3 Partitioned Hash Join

In this assignment we are looking at the cache characteristics of the *Partitioned Hash Join* algorithm. The application provided in the exercise already performs a simple *Nested Loops Join* and the *Classical Hash Join* algorithms over two relations R and S. Each relation contains a set of tuples. A tuple is identified by an ID (join attribute) and has a certain payload. For the purpose of this exercise, the join function only counts the number of results. We do not need to return the joined relations.

```
typedef struct {
  long int id;
  long int padding[NPAD];
} tuple_t;
```

3.1 Hash Join

The classic hash join algorithm for an inner join of two relations has the following steps:

```
for each tuple s in S do
{ hash on join attributes s(b)
  place tuples in hash table based on hash values};
for each tuple r do
{ hash on join attributes r(a)
  if r hashes in a nonempty bucket of hash table for S
  then
    {if r hash key matches any s in bucket concatenate r and s
    place relation in Q}};
```

1. *Hash phase* : Creating a hash table for one of the two relations by applying a hash function to the join attribute of each row. Ideally we should create a hash table for the smaller relation. Thus, optimizing for creation time and memory size of the hash table.
2. *Join phase* : Scanning the larger relation and finding the relevant rows by looking in the hash table created before.

This algorithm is simple, but it requires that the smaller join relation fits into memory, which is sometimes not the case. Also the same hash function should be used for both phases. The complexity of the algorithm is O(m+n) since both relations are scanned once.

3.2 Partitioned Hash Join

Partitioned hash joins try to optimize the join execution by partitioning the problem into separate parts using hashing to decompose the join problem into several easier sub-problems. For the partitioned hash join algorithm different solutions have been proposed during time for example GRACE hash join, hybrid hash join and the hashed loops join [2]. The main memory variant of Grace Join [4], partitions the relations on a hash-number into H separate clusters, so that each fits the memory cache. Even if it performs better than normal bucket-chained hash join the performance depends on the size of S. If H is too large we have two problems:

- if H exceeds the number of TLB entries each memory entrance will become a TLB miss
- if H exceeds the number of available cache lines (L1 or L2) cache trashing occurs causing an explosion of cache misses

The algorithm we are going to implement in this exercise is based on the Radix-Cluster Algorithm [1] implemented in MONET DB which is an optimization of the GRACE algorithm. By dividing the relations into H clusters using multiple passes, this algorithm has a memory access pattern that reduces random access for high values of H.

3.3 Task

Download the application framework from the course website http://www.systems.ethz.ch/sites/default/files/file/dpmh_Fall2012/src-handout-02_tar.bz2.

- Compile the application (→ `make`) and run it.
- How does the *Classical Hash Join* algorithm behave? Hint: Vary relation size and number of buckets.
- Implement the *Partitioned Hash Join* algorithm:
  `partitioned_hash_join(R, S)` in `PartitionedHashJoin.c`.
- Improve the *partition phase* → multiple passes:
  `multipass_partitioned_hash_join(R, S)` in `PartitionedHashJoin.c`.
- Plot your results.
- Profile your application and explain your results.

3.4 Customizing the application

The application contains a few C macros that you can use to adjust the application to your needs, e.g., you probably soon want to turn of the *Nested Loops Join* algorithm because it will get very slow as you increase the size of the relations. To see the cost of partitioning you will need to increase the size of your relations significantly. Hint: Your relations should be a lot larger than your caches. You can also adjust the distribution of the tuple-IDs. How does your application behave when your tuple IDs follow a zipf distribution? Below the most important macros in the application are described:

- `PartitionedHashJoin.h`
  - ▷ Turn off *Nested Loops Join* → `//#define NESTED_LOOPS_JOIN`
  - ▷ Use larger relations for *Hash Join* →
    `#define NUM_RELATION_R 80000000`
  - ▷ `NUM_BUCKETS`: How does the *Classical Hash Join* perform for different bucket sizes?
- `Relations.h`
  - ▷ `USE_ZIPF` : use *zipf* instead of *random* distribution
    `ZIPF_PARAM` : small value results in small skew, 1 is large
    `ZIPF_PERCENTAGE` : size of the value array compared to number of tuples (in [0,1])
  - ▷ `NPAD` : tuple padding (default = 0)

Note: Since the data is loaded into main memory, you should try to choose a number of tuples for the relations according to the RAM size of your system. Use `sizeof(tuple_t)` to determine the size of a row and then calculate how many rows can fit into main memory and the various caches of your system.

Questions:

1. When does performance of the *Classical Hash Join* algorithm decrease?
2. What causes the performance decrease? Cache misses? TLB misses? → check with Oprofile.
3. Which performance problem of the *Classical Hash Join* does the *Partitioned Hash Join* address?
4. Does the *Partitioned Hash Join* introduce any new bottlenecks? Verify your assumption with Oprofile.
5. How does partitioning in multiple passes affect cache misses and TLB misses?
6. For each of your optimizations, what is the speedup to the previous version?

References

[1] P. Boncz. Monet : A Next-Generation DBMS Kernel for Query-Intensive Applications http://oai.cwi.nl/oai/asset/14832/14832A.pdf. *Technology*, 26:1, 1997.

[2] P. Mishra and M.H. Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.

[3] L.D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems (TODS)*, 11(3):239–264, 1986.

[4] A. Shatdal, C. Kant, and J.F. Naughton. Cache conscious algorithms for relational query processing. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 510–510. Citeseer, 1994.