash join:

  - the total number of inner input tuples processed
  - the total number of outer input tuples processed
  - the total number of output tuples obtained by probing with inner inputs
  - the total number of output tuples obtained by probing with outer inputs

  Unlike the summary information, which is generated when the join is finished, the trace information is to be logged while the join is running. You server should log a trace record each time one of the following events occurs

  - the hash join obtains a tuple from the outer input - the trace record should identify which bucket of the inner hash table this tuple will be used to probe.

- the hash join obtains a tuple from the inner input - the trace record should identify which bucket of the outer hash table this tuple will be used to probe.
- the hash join matches a tuple from the outer input with a previously-hashed inner tuple, i.e., the hash join has identified a pair of input tuples that will produce an output tuple.
- the hash join matches a tuple from the inner input with a previously-hashed outer tuple, identifying a pair of input tuples that will produce an output tuple
- an inner tuple has been inserted into the inner hash table - the trace record should identify which hashtable bucket the tuple was inserted into
- an outer tuple has been inserted into the outer hash table - the trace record should identify which hashtable bucket the tuple was inserted into

Specific formatting guidelines for these trace log records will be published on the course web page. Your implementation must generate these records in the specified format so that they can be identified easily in the server's log.

6 Deliverables

The following files should be submitted : `nodeHashjoin.c`, `nodeHash.c`, `execnodes.h` and `createplan.c`. These are the only files that will be accepted. You must limit your code changes to these files, though you may need to read additional files to understand what PostgreSQL is doing.

You should submit your code using the `submit` command. To do this, create a submission directory and copy the files to be submitted into that directory. Then, from the submission directory, run the following command to submit your files

```
submit cs448 a2 .
```

References

[WA91] Annita N. Wilschut and Peter M. G. Apers, *Dataflow query execution in a parallel main-memory environment*, PDIS '91: Proceedings of the first international conference on Parallel and distributed information systems (Los Alamitos, CA, USA), IEEE Computer Society Press, 1991, pp. 68–77.