Analysis of Multi-Join Benchmarking

ABSTRACT

1. INTRODUCTION

Performance of join operations is crucial for relational data analytics. For a long time, database community has tried to improve the performance of join operations. The two main approaches in join algorithm designs are - partition-based and no partition-based join implementations.

Remarkably, most of the prior work on join performance [1, 2] has focused on a single join, usually implemented with synthetic schema and data. The prior work is valuable in understanding the implications of various parameters on the performance of one join operation. However, real life decision support queries often involve multiple join operations (multi-join), typically executed one after another on a dataset organized as star-schema or snowflake-schema. The debate on partitioning-based *vs* no partitioning-based joins has not been extended to the settings of multi-joins.

An additional complexity associated with analyzing multi-joins is the potential re-partitioning operation between two joins. The re-partitioning is needed when the input to the partitioned join is either not partitioned at all or is partitioned on a different attribute than the join attribute.

Our goal is to devise a high performance hash-based multi-join implementation algorithm for a right deep pipeline. $\rightarrow$ *(Justify why we choose only hash-based implementations)* $\leftarrow$. One example of a right deep pipeline is shown in Figure 1.

An important aspect of the right-deep pipeline in Figure 1 is that the build side in all the join operations is always some dimension table. We will leverage this aspect of the right-deep pipeline in our algorithm design. $\rightarrow$ *(Think of extending the idea to left-deep or zig-zag trees.)* $\leftarrow$

Also note that Figure 1 is a *logical plan* produced by the optimizer. It gives information about join ordering. e.g. join between $Fact$ and Dim_1 relations should be performed first, followed by the join of the first join's result and Dim_2 relation. Figure 1 does not specify the type of join (e.g. hash-based or sort-merge) and/or the partitioning nature of the join such as radix-based partitioned join or no partition-based join.

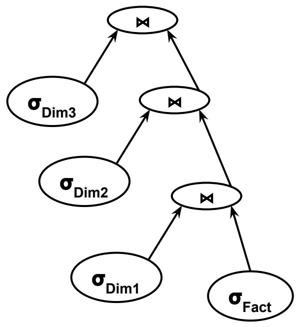


Figure 1: **A right-deep pipeline consisting of three join operations. Dim_i indicates i^{th} dimension table and $Fact$ denotes the fact table in a star-schema.**

The goal of our algorithm is to produce a high performance *physical plan* which consists of the partitioning nature of each join in the pipeline. We formally describe the problem in the next section.

1.1 Problem Statement

Given a right deep pipeline (as shown in Figure 1), consisting of k join operations viz. $J_1, J_2, \ldots J_k$, between fact and dimension tables, find the partitioning characteristic of the implementation of each join $p(J_i)$, which can result in overall high performance of the pipeline.

Our objective is to minimize the execution time of the multi-join pipeline. Let us define the execution time of join operation (probe + build phase) J_i as $T(J_i)$, or simply T_i. Therefore the objective is to minimize $\sum_{i=1}^{k} T_i$.

Note than if the execution exhibits pipelined parallelism, the above objective might be incorrect. In such a case, the total execution time for the pipeline is simply the time difference between end of execution of the last join and beginning of execution of the first join.

We consider only two options for the partitioning characteristic of the join implementation, i.e. $p(J_i)$:= PARTITION-BASED or $p(J_i)$:= NO PARTITION-BASED. Thus, our problem can be viewed as an assignment problem, where we have to assign a value to k binary variables. Therefore, the size of the solution space is 2^k, i.e. there are 2^k number of different possible physical plans.

In the next section, we discuss some proposals to solve the above problem.

1.2 Proposed Methodology

We first discuss the parameters involved in this problem. The execution time T_i per join operation J_i consists of both build and probe phases. If the join implementation is partitioning-based, then the time to partition (or re-partition)the relations is included in T_i.

We use a metric to differentiate the performances of partitioned and non-partitioned probes. This metric is the number of CPU cycles per probing tuple (C_{probe}), and it is impacted by the size of the probed hash table $Size_{HT}$. In the two implementations, viz. partitioned and non-partitioned the relationship between C_{probe} and $Size_{HT}$ is different. We study this relationship and estimate the cost of a given probe operation. In the next section, we present some initial empirical results to understand this relationship.

The partitioned implementation of a join operator may consist of partitioning cost. Therefore, we define another metric C_{part}, which is the number of CPU cycles spent in partitioning a single input tuple.

Finally, we estimate the cost of each join operations in terms of CPU cycles per input tuple.

We may use a well known technique such as dynamic programming to solve the assignment problem.

Note that we restrict ourselves to one form of materialization of the results i.e. eager materialization. This means that each join operation's result is immediately materialized as soon as the operation is complete. (An alternative could be late materialization, as is used by MonetDB.) $\rightarrow$ *(Unanswered question: Do assignment change as the query is in progress? e.g. If there are skewed results of the lower join operations in the query plan.)* $\leftarrow$

1.2.1 Heuristics

As described earlier, the total number of possible physical plans is exponential in the number of join operations. It may be prohibitively expensive to analyze all possible plans. Therefore, we can prune the number of plans being analyzed, by applying some heuristics. Examples of such heuristic could be: flip-flop between partitioned and non-partitioned joins is not desirable for successive join operations, prefer partitioned joins for lower level join operations in a query plan.

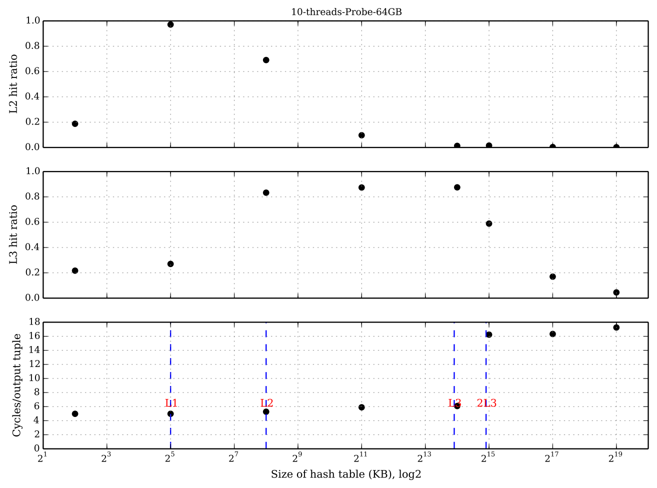
2. EXPERIMENTAL EVALUATION

In this section, we report the initial findings of a benchmarking study we conducted. The goal of the benchmark study was to understand the relationship of the metric number of CPU cycles per probe input tuple (C_{probe}) with size of the probed hash table ($SizeHT$).

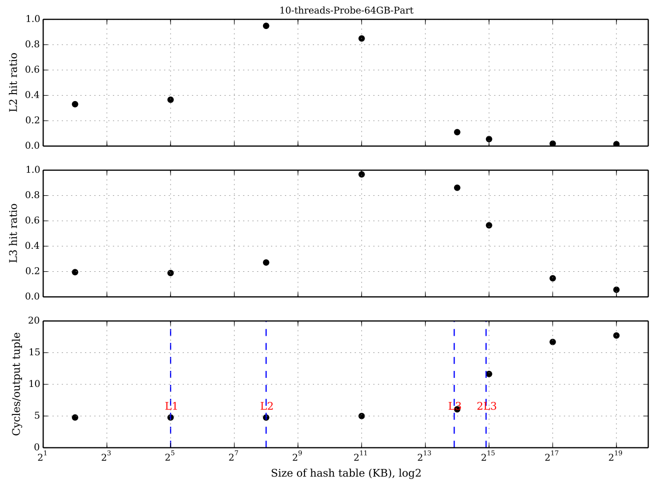
$\rightarrow$ *(Machine description.)* $\leftarrow$

We developed a C++ prototype that mimics a probe operation. We implemented the probe operation in both partitioned and non-partitioned ways. Note that the measurement does not include time to partition the probe table. We use Intel PCM for measuring the cache hit ratios and rdtsc counters for measuring the CPU cycles.

Figure 2 presents the results from a probe operation using 10 threads and an input probe table of size 64 GB, and Figure 3 refers to the results of a probe operation that uses 10 threads and a 32 GB input probe table. On the X-axis is the size of hash table in KB (log scale). The three scatter plots indicate the CPU cycles per input tuple, L2 hit ratio and L3 hit ratio. To put the cache sizes in perspective, the



(a) Non Partitioned Probe



(b) Partitioned Probe

Figure 2: **Results for 10 threads and 64 GB probe input**

blue dotted lines mark the sizes of L1, L2, and L3 caches. As the server machine has two NUMA sockets, we also added another line to mark 2*L3 size.

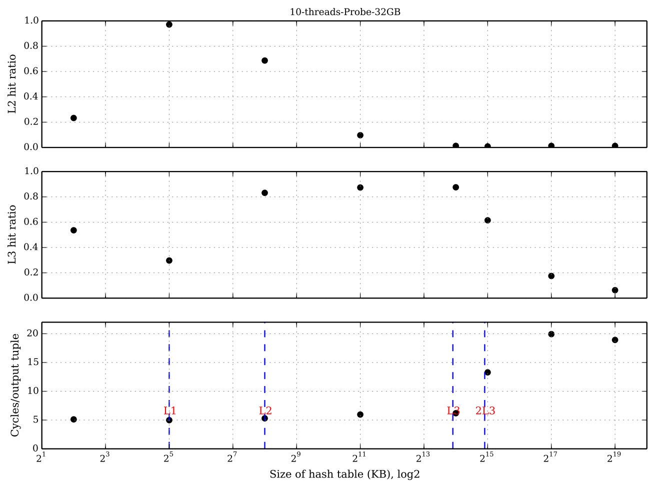
Note the increase in CPU cycles per probe operation after the hash table sizes greater than L3.

Note that the experiments using 10 threads don't make use of hyper threading. We perform the same experiment using all 20 threads on a single socket, which also includes the hyper-threading contexts. Figure 4 presents the results of this experiment. Notice that on sizes greater than 2*L3, the CPU cycles per input tuple is drastically higher (around 40) than that in the 20 threads case. However the impact of hyper threading is not as drastic for hash table sizes smaller than the size of the L3. cache

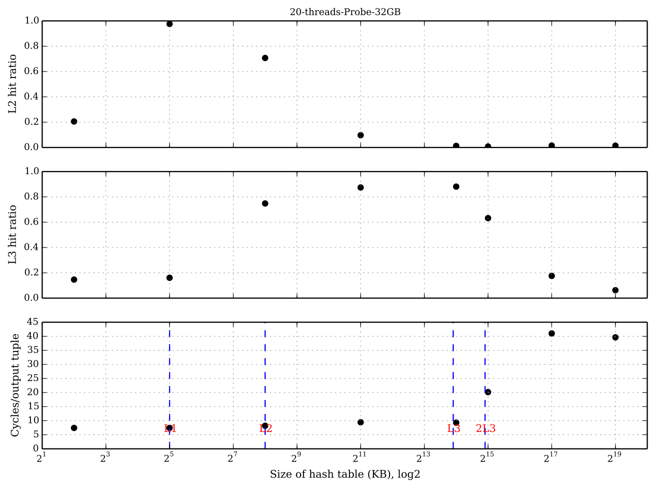
3. CONCLUSIONS

Based on this preliminary analysis, we can draw some conclusions which are described below:

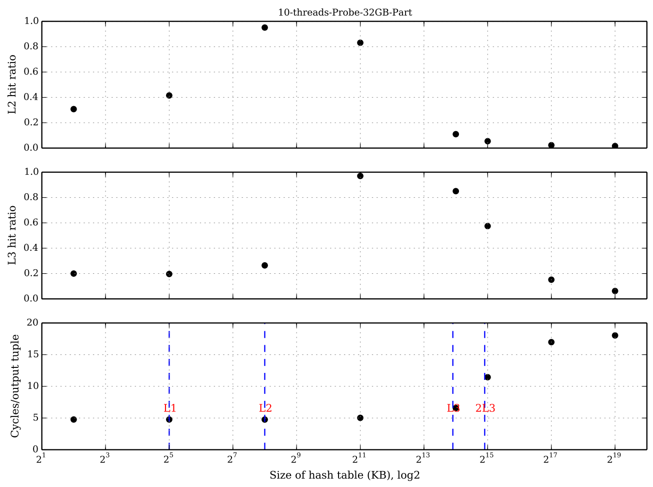
1. Larger hash tables (size greater than the size of the L3 cache) are not desirable for probe performance.



(a) Non Partitioned Probe

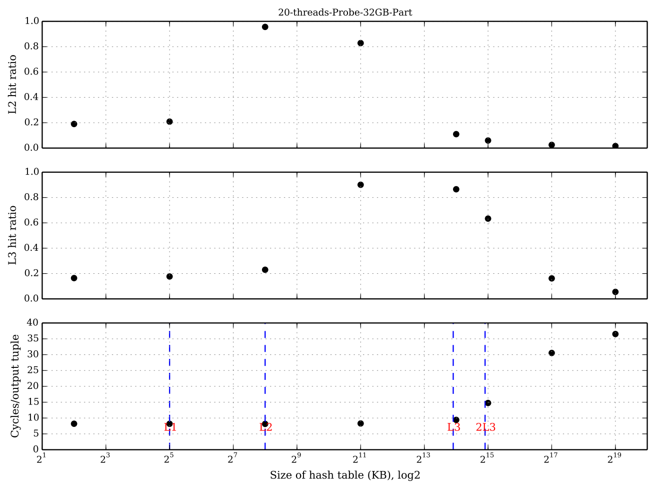


(a) Non Partitioned Probe



(b) Partitioned Probe

Figure 3: **Results for 10 threads and 32 GB probe input**



(b) Partitioned Probe

Figure 4: **Results for 20 threads and 32 GB probe input**

2. The CPU cycles per input tuple remains fairly stable for a large spectrum of hash table sizes i.e. Hash table sizes larger than L1 and smaller than L3. An implication of this observation could be that less efforts could be spent in tuning the partition size; as long as the partition size is within the range [L1, L3], the performance will be similar.

3. Hyper threading adversely affects the probe performance, however the impact is higher for larger hash tables (size greater than 2*L3)

4. RELATED WORK

The database community has paid significant attention to improve join performance in the past decade. Blanas et al. [1] analyzed the join performance in single socket, large main-memory server environments using various join implementations such as radix-based partitioning, no-partitioned join and using uniform as well as skewed data sets.

Schuh et al. [2] reviewed various join implementations from the research literature from the past decade. They classified the join algorithms in two categories - partitioning based and no-partitioning based join algorithms, and compared their performance characteristics. As they observed, prior work on join performance has mostly focused on synthetic queries and synthetic datasets. In the interest of discussing the join algorithm performance in the context of real life queries, they measured the performance of various join algorithms on a real query (TPC-H Query 19).

5. REFERENCES

[1] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48, 2011.

[2] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*, pages 1961–1976, 2016.