East China Normal University

Advanced database system, Autumn 2019

多核 CPU 下并行哈希连接算法的实现

杜云滔　10153903105

10153903105@stu.ecnu.edu.cn

2019 年 6 月

摘　要

本文总结了在多核 CPU 处理器下，如何实现高效的并行哈希连接算法。从 Partition、Build 和 Probe 三个阶段出发，分别介绍了各阶段的主要实现方式：在 Partition 阶段，可以分为 *non-blocking* 和 *blocking* 两类，着重介绍了 Radix partition 的算法流程；在 Build 阶段，介绍了两种构建哈希表的方法。最后，在不同数据分布和不同超参数下，对比了不同哈希连接算法的实验结果。结果发现，No partition 的简单哈希连接算法效率最高，尤其是在数据分布有偏的情况下。

关键词：Hash Join，Radix partition，Multi-core CPUs

目录

1 引言

目前，大多数计算机已经是多核 CPU 处理器，且核数不断增加。与此同时，内存的价格在不断下降，这导致很多数据库系统都可以将大部分数据直接放入内存中，在这种情况下，磁盘 I/O 已经不是主要的瓶颈。

因此，数据库系统需要考虑如何利用多核 CPU 来提高算子的效率。在此之前，已经出现了很多形式的并行数据库框架，例如：pure shared nothing, shared-memory, 以及 shared disk[5]。本文主要讨论了在内存型数据库下的哈希连接算子的实现。

对于并行哈希算子的实现，主要有两方面的考虑：一是希望能最小化 cache miss，例如 Radix partition；二是希望能最小化 CPU 同步代价，例如 No partition 的简单哈希连接。根据论文 [1] 的实现结果，我们发现，使用多线程可以使得 cache miss 对算子效率的影响降低。例如，No partition 的实现方式比大部分进行 Partition 的哈希连接的效率都要高，特别是在数据有偏的情况下。同时，No partition 还有“parameter-free”的优势。

本文主要介绍了在并行哈希连接算法中的每个阶段，可供选择的实现方法。同时对论文 [1] 中的实验结果进行了简单的分析。

2 并行哈希连接算法实现

一个 Join 算子作用在两个关系上，记为 R 表和 S 表。这里假设 $|R| < |S|$。一个典型的哈希连接算法一般包含三个阶段，分别为：Partition，Build 和 Probe[2]。Partition 阶段是可选的：在 join key 上使用哈希函数将 tuples 分成多个不相交的子集存放在内存中；Build 阶段会扫描 R 表，并在内存中以 join key 建立一个哈希表；Prob 阶段：扫描 S 表，通过 join key 在哈希表中查找对应的 tuples，并进行输出。

下面将分别讨论每个阶段的不同实现方式。

2.1 Partition 阶段

该阶段是可选的。主要想法是：如果 R 和 S 表的某个 partition 能够放在 CPU cache 中，那么在 Build 阶段和 Prob 阶段的 cache miss 就几乎可以忽略不计。Shatdal [4] 等人已经通过实验得到：在 Partition 阶段花费在写入内存的代价

小于 cache miss 的代价。因此通过 Partition 阶段，可以提升哈希连接算子的总效率。

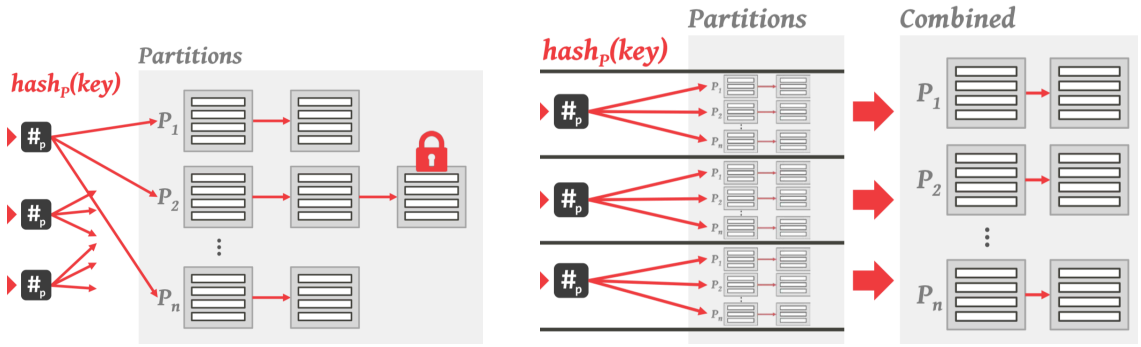
Partition 的过程很简单：所有的线程先从 R 表读取多个 tuple，根据每个 tuple 的 join key，通过一个哈希函数 $h_p(k)$，写入到 partition $R_{hp}(k)$ 中。对 S 表进行同样的操作，得到 partition $S_{hp}(k)$。

我们可以将 Partition 阶段的实现方式分为两类：*non-blocking* 和 *blocking* 。*non-blocking* 可以边扫描数据边输出结果；而 *blocking* 需要多次扫描数据，最后一起输出结果。需要注意的是，哈希连接算子本身是阻塞的，而这里的两类实现方式在时间上只占整个操作时间的很小一部分，因此都可以看作是非阻塞式的。

2.1.1 Non-blocking algorithms

主要有以下两种实现方式：

- **Shared Partitions**。对于所有线程来说，共用同一个 Partition 集合。由于多线程会并行对 Partition 进行修改，因此这种方式需要用锁进行同步。
- **Private Partitions**。每个线程有自己的 Partition 集合，因此不需要使用同步机制。但最终需要将所有的 Partition 子集进行合并。



(a) Shared Partitions　　(b) Private Partitions

图 1: Non-blocking algorithms

2.1.2 Blocking algorithms

另一种 Partition 的方式是使用 Radix partitioning 算法 [3]，其主要有三步：

1. 对 R 表进行第一遍扫描。将 Tuples 中的 key 的第 k 位进行哈希映射得到哈希值 $h_p(k)$，并得到哈希值的直方图。

2. 通过直方图的统计信息，计算相同哈希值 Tuples 数量的累加和，得到其在总的 Partition 上的偏移量，即 (Partition i,thread j)。也就是找到 tuples 进行 Partition 后的对应位置。
3. 再次扫描 R 表，通过偏移量 (Partition i,thread j)，将数据分配到相应的 Partition。该操作可以并行执行。根据需要，可以递归地对 $k+1$ 位置上的值再次进行 Radix partitioning。

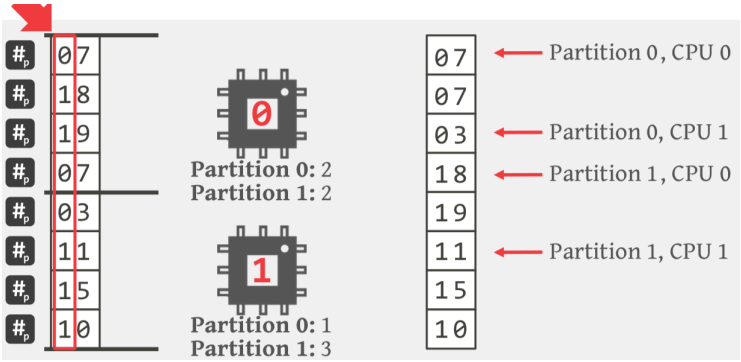


图 2: Radix Partitioning Algorithm

通过两次扫描数据，可以准确得到所需要内存的大小，使得数据可以连续的存储在内存块中，能有效降低 cache 和 TLB 的 miss rate[3]。同时，各线程可以并行执行而不互相干扰，也避免了使用同步机制带来的性能下降问题。

2.2 Build 阶段

在该阶段中，线程通过扫描 R 表或者创建好的 Partition 进行哈希映射，构建哈希表。需要注意的是：为了减少 Probe 阶段的 cache misses，每一个 bucket 的大小应和 cache lines 的大小相当；同时，该阶段使用的哈希函数，必须和 Partition 阶段的哈希函数不同（不然就没有意义了）。

若数据已经经过 Partition，那么对于每个线程，选取特定的 Partition，并建立自己的哈希表。而如果数据没有进行 Partition，那么所有线程共享一个哈希表，将数据加入某个 bucket 时，需要进行加锁进行同步。

常用的建立哈希表的方法有两种，以下分别进行介绍。

2.2.1 Chained Hash Table

该算法对每个 slot 维护一个单向链表，将所有具有相同哈希值的 tuple 放到该链表中。在插入、删除、查找等操作中，首先需要将 key 根据哈希函数映射到

对应的 bucket 头指针，然后对该链表进行扫描。

为了减少在 Join 操作中 tuple 的比较次数（也就是减少链表的长度），需要尽可能的减少哈希冲突。一般来说，需要使得 Chained hash table 的 slots 在 R 表元素个数的两倍以上。

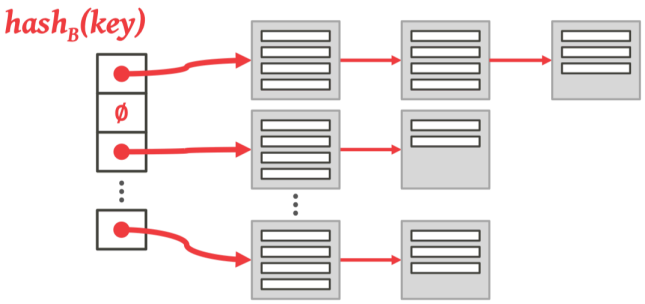


图 3: Chained Hash Table

2.2.2 Cuckoo Hash Table

Cuckoo hash 是一种解决哈希冲突的方法，其目的是使用简单的哈希函数来提高哈希表的利用率，同时保证 $\mathcal{O}(1)$ 的查询和删除时间。

基本思想是使用多个哈希函数来处理碰撞，从而每个 key 都对应到不同位置。插入操作如下：

1. 对 key 值进行哈希，生成多个哈希值，例如 $hash_{k1}$ 和 $hash_{k2}$，如果对应的两个位置上有一个为空，那么直接把 key 插入即可。
2. 否则，任选一个位置，把 key 值插入，把已经在那个位置的 key 值踢出来。
3. 被踢出来的 key 值，需要重新插入，直到没有 key 被踢出为止。

2.3 Probe 阶段

对于每一个 tuple，计算 key 的哈希值，从而到对应的 bucket 中查找是否存在匹配的 tuple，然后输出即可。

如果已经对数据进行了 Partition，那么相应的 key 值已经在同一个 Partition 中，那么只需要在对应线程建立的哈希表中进行查找即可。

	No-P	Shared-P	Private-P	Radix
Partitioning	No	Yes	Yes	Yes
Input scans	0	1	1	2
Sync during partitioning	-	Spinlock per tuple	Barrier, once at end	Barrier, 4 * #passes
Hash table	Shared	Private	Private	Private
Sync during build phase	Yes	No	No	No
Sync during prob phase	No	No	No	No

表 1: 不同哈希连接算法差异对比

3 不同并行哈希连接算法对比

本文主要对比以下几种哈希连接算法的实现：

1. **No partitioning join**。不进行 Partition 阶段的操作，在 Build 阶段直接创建一个共享的哈希表。
2. **Shared partitioning join**。在 Partition 阶段对 R 表和 S 表创建 Shared partitions。这需要用锁实现同步机制。
3. **Private partitioning join**。在 Partition 阶段对每个线程分别创建自己的 Partition 子集。
4. **Radix partitioning join**。在 Partition 阶段使用 Radix partitioning 算法建立 Partition，相同哈希值的 tuple 在内存中是连续存放的。

他们各自的特点对比如表 1所示。

3.1 Uniform Data Set

根据论文 [1] 的实验，如图 4所示，在数据分布均匀的情况下，Radix partitioning join 比不进行 Partition 的哈希连接算法快 20%，这是由于 Radix partitioning 算法让数据连续存放在内存块中，可以减少 cache miss 和 TLB miss。

3.2 Skewed Data Set

而在数据有偏的情况下（也就是相同 key 值的 tuples 很多），可以发现：

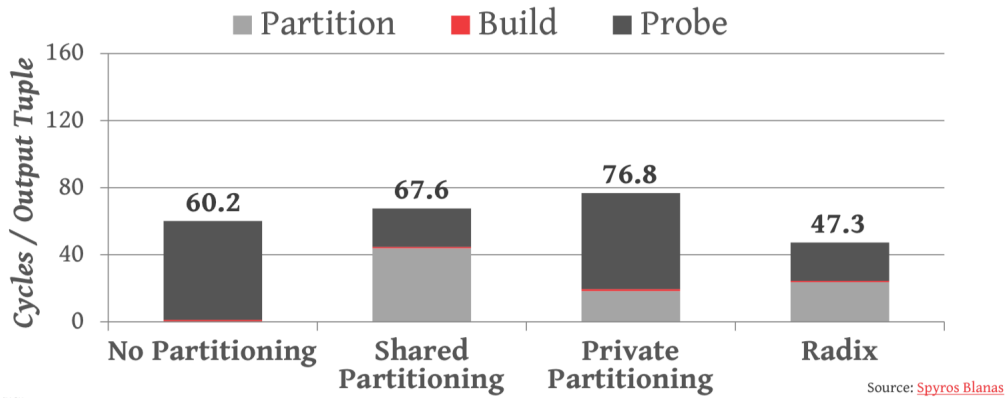


图 4: HASH JOIN –UNIFORM DATA SET

- Shared Partitioning 的效率最低。这是因为在该算法的 Partition 阶段，所有的线程都共用一个哈希表，需要使用锁进行同步。而当数据分布有偏时，相同哈希值数量很多，线程等待锁的时间增加，同步代价很大。
- Radix partition join 的效率比 No partitioning 低。这是因为，在 Radix 算法中，Partition 之后的数据连续存放在内存中，而由于数据有偏，无法将相同哈希值的 tuple 全部存放在 cache line 中，因此降低 cache miss 并不明显，反而因为 Partition 阶段需要扫描两次数据，而使得花费的总时间比不进行 Partition 更低。

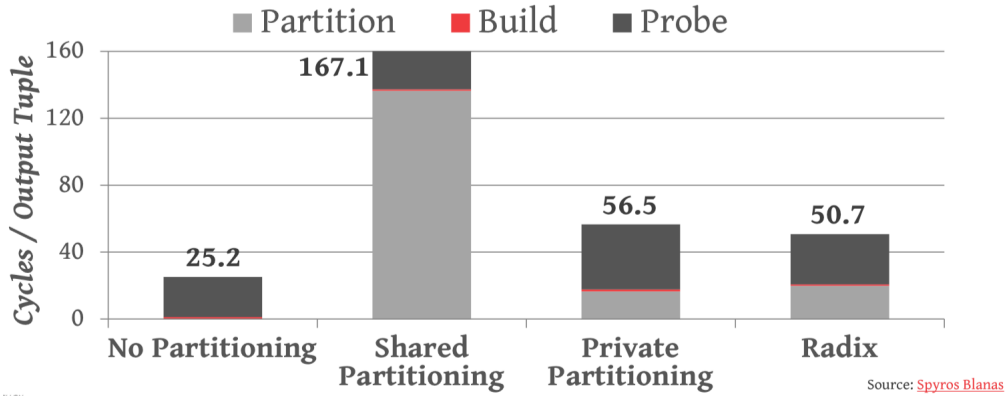


图 5: HASH JOIN –SKEWED DATA SET

3.3 超参数选择问题

在并行哈希链接算法算法中，有很多超参数的设置需要我们思考，例如：

- 由于 Partition 并不一定能带来效率的提高，那么我们什么时候该进行

Partition?

- 应该使用多少个 Partition?
- 在 Radix partitioning 算法中，应该迭代多少次?
- 如何决定最佳的线程数?

在现实环境中，这些超参数是由优化器自动选择，并没有一个明确的规则可供参考。同时，这些参数除了于数据本身的特点有关，也与硬件有关，加大了超参数选择的难度。

以线程数为例。如图 6所示，随着线程数的增多，No Partitioning 的速度几乎以线性进行增长。而对于 Radix partitioning 算法来说，当线程数多于实际的 CPU 核数，其性能几乎不再提升。这可能是由于，多线程可以充分利用 CPU，而 cache miss 和 TLB miss 的影响逐渐降低。

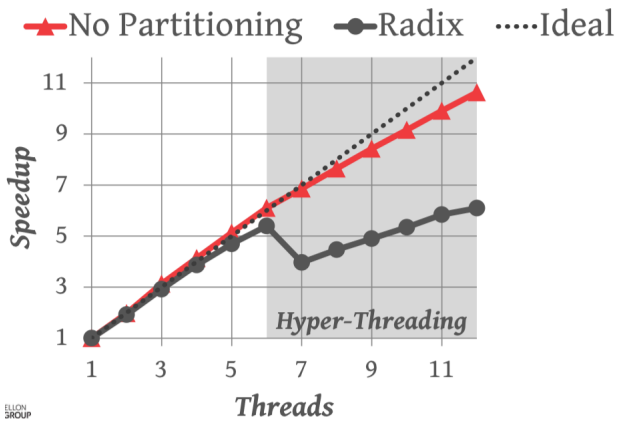


图 6: EFFECTS OF HYPER-THREADING

4 总结

本文从 Partition、Build 和 Prob 三个阶段介绍了并行哈希连接算法的基本实现方式，并通过对论文实验的分析，发现 No partition 的实现方式能最大化的利用多核 CPU，且随着线程数的增多，效果越好。在商业数据库中，由于大部分数据需要从磁盘中获取，磁盘 I/O 是主要的瓶颈，因此 Radix partitioning 算法中需要扫描两遍数据的代价就显得很大，在 Join 算子中并不会实现 Partition 阶段的操作。

参考文献

[1] Join Algorithms (Hashing), https://15721.courses.cs.cmu.edu/spring2016/slides/11.pdf

[2] Blanas S , Li Y , Patel J M . Design and evaluation of main memory hash join algorithms for multi-core CPUs[J]. Proceedings of the 2011 international conference on Management of data, 2011:37.

[3] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D.Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. PVLDB, 2(2):1378–1389, 2009.

[4] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.

[5] M. Stonebraker. The case for shared nothing. In HPTS, 1985.