

Parallel Join Algorithms

Two main parallel join algorithms:

- Hash Join
- Sort-Merge Join

许多OLTP DBMS是不会实现hash join的，因为没有必要build hashtable，有现成的index

index nested-loop join with small number of target tuples 可以大概等效于一个hash join

大概意思是说，hash join的hashtable是query查询临时构建的，查询完毕之后hashtable就会销毁。而index是本身就存在了的。

Join Algorithm Design Goals

- 减少同步开销

在执行过程中尽量避免用latches

- 最小化内存访问的开销

当data还在cpu cache的时候，尽最大可能重用

确保数据和工作线程的局部性

影响DBMS中cache misses的因素

- Cache + TLB容量(TLB就是快表，存储virtual address到physical address的mapping)
- 时空局部性

Non-Random Access:(sequential scan)

- Clustering data to a cache line.
- 那么可以对cache line的数据执行尽可能多的操作

Random Access: (lookups)

- partition data to fit in cache + TLB

什么是Hash Join?

可以划分为三个阶段

Phase #1: Partition (*optional*)

→ Divide the tuples of **R** and **S** into sets using a hash on the join key.

Phase #2: Build

→ Scan relation **R** and create a hash table on join key.

Phase #3: Probe

→ For each tuple in **S**, look up its join key in hash table for **R**. If a match is found, output combined tuple.

Partition Phase

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 build phase.
- Sometimes called *hybrid hash join* / *radix hash join*.

Contents of buffers depends on storage model:

- **NSM**: Usually the entire tuple.
- **DSM**: Only the columns needed for the join + offset.

如果是行存：整个tuple

如果是列存：只需要存储部分列的数据

PARTITION PHASE

Approach #1: Non-Blocking Partitioning

- Only scan the input relation once.
- Produce output incrementally.

Approach #2: Blocking Partitioning (Radix)

- Scan the input relation multiple times.
- Only materialize results all at once.
- Sometimes called *radix hash join*.

- Non-Blocking Partitioning

NON-BLOCKING PARTITIONING

Scan the input relation only once and generate the output on-the-fly.

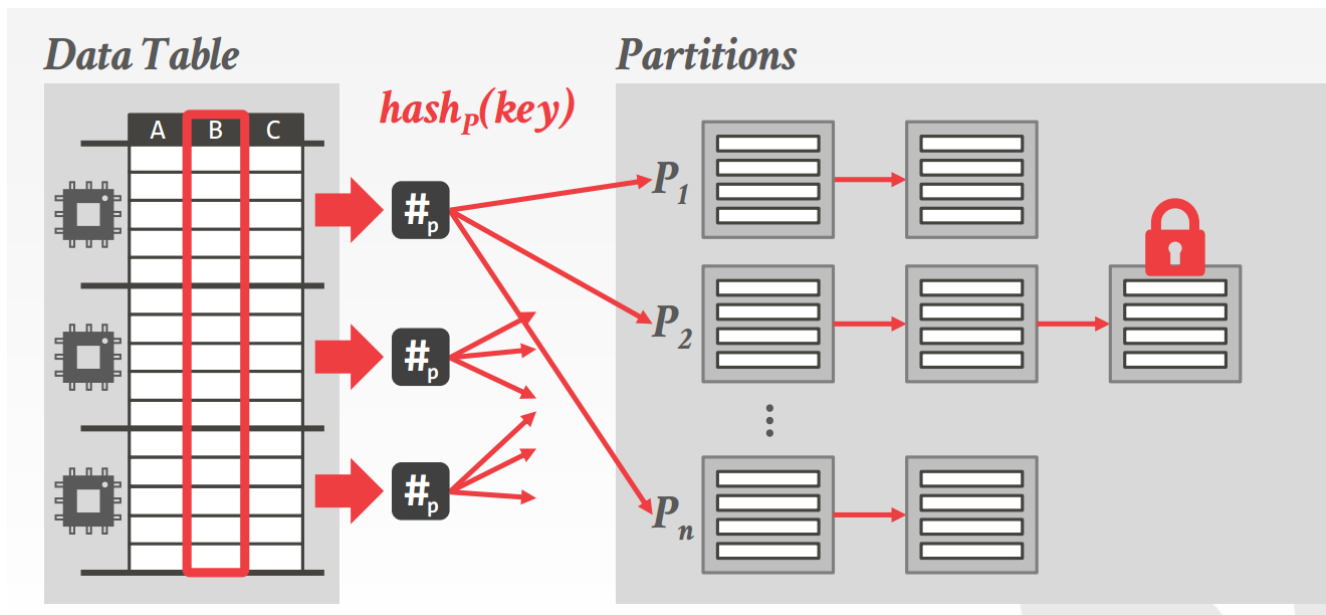
Approach #1: Shared Partitions

- Single global set of partitions that all threads update.
- Must use a latch to synchronize threads.

Approach #2: Private Partitions

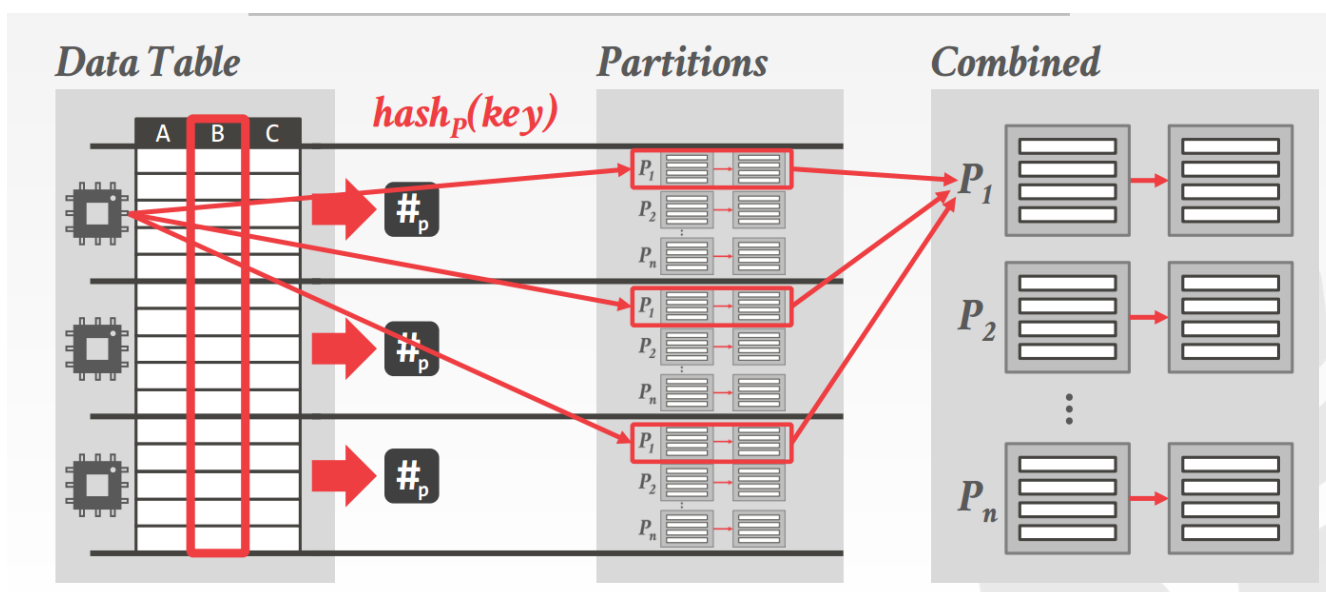
- Each thread has its own set of partitions.
- Must consolidate them after all threads finish.

- Shared Partitions



性能开销在于latch同步比较影响性能

- Private Partitions



性能开销主要是copy data了两次，如果是行存而且一行tuple很大的话，copy开销其实不小。

- Blocking Partitioning (Radix Partitioning)

RADIX PARTITIONING

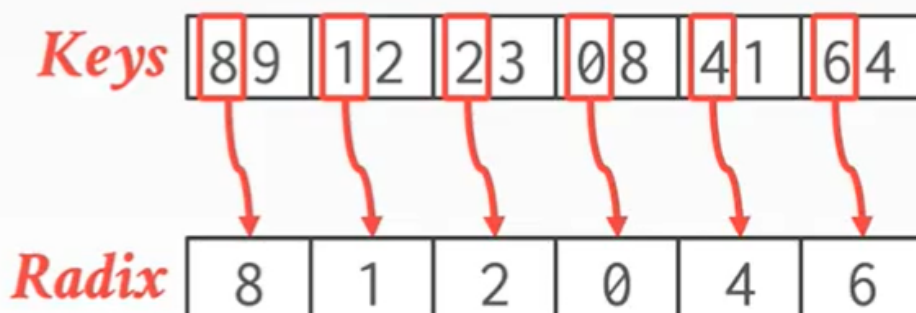
Scan the input relation multiple times to generate the partitions.

Multi-step pass over the relation:

- **Step #1:** Scan **R** and compute a histogram of the # of tuples per hash key 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.

RADIX

The radix of a key is the value of an integer at a position (using its base).



Build Phase

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.

- hash table

HASH TABLE

Design Decision #1: Hash Function

- How to map a large key space into a smaller domain.
- Trade-off between being fast vs. collision rate.

Design Decision #2: Hashing Scheme

- How to handle key collisions after hashing.
- Trade-off between allocating a large hash table vs. additional instructions to find/insert keys.

HASHING SCHEMES

Approach #1: Chained Hashing

Approach #2: Linear Probe Hashing

Approach #3: Robin Hood Hashing

Approach #4: Hopscotch Hashing

Approach #5: Cuckoo Hashing

Probe Phase

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

