Lecture #09: Hash Join Algorithms

15-721 Advanced Database Systems (Spring 2024)
https://15721.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Prof. Andy Pavlo

1 Background

The **Parallel Join Algorithms** perform a join between two relations on multiple threads simultaneously to speed up operations. In this lecture, the discussion focuses on two-way (two relations) **Hash Join** operator design in **OLAP** system and tries to compare their performance along with [5], which has given an overall benchmark of different join algorithms. Notice that we have the following assumptions:

1. The tuples to join can fit into memory (compared to how pages are spilled to disk in intro class)
2. Hash join are better than sort-merge join in most cases (thus hash join will be the focus today)

Join Algorithm Design Goals

- **Minimize Synchronization**:
  1. Avoid taking latches during execution.
- **Minimize Memory Access Cost**:
  1. Ensure that data is always local to the worker thread.
  2. Reuse data while it exists in CPU cache: For **Non-Random Access (Scan)**, clustering data to a cache line and executing more operations per cache line. For **Random Access (Lookups)**, partition data to fit in cache + TLB.

2 Parallel Hash Joins

Hash join is the most important operator in a DBMS for OLAP workloads. We must speed up our DBMS's join algorithm by taking advantage of multiple cores. We want to keep all cores busy, without becoming memory-bound.

Hash Join ($\mathbf{R} \bowtie \mathbf{S}$)

The Hash Join can be divided into three phases[5]:

- **Phase #1: Partition (optional)**: Divide the tuples of R and S into disjoint sets using a hash on the join key.
- **Phase #2: Build**: Scan relation R and create a single logical hash table (it could be physically stored in different places) on the join key.
- **Phase #3: Probe**: For each tuple in S, look up its join key in the hash table for R. If a match is found, output the combined tuple. Notice that the overhead to combine and materialize these tuples is not negligible while some papers did ignore this in their experiment setups, pointed out by [5].

Partition Phase

Split the input relations into partitioned buffers by hashing the tuples' join key(s). Ideally, the cost of partitioning is less than the cost of cache misses during the build phase. Sometimes called hybrid hash join/radix hash join.

There are two general approaches to paritioning:

- **Non-Blocking Partitioning**: Only scan the input relation once. Produce output incrementally on the fly.
  - **Approach #1: Shared Partitions**: Single global set of partitions that all threads update. Must use a latch to synchronize threads.
  - **Approach #2: Private Partitions**: Each thread has its own set of partitions. Must consolidate them after all threads finish. (late materialization can benefit here to reduce data exchange between cores)
- **Blocking Partitioning (Radix)**: Scan the input relation multiple times:
  - **Step #1**: Scan R and compute a histogram of the number of tuples per **hash key range** for the radix at some offset.
  - **Step #2** Use this histogram to determine output offsets by computing the prefix sum.
  - **Step #3**: Scan R again and partition them according to the hash key.

  Only materialize results all at once. Sometimes called radix hash join.

Build Phase

The threads are then to scan either the tuples (or partitions) of R. For each tuple, hash the join key attribute for that tuple and add it to the appropriate bucket in the hash table. The buckets should only be a few cache lines in size.

There are two design decisions for a hash table:

- **Hash Function**: How to map a large key space into a smaller domain. The trade-off between being fast vs. collision rate. A Github repo SMhasher addresses on this.
- **Hashing Scheme**: How to handle key collisions after hashing. The trade-off between allocating a large hash table vs. additional instructions to find/insert keys.

  Some common approaches include:
  - **Chained Hashing**

    Maintain a linked list of buckets for each slot in the hash table. Resolve collisions by placing all elements with the same hash key into the same bucket. To determine whether an element is present, hash to its bucket and scan for it. (HyPer uses the remaining 16 bits of 48 bits out of 64-bit address space as bloom filter to accelerate this) Insertions and deletions are generalizations of lookups.
  - **Linear Probe Hashing**

    Single giant table of slots. Resolve collisions by linearly searching for the next free slot in the table. To determine whether an element is present, hash to a location in the table and scan for it. Must store the key in the table to know when to stop scanning. Insertions and deletions are generalizations of lookups. This scheme could suffer from hash results clustered together and lead to long jumps while lookup, and have to set slots to 2x the total number of keys by observation.
  - **Robin Hood Hashing**

    Variant of linear probe hashing that steals slots from "rich" keys and gives them to "poor" keys. [2] Each key tracks the number of positions they are from where its optimal position is in the table. On insert, a key takes the slot of another key if the first key is farther away from its

optimal position than the second key. This scheme does not guarantee the boundary of the jump.
  - **Hopscotch Hashing**
    Variant of linear probe hashing where keys can move between positions in a neighborhood [3]. A neighborhood is a contiguous range of slots in the table. The size of a neighborhood is a configurable constant. A key is guaranteed to be in its neighborhood or not exist in the table.
  - **Cuckoo Hashing**
    Use multiple tables with different hash functions. On insert, check every table and pick anyone that has a free slot. If no table has a free slot, evict the element from one of them and then re-hash it to find a new location. Look-ups are always O(1) because only one location per hash table is checked.
    Threads have to make sure that they don't get stuck in an infinite loop when moving keys. If we find a cycle, then we can rebuild the entire hash tables with new hash functions. With two hash functions, we (probably) won't need to rebuild the table until it is at about 50% full. With three hash functions, we (probably) won't need to rebuild the table until it is at about 90% full.

There are choices of what to put in the hash table:

- **Key**
  - **Join key only**: Have to do string (in case the join key is string) comparison
  - **Join key and Hash**: Some storage overhead but faster comparison
- **Value**
  - **Actual tuple data**: If hash scheme is open address, only fixed length data can be stored
  - **Pointers/offsets**: Materialize later while probing

Probe Phase

For each tuple in S, hash its join key and check to see whether there is a match for each tuple in corresponding bucket in the hash table constructed for R. If inputs were partitioned, then assign each thread a unique partition. Otherwise, synchronize their access to the cursor on S.

Bloom Filter: Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table [4]. Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches. Sometimes called sideways information passing.

3 Benchmarks Results

In a real DBMS, the optimizer will select what it thinks are good values based on what it knows about the data (and maybe hardware).

Partitioned-based joins outperform no-partitioning algorithms in most settings [1], but it is non-trivial to tune it correctly. Every DBMS vendor picks one hash join implementation and does not try to be adaptive.

References

[1] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 37–48, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306614. doi: 10.1145/1989323.1989328. URL https://doi.org/10.1145/1989323.1989328.

[2] P. Celis, P. Larson, and J. I. Munro. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 281–288, 1985.

[3] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In G. Taubenfeld, editor, *Distributed Computing*, pages 350–364, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87779-0.

[4] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1231–1242, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320375. doi: 10.1145/2463676.2465292. URL https://doi.org/10.1145/2463676.2465292.

[5] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1961–1976, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2882917. URL https://doi.org/10.1145/2882903.2882917.