How efficient is our radix join implementation?

Spyros Blanas Jignesh M. Patel

Computer Sciences Department
University of Wisconsin-Madison, USA

Friday, August 5, 2011

We recently published a paper [2] that examines the design choices available to create a high-performance main-memory hash join algorithm. We experimentally evaluated four hash join variants on two different architectures, and we showed that an algorithm that does not do any partitioning on the input tables often outperforms the other more complex partitioning-based join alternatives. Our claim is that in an environment with a single processor and multiple cores, the non-partitioning method has many advantages over the more complex methods that have been proposed before. If the memory access latency between different processors is non-uniform, partitioning will be more beneficial; the non-partitioning method could then be used as a building block for an efficient hash join algorithm for data that has been partitioned to each processor (NUMA node). A full exploration of hash join methods for NUMA environments is part of our future work.

In our paper [2], one of the algorithms that we evaluated this simple hash join algorithm against was the radix-partitioned hash join [3]. We implemented the parallel radix-partitioned hash join algorithm that is described in [4]. During the conference, there were some questions regarding how efficient our radix join implementation is. Unfortunately, because of three differences in the experimental setup, a direct comparison between [2] and [4] is impossible. First, the tuples in [2] are 16 bytes wide, but only 8 bytes in [4]. Second, most of the experiments in [2] show results where the size ratio between two the join inputs is $\frac{1}{16}$, to mimic a primary key–foreign key join in a decision support environment where there is a small dimension table and larger fact table. In [4], the two inputs have equal size. Finally, although both papers use CPUs based on the Intel Nehalem microarchitecture, the hardware is not the same. Among other differences, the available memory bandwidth per core is the most striking: In the Xeon X5650 CPU used in [2], twelve hardware threads share the 6.4GT/s memory bus. In the i7-965 that is used in [4], only eight hardware threads share the memory bus of the same bandwidth. Although the exact effect of reduced memory bandwidth per core on parallel radix hash join performance is unknown, Intel reports that the i7-965 has 1.2X the effective bandwidth per core than the Xeon X5650 [1].

Although the implementation used in [4] was unavailable, the co-authors of the Kim et al. [4] paper at Oracle generously provided us with a different implementation of the parallel main-memory radix-partitioned join. We compared the two hash join implementations, and this report summarizes our findings. In Section 1, we compare the efficiency of the radix

Implementation	Partitions					
	64	256	1K	**4K**	16K	64K
Wisconsin–original	21.10	13.15	13.19	**14.10**	16.83	27.10
Wisconsin–hardcoded	7.59	6.80	7.77	**9.86**	13.29	25.49
Oracle	5.32	6.49	9.91	**10.68**	12.19	18.24

Table 1: Average time taken to partition a 256M tuple input table, in machine cycles per tuple, where each tuple is 8 bytes wide. We have configured all implementations to partition in one pass, and we highlight the column corresponding to the "radix-best" configuration from [2].

partitioning phase of the two implementations. In Section 2, we show where the time is spent in our radix-partitioned hash join implementation.

All the experiments are run on an Intel Xeon X5650, clocked at 2.67GHz. This CPU has 6 cores, is two-way multi-threaded, and has 12MB of shared L3 cache. The OS is Scientific Linux SL release 5.3, and we run the 64-bit 2.6.18 kernel, with Redhat 128.1.1.el5 patches applied. (This is the same machine and CPU that we used in [2].) We start twelve threads in all the experiments described here, and we run with HyperThreading on.

1 Radix partitioning efficiency

In this section we focus on the efficiency of the partitioning phase. We use the term "Wisconsin–original" to refer to the implementation used in [2], and the term "Oracle" to refer to the implementation from Oracle.

By inspecting the Wisconsin and Oracle implementations, we noticed that the Oracle implementation differs in the handling of index metadata: it assumes that that the tuple is eight bytes, the key is four bytes, and that the key is the first element stored in the tuple (that is, the key is stored at offset 0 from the beginning of the tuple). Furthermore, hashing is always a binary AND on the four byte key, followed by a shift. In comparison, the Wisconsin implementation has separate classes to store index metadata, and the partitioning code calls appropriate class methods for operations like attribute retrieval, value hashing and tuple copying. To facilitate direct comparisons, we hacked the Wisconsin implementation to perform key retrieval, key hashing and tuple copying inline, using the same values appearing in the Oracle implementation. We use "Wisconsin–hardcoded" to refer to this modification.

For this experiment, we partition a single input table that contains 2^{28} (256M) tuples. Mimicking the experimental setup of [4], each tuple has a four byte key, followed by four bytes of data, for a total of eight bytes per tuple. We report averages over ten runs, where each run partitions random input that is read directly from `/dev/urandom`. We have configured all implementations to always partition in one pass after we experimentally verified that one-pass partitioning is the optimal setting for this range of output partitions, for all implementations.

Table 1 shows the average time to partition, in machine cycles per tuple, for each implementation, for a different number of partitions. We highlight the column that corresponds to the "radix-best" setting we used in [2] for a dataset of the same size.

From this experiment, we conclude that the Oracle and Wisconsin–hardcoded implementa-

Phase	Action	Cycles per tuple per core: As measured in [2]	Hardcoding data operations	As reported in [4]
Partition	Data staging	32.72	—	90–95
	Core partitioning	131.75	97.43	
Build	Project on build tuple	3.87	—	35–40
	Hash table insert	64.26	64.26	
Probe	Hash table cursor read	47.10	47.10	
	Assemble output tuple	58.35	—	
	Total	338.05	208.79	$\sim$ 130

Table 2: Cycle breakdown during radix partitioned hash join.

tions are equally optimized, as they have similar performance between 256 and 16,384 partitions, after we account for experimental noise. Furthermore, we conclude that the partitioning numbers published in [2] could be improved by an average of 4-5 cycles per tuple, if we hardcoded the data access operations for the particular dataset.

2 Overall join efficiency

In this section we turn our attention to the overall join efficiency of our implementation [2], when compared with the previously published numbers in [4].

As the Intel implementation is unavailable, one question is how do we scale the numbers published in 2009 [4] on hardware that became publicly available in 2010. The main problem is that both CPUs have the same memory bus, but the Xeon CPU has 1.5X more cores. If the entire radix hash join algorithm was memory bound, the bottleneck would be on the memory bus, and we would expect to see no performance improvement from adding more cores. On the other hand, if the entire radix hash join algorithm was computation bound (and 1.5X more cores didn't saturate the memory bus), we would expect to see up to 1.5X improvement from adding 1.5X cores. Unfortunately, the radix hash join exhibits both behaviors, as some steps are memory bound and some others are computation bound [4].

Although predicting actual performance is hard, we can estimate what the upper bound on performance would be. Let's disregard the different memory bandwidth per core available, and let's optimistically assume that performance scales linearly with the number of cores. We therefore assume that the (unavailable) Intel implementation would be 1.5X faster when upgrading from the 4-core i7 used in [4] to the 6-core Xeon used in [2]. Looking at the figures of [4] closely, we see that for 8 byte tuples where the input tables have equal size the join finishes at 30-35 cycles per tuple. As the i7 CPU has four cores, this becomes 120-140 cycles per tuple per core. We therefore assume that, if the Intel implementation ran on the Xeon X5650 CPU, the upper bound on performance would be 130 cycles per tuple per core.

We run an experiment where the ratio between the two join inputs is 1:1. Each input table has 2^{24} (16M) tuples, and all tuples are 8-byte wide. Results are shown in Table 2, expressed in cycles per tuple per core. The left column corresponds to the experiment shown in Fig. 6 in [2], if ran on a dataset that has 8-byte tuples. The right column discounts the overhead of staging the join input data contiguously in memory (a prerequisite of the radix partitioning implementation that no storage engine that we know of is capable of producing), as well as the overhead of supporting generic, user-defined data operations like typed value comparisons, user-defined key hashing, and arbitrary projection expressions. We believe that the numbers published in [4] do not account for these overheads.

As the results show, hardcoding the data operations can save more than 100 cycles per tuple per core. Similar optimizations could be applied to the hash table code, however we have not endeavored to tune it further, as this is an orthogonal optimization that will improve performance across all algorithms we compare against. Finally, we make no effort to exploit data-level parallelism through SIMD optimizations – these two factors could close the 75 cycle per tuple per core performance gap between the two implementations even further.

In conclusion, there are some implementation differences between the two code bases, but at the level of the common core components, the differences are small. The code that we used in [2] is part of a much larger data processing engine, so it is by necessity more generic (for example, we can handle different schemas, and we support data types other than integers), although it is by no means complete (for example, we are missing code to evaluate arbitrary expressions). This report demonstrates that optimizations to selected portions of a hash join algorithm, and a performance comparison based on these portions, only shed light on part of the picture. The results should be put in the context of the entire machinery that is often needed in a real data processing engine to gauge the overall impact – in other words we shouldn't forget Amdahl's law. (We are making a note of this to ourselves too.)

Acknowledgment

We would like to thank Eric Sedlar, Tarik Ono, and Jatin Chhugani for providing us with their implementation and documentation discussing the observed performance characteristics.

References

[1] "Compare Intel(r) Products" website. `http://ark.intel.com/compare/47922,37149`. [Online reference; accessed July 15, 2011].

[2] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD Conference*, pages 37–48, 2011.

[3] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[4] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.