Term Project — Symmetric Hash Join[1]

1 Objective

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

2 Hash Join

In this section we introduce the regular hash join operator which is currently implemented in PostgreSQL 8.1.4. Hash join can be used when the query includes join 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 with the same data type. One of the relations is designated as the inner relation while the other is designated as the outer relation. We will assume that T_1 is the inner relation and T_2 is the outer relation.

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

Building Phase: In this phase, the inner relation (T_1) is scanned. The hash value of each tuple is computed by applying a hash function f to the join attribute ($attr_1$). The tuple is then inserted into a hash table. No outputs are produced during this phase.

The probing phase: When the inner relation has been completely processed 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 on two steps. First, the outer tuple's join attribute, $attr_2$, is hashed using the same hash function f. The resulting hash value is used to probe the inner hash table. Next, if the appropriate bucket of the inner hash table is not empty, the $attr_1$ value of each inner tuple in the bucket is compared to $attr_2$ of the outer tuple. If the actual values match, then the corresponding tuples are returned as one tuple in the join result. This comparison of the actual attribute values of the two tuples is necessary because of the possibility of hash collisions (multiple attribute values having the same hash value).

One property of this algorithm is that no join outputs are produced until *all* of the tuples of the inner relation are completely processed and inserted into the hash table. This represents a bottleneck in the query execution pipeline. Symmetric hash join, which you will implement in this assignment, avoids this problem.

One of the drawbacks of the hashing algorithm that was just described is that there must be sufficient memory available to hold the hash table for the inner relation. This can be a problem for large relations. For this reason, PostgreSQL actually implements a more complex algorithm, known as *Hybrid Hash Join*, which avoids this problem. The hybrid hash join algorithm divides the tuples of the two relations into multiple *partitions*, or batches. For example, each partition can be associated with a specific range of hash values. Only one partition of the inner relation's tuples is kept in main memory at any time. The remaining batches reside on disk. When an outer tuple is found to belong in a partition other than the current one (i.e. the memory-resident partition), it is written to a temporary file and its processing is postponed until the corresponding partition from the inner relation is retrieved. Partitions are processed in sequence until all partitions have been processed.

3 Symmetric Hash Join

Here we describe the symmetric hash join [1] that you are required to implement. The symmetric hash join operator maintains two hash tables, one for each relation. Each hash table uses a different hash function. It supports the traditional demand-pull pipeline interface. The symmetric hash join works as follows:

[1]Do not redistribute this project which is partof an archive that Ihab Ilyas gave to me for my personal use in my course.

- Read a tuple from the inner relation and insert it into the inner relation's hash table, using the inner relation's hash function. Then, use the new tuple to probe the outer relation's hash table for matches. To probe, use the outer relation's hash function.
- When probing with the inner tuple finds no more matches, read a tuple from the outer relation. Insert it into the outer relation's hash table using the outer relation's hash function. Then, use the outer tuple to probe the inner relation's hash table for matches, using the inner table's hash function.

These two steps are repeated until there are no more tuples to be read from either of the two input relations. That is, the algorithm alternates between getting an inner tuple and getting an outer tuple until one of the two input relations is exhausted, at which point the algorithm reads (and hashes and probes with) the remaining tuples from the other relation.

Note that the symmetric hash join operator must be implemented to adhere to the demand-pull pipeline operator interface. That is, on the first pull call, the algorithm should run until it can produce one join output tuple. On the next pull call, the algorithm *should pick up where it left off*, and should run until it can produce the second join output tuple, and so on. This incremental behaviour requires that some state be saved between successive calls to the operator. Essentially, this state records where the algorithm left off after the previous demand-pull call so that it can pick up from there on the next call. Clearly, this state should include an indication of whether the operator is currently working on an inner or an outer tuple, the current tuple itself, the current "probe position" within the inner or outer hash table, and so on. The symmetric hash join operator records this state into a structure called a *state node*.

As an example of join execution, consider a join with join predicate $T_1.attr_1 = T_2.attr_2$. The join operator will incrementally load a hash table H_1 for T_1 by hashing $attr_1$ using hash function f_1, and another hash table H_2 for T_2 by hashing $attr_2$ using hash function f_2. The symmetric hash join operator starts by getting a tuple from T_1, hashing its $attr_1$ field using f_1, and inserting it into H_1. Then, it probes H_2 by applying f_2 to $attr_1$ of the current T_1 tuple, returning any matched tuple pairs that it finds. Next, it gets a tuple from T_2, hashes it by applying f_2 to $attr_2$, and inserts it into H_2. Then, it probes H_1 by applying f_1 to $attr_2$ of the current T_2 tuple, and returns any matches. This continues until all tuples from T_1 and T_2 have been consumed.

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 hash join algorithm that is already implemented in PostgreSQL .

4.1 Optimizer

The optimizer uses the output of the query parser to generate an execution 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 that generates the desired outputs and converts it into a `Plan` to pass to the executor. A Path (or Plan) is represented as a set of nodes, arranged in a tree structure with a top-level node, and various sub-nodes as children. There is a one-to-one correspondence between the nodes in the Path and Plan trees. Path nodes omit information that is not needed during planning, while Plan nodes discard 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 and their 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 `baserel`'s. There is only one join `RelOptInfo` for any given set of `baserel`s. For example, the join $\{A \bowtie B \bowtie C\}$ is represented by the same `RelOptInfo` whether it is built by first joining A and B and then adding C, or by first joining B and C 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 implement the scan or join of that relation. Once we have considered

all of the plausible Paths for a relation, we select the cheapest one 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 the two input relations.

The join tree is constructed using a dynamic programming algorithm. In the first pass we consider ways to create `joinrel`s representing exactly two relations. The second pass considers ways to make `joinrel`s that represent exactly three relations. The next pass considers joins of four relations, and so on. The last pass considers how to make the final join relation that includes all of the relations involved in the query. For more details about the construction of query Path and optimizer data structures, refer to `src/backend/optimizer/README`.

4.2 Executor

The executor processes a tree of `Plan` nodes. The plan tree is essentially a demand-pull pipeline of tuple processing operations. Each node, when called, will produce the next tuple in its output sequence, 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 to represent, e.g., qualification conditions.

During executor startup, PostgreSQL builds a parallel tree of identical structure containing executor *state nodes*. 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 simplify plan caching and plan 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 types of Plan node execution: *single tuple retrieval* and *multi-tuple retrieval*. These are implemented using the functions `ExecInitNode` and `MultiExecProcNode`, respectively. In single tuple retrieval, `ExecInitNode` is invoked each time a new tuple is needed. In multi-tuple retrieval, the function `MultiExecProcNode` is invoked only once to obtain all of the tuples, which are returned in a 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 $HashNode$.

As was described in Section 2, PostgreSQL implements hybrid hash join so that it can deal with large relations. Recall that hybrid hash join processes tuples in batches based on their hash values. To make the implementation of the symmetric hash join algorithm simpler, you may ignore this additional complexity. Specifically, your symmetric hash join implementation may assume that both hash tables will fit memory without resorting to paritioning into multiple batches. However, you will need to find out how to disable the use of multiple batches in the current implementation of hashing.

4.4 Relevant Files

Here we present a list of files that are relevant to the assignment.

- `src/backend/executor/`
  - `nodeHashJoin.c`: This file implements the actual processing of the hash join operator.
  - `nodeHash.c`: This file is responsible for creating and maintaining a hash table.
- `src/backend/optimizer/plan/`

  - `createplan.c`: This file contains the code that creates a hash join node in the query plan.

- `src/include/nodes/`
  - `execnodes.h`: This file contains the structure *HashJoinState* that maintains the state of the hash join during execution.

4.5 Main Functions

The implementation of the hash join operator consists of a number of functions as follows:

- *ExecHashJoin*: This is the main function that is called each time a new tuple is required by the hash join node. Note that the first time this function is called, it has to create the hash table of the inner node. The join algorithm goes as described in Section 2.
- *ExecInitHashJoin*: This function is responsible for initializing the state of the join node as well as invoking initialization procedures for inner and outer nodes.
- *ExecHashJoinOuterGetTuple*: When a new outer tuple is required, this function is invoked to get the next tuple from the outer node. Note that after all tuples are retrieved from the outer relation, it is still necessary to retrieve the tuples that were previously saved to temporary files as soon as their correct batch of the inner hash table becomes available in the memory.
- *ExecHashJoinSaveTuple* and *ExecHashJoinGetSavedTuple*: These support saving and retrieving the outer tuples to/from temporary files. Saving a tuple of the outer relation is necessary when its batch number is not the same as the current memory resident batch of the hash table. Retrieving all saved tuples is done after consuming all tuples of the outer relation.
- *ExecHashJoinNewBatch* This function retrieves the next batch of the inner hash table.

To implement hash node, a number of functions are provided in `nodeHash.c`. The following functions are relevant to the assignment.

- *MultiExecHash*: This function retrieves *all* tuples from the subnode and insert them into the hash table. It returns the created hash table to the caller.
- *ExecHash*: Pipelined execution of hash is not implemented by default in PostgreSQL . This function will simply return an error message indicating that this execution mode is not supported for hash nodes.
- *ExecHashGetBucketAndBatch*: This function retrieves the 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 traditional hash join. You should alter the necessary files/functions to achieve all requirements. The assignment can be divided into subtasks as follows:

- Change the optimizer so that both the inner and outer relations are first processed by *hash nodes* before being processed by the *hash join* operator. In the current implementation, only the inner relation is hashed, as described in Section 2. The hashing nodes is necessary to *incrementally* retrieve a tuple from each relation, insert it into the corresponding hash table and return it to the hash join node. This effect can be implemented by modifying the function that creates the hash join node in the planner component, which can found in the file `createplan.c`. **(10%)**
- Modify the hashing algorithm to support pipelined execution instead of the multi-tuple execution mode that is currently implemented. This means that you will need to implement the *ExecHash* function, which currently returns an error message to indicate that this execution mode is not supported. **(20%)**

- For this assignment you can disable the use of multiple batches, i.e. you may assume that the whole hash table consists one batch that is resident in memory for the duration of the join processing. Modify the file `nodeHash.c` to achieve this. Note that related code in `nodeHashjoin.c` that handles saving and retrieving tuples to temporary batch files can now be discarded. **(0%)**
- Modify the structure of *HashJoinState* to support the symmetric hash join algorithm by altering the file `execnodes.h`. You need to replicate the existing structures to support bi-directional probing. **(10%)**
- Replace the hash join algorithm with the symmetric hash join. This algorithm should be implemented in `nodeHashjoin.c`. After completing the join operation, your operator should print the number of resulting tuples that were found by probing inner hash table and outer hash tables, respectively. **(60%)**
- Disable other join operators (i.e. merge join and nested loop join) to force the query optimizer to use the symmetric hash join operator. This can be done by modifying the configuration file `postgresql.conf` which can be found in the database directory. **(0%)**

You can test that the symmetric hash join is working properly by comparing the tuples resulting from an arbitrary join query before and after the modification. You can use the provided test case from the course web site to test your implementation.

6 Deliverables

The following files should be submitted : `nodeHashjoin.c`, `nodeHash.c`, `execnodes.h` and `createplan.c`. Although you do not have to change other files, you may need to explore additional files to understand how certain procedures and structures are implemented.

You can submit these files by sending them to the TA by email in a ZIPed file. To use the `zip` command, make sure that all the required files are in the current directory.

Make sure to clearly emphasize parts of the files that you have modified by inserting comments before and after changes. Also, include any necessary comments to describe how modifications are done or how you implemented new features. All comments should be preceded by `'CSI3530:'`.

References

[1] Annita N. Wilschut and Peter M. G. Apers, Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991), Miami Beach, Florida, December 4-6, 1991*, pp. 68–77.