CSCD43 DATABASE SYSTEM TECHNOLOGY

Assignment 2

Assigned Friday Feb 7

Due Monday March 10

Weight: 12% of your final grade

Hash Joins

1 Introduction

For the first assignment, you modified the storage layers of PostgreSQL , in particular the buffer manager. The focus of this assignment is on **query processing**. In order to implement a runtime optimization, you will have to modify **hash-joins** in PostgreSQL .
As in the previous assignment, you will be modifying PostgreSQL 7.4.13. The code for this assignment *does not* build upon the code of the previous one, and you will be modifying the source code as downloaded from the web.

2 Background

2.1 Hash-Join

PostgreSQL implements the **hybrid hash-join** algorithm. A short overview of this algorithm is provided below; for more details, you can to consult other resources.

During our discussion, we will assume that we are computing

$$R \bowtie_{R.A=S.B} S \tag{1}$$

- relation S will be referred as the **inner(build)** relation, and
- relation R will be referred as the **outer(probe)** relation.

The steps of the hash-join algorithm are presented below:

1. The first step consists of building the **hash table**. The hash algorithm reads in the inner relation and partitions it into buckets (according to the hash function). From these buckets, one is kept in memory and the rest of them are kept on the disk. We refer to the tuples of this build phase as build tuples.
2. The hash-join algorithm starts reading in data from its outer relation, and partitions it (in the same way it partitioned the inner relation). We refer to these tuples as probe tuples. For the tuples that match some tuples from the partition in memory, it performs a regular in-memory hash join of the partitions. The other tuples are written out to the disk in temp files, corresponding to their partitions. These are later read in later passes as the join is performed partitionwise.

Suppose now that there is a way to filter out tuples from the probe input which will not join with any build tuple. For such filtering to be useful, it should be performed during the first pass (when a tuple is read for the first time). If we detect that a (probe) tuple will not join with any (build) tuple, we can drop that tuple right then. By dropping that tuple, we will avoid the expensive operations of

- finding a partition for the tuple,
- writing it out to the temp-file,
- reading it back again and trying to join it with some other tuple.

Essentially, we want to be able to do membership tests on the join attribute of the probe input, in order to check whether there is a build tuple that might match with it. Convince yourselves that such a membership test

- may have **false-positive**s: a tuple that does not join with anything is let to pass;

- but it should not have **false-negatives**: a tuple that could join with another tuple is dropped (this should not happen).

For such a membership test to be effective, the false-positive rates have to be low.

The purpose of this assignment is to implement the optimization described above, by using *bloom filters*.

2.2 Bloom Filters

A **Bloom filter** is a probabilistic data structure that allows us to implement the membership test described in Section 2.1,

- in a space efficient manner, and
- satisfying the condition of having low false-positive rates, and no false-negatives.

Bloom filters are extremely useful in practice, and are used in a wide variety of application domains.

A brief description of the Bloom filters is provided below; however, you are encouraged to refer to other resources for a more detailed overview.

- A Bloom filter for representing a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements is described by an array of m bits, which are all initially set to 0.
- The filter uses k independent hash functions $h_1, \ldots, h_k$ within the range $\{1, \ldots, m\}$.
- For each element $x \in S$, we compute $h_1(x), \ldots, h_k(x)$, and set the corresponding positions in the bit array to 1.

  It is possible that multiple elements from S will map to the same position in the bit array. Once we have done this operation for each element of S, we should have a bit array that acts as an *approximate representation of the set*.
- To **check if** $y \in S$, we check whether for each of the k hash functions, the position $h_i(y)$ is set to 1 in the bit array. If at least one of these positions is set to 0, it is clear that $y \notin S$. If however, all the positions are set to 1, we know that y *may be* a member of S with some probability.

  ** (You may check the derivation of probabilities of error rates if you are interested).

In order to use Bloom filters to optimize the hash-join algorithm, we can consider the elements of the join attribute (of either relation) as the members of the set (S above); the details are indicated in the following:

1. First, build a Bloom filter on the elements of the join attribute in the build relation. This can be done while we are creating the build hash table.
2. With such a bit array created, we can check the join attribute of incoming probe tuples, to see if they *potentially* could join with some build tuple. If a tuple is hashed to a position at which the bit array is set to 0, we can instantly drop the tuple and not worry about it any more.

2.2.1 References

1. **Burton H. Bloom**, "Space/Time Trade-offs in Hash Coding with Allowable Errors". Commun. ACM 13(7): 422-426 (1970)
2. **Lecture Notes**: `http://www.cs.berkeley.edu/~daw/teaching/cs170-s03/Notes/lecture10.pdf`

3 Implementation

For this assignment, the following assumptions are allowed:

1. The join condition is an equality condition on one column of the build relation and one column of the probe relation, as indicated in Equation 1.
2. The join attributes are on the `integer` domain.

In a more general case, when the join attributes are on general domains, your code may not work (it can even crash) under these assumptions. However, this should not be a major issue if your evaluation is on tables containing only integers.

3.1 Starting Code

1. src/backend/executor/nodeHashjoin.c

   – This file contains the hash-join source code.

2. src/backend/executor/nodeHash.c

   – This file contains the hash operation source code.

3. src/include/nodes/execnodes.h
   src/include/nodes/plannodes.h

   – These header files contain the data structures which maintain the state of an operator. You may need to modify them.

4. src/include/executor/nodeHash.h
   src/include/executor/nodeHashjoin.h

   – These header files contain function definitions for the hash/hash-join operators.

5. src/backend/executor/execProcnode.c

   – This file contains the source code used by PostgreSQL to control the tuple flow across operators.

You should be able to complete this assignment by modifying only the files from (1) to (4).

Another file you might need as a reference is the following:

6. src/include/postgres.h

   – This file contains definitions of the PostgreSQL type system. Everything in PostgreSQL is of the type "Datum". From this data type it is possible to extract the usual `C`-representation of the data (in this assignment you will be extracting integers).

Note that this assignment does not build upon the first assignment. You need to change the code of regular unmodified PostgreSQL .

3.2 Implementation Steps

Roughly, the implementation involves:

- **Define a few** (up to 4 or 5 should be enough) **hash functions for the Bloom filter**. Designing a good hash function is very difficult though, and you are encouraged to look around and utilize hash functions that other people have defined. However, *your source code must be your own*. Include the reference from where you picked up your hash function in the report.

  These functions should be able to hash an integer to a number between 1 and m, where m is the size of the array.

- **Implement a bit array and add it to the structures associated with the hash/hash-join operators**.

  Note that you should not be using more than one bit of storage per position of the array. An implementation that tries to fake such an array by building an array of integers with one integer per "bit" will be considered as *unacceptable*, since this approach consumes much more space than a compact bit array.

  Recall that the `C` language does not let you to declare a bit array explicitly. However, you can declare an array of bytes (or chars), and then use bit-level operations to set the positions in the array.

- **Extract the values of the join attribute from the build and probe relations**.

  This is a non-trivial task, but the source code of the operators does such extraction, since it also hashes tuples to buckets. Check the code carefully, and you should be able to figure out how it works.

  (**Hint**: "hashkeys" encodes this information.)

- **Build a Bloom filter, transfer it to the hash-join operator, and use it** to quickly drop tuples that will not join with other tuples.

4 Evaluation

Your task is to evaluate the Bloom filters, by varying parameters like the size of the filter and the number of the hash functions used. The steps of accomplishing this task are presented in the following.

1. Create 2 database tables with 10,000 tuples each. The key attribute in each of these tables are of `integer` type.
2. Populate the two tables with tuples such that the key attributes have values between 1 and 10,000.
3. Generate a workload of join queries, where tables of the same size (i.e., number of tuples) are joined, and additional selection conditions on the key attributes are specified.

   An example of such a query is the following (we assume that both relations R and S have the key attribute ID):

   ```
   SELECT COUNT(*)
   FROM R, S
   WHERE R.ID = S.ID AND
         R.ID < 3000 AND
         S.ID > 1000;
   ```

   Suppose that R is the build relation and S is the probe relation. The selection condition on S lets all the tuples satisfying the condition $ID > 1000$ to pass through. However, all the tuples from R satisfying the condition $ID \geq 3000$ will not join with any tuple of S. The Bloom filter should be able to drop most of these tuples.
4. Before you start the actual evaluation,
   - first install an unmodified copy of PostgreSQL ,
   - start up `psql`,
   - and then run the command `vacuum analyze`. This command will update the statistics used by PostgreSQL to estimate the costs and result sizes of the operators.

   Note: you will probably need to run `vacuum analyze` using your copy of unmodified PostgreSQL since the modifications in your code (due to the assumptions made for the assignment) might cause the system to crash for general queries.
5. Evaluate various configurations of the Bloom filter (filter size, number of hash functions) with regard to your query workload, and using the false- positive rate as a metric. Plot a few graphs which illustrate "interesting" trends, and write a brief analysis describing the trends you notice.

   In particular, we would like to see the following graphs:
   - Keeping the size of the bloom filter fixed, vary the size of the inner input of the hash-join, by changing the selectivity condition on the inner, and plot the false-positive rates when 2, 3 and 4 hash functions are used.
   - Keeping the size of the inner input fixed, vary the size of the Bloom filter and plot the false-positive rates when 2, 3 and 4 hash functions are used.
   - Keeping the size of the inner, and the size of the Bloom filter fixed, vary the size of the outer (again, by changing selection conditions), and plot the false-positive rates when 2, 3 and 4 hash functions are used.

By default, PostgreSQL does not use the hash-join algorithm to evaluate the join operator.
To force PostgreSQL to use hash-joins, you have to set the `enable_nestloop` and `enable_mergejoin` flags off before evaluating the query. Check that the plans are hash-join plans by using the `explain` command in `psql`.

One issue here is that you don't really have any control over which relation is made the inner and which one is made the outer by the PostgreSQL optimizer. Make sure that for a graph, you keep the **same** relations as inners and outers as you vary other parameters. Since the optimizer may interchange the inner and outer relations as you change the selectivity, this will limit the range over which you may do your experiments. That is fine.

6. Based on the experimental evaluation described above ((1) - (5)), indicate a general strategy for filtering. Assuming that you were the system designer, how would you answer the following questions:

   - How many hash functions would you use?
   - What would be the size of your bit array?

   Your choices should be justified by the graphs you generated in the first part of the evaluation.

7. Note that the size of the bit array clearly depends on the size of the set it is trying to capture, so a better metric for your filter is the number of bits per tuple. For example, suppose you decide that 20 bits per tuple is enough. Then, if your inner relation is estimated to produce 1000 tuples (after selection conditions, etc.), then you would allocate 20,000 bits to your filter. If the estimate is 100 tuples, then you might just want to allocate 2000 bits.

   Describe the strategy you finally choose, and implement it in your code.

   Note: The `Plan` structure contains the optimizer size estimates, which maybe wrong. You should take this into account and see how wrong the estimates usually are. However, if you are seeing estimates that are way off, then you probably haven't run the `vacuum analyze` command as instructed.

Unlike the previous assignment, you don't need to run the code in `postgres` mode. Instead, you may choose to work with the `postmaster` and `psql` server and client. It might be useful to keep an unmodified copy of the source code at hand, in order to be able to run database create/update commands, without risking crashes (due to the assumptions made in your code).

5 Good Citizenship

Make sure you follow the instructions given in the installation guide so that PostgreSQL runs with minimal utilization of resources on `mathlab`. This involves limiting the buffer size, and the number of shared connections allowed by `postmaster`. Similarly, once you're done, make sure you turn off the `postmaster` server using the `pg_ctl` command. Also, check the current set of processes on the system to ensure that you do not have any zombie processes running and consuming resources.
Students who ignore these instructions and cause other users of `mathlab` *to starve* `:)` *run this risk of being penalized on the assignment*

6 Submission instructions

1. You have to submit **electronically** the following files:

   - Your `report`(`pdf` ONLY):
     – it should contain the graphs and analysis, a description of your strategy for using filters, and any other information you deem pertinent.
   - A `README` file:
     – it should contain your name, student number, what works and what doesn't work in your assignment.
   - `workload.sql`
     – it should contain the `SQL` queries that you used in your evaluation.
   - The `C` files:
     `nodeHashjoin.c, nodeHash.c, execnodes.h, plannodes.h, nodeHash.h, nodeHashjoin.h`.
     You will have to submit all these files even if you didn't have to modify all of them.

Good luck!