Distributed Join Algorithms on Thousands of Cores

1170300513 陈鋆（单人完成阅读）

目录

一、概述

1.本文解决了什么问题？

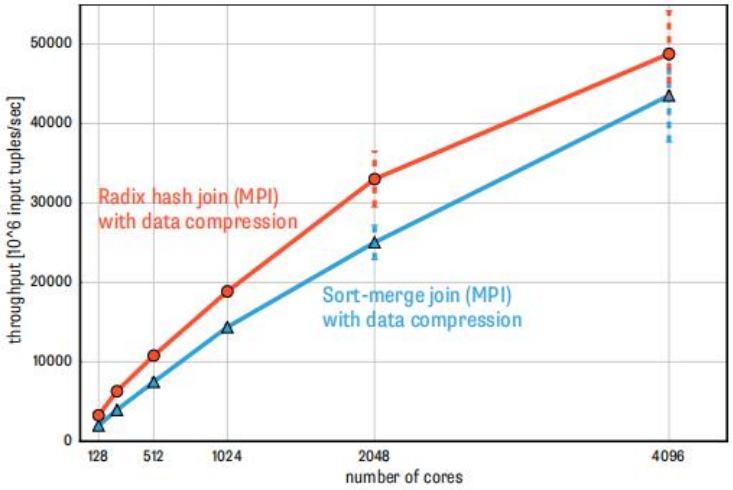
作者通过对 4096 个处理器核心上运行的高达 4.8TB 输入数据的分布式连接进行性能分析，探讨了在使用高带宽、低延迟网络和专用通信库取代手动调优代码时，连接算法的行为方式。

2.采用什么思想解决此问题？

首先，在实验环境的准备与配置上，本文的作者们采用了运用最新的工具的创新思想，使用了最新的远程直接数据存取（RDMA）和远程内存访问（RMA）网络，以及可移植的高性能通信网络计算接口 MPI；

而在实验的设计、分析中，本文的作者们则大量使用了控制变量、对照分析的思想：

通过基数散列连接和排序合并连接这两种连接方法在高带宽，低延迟网络构成的 4096 个处理器核心上运行的结果对比，分析连接算法在使用高带宽，低延迟网络和专用通信库取代手动调优代码时的行为方式；



在算法的描述中，作者给出了算法的代价预测模型，并在实验完成后与具体实验的结果进行对比分析，从而得出现实中连接算法在使用高带宽，低延迟网络和专用通信库取代手动调优代码时的行为方式的一些结论；

Radix hash join			
Phase	Exec. Time	Model	Diff.
Histogram Comp.	0.34s	0.36s	−0.02s
Window Allocation	0.21s	—	+0.21s
Network Partitioning	2.08s	0.67s	+1.41s
Local Partitioning	0.58s	0.67s	−0.09s
Build-Probe	0.51s	0.51s	+0.00s
Imbalance	0.62s	—	+0.62s
Total	4.34s	2.21s	+2.13s
Sort-merge join			
Partitoning	1.20s	1.02s	+0.18s
Window Allocation	0.06s	—	+0.06s
Sorting	1.99s	1.45s	+0.54s
Merging	1.81s	1.78s	+0.03s
Matching	0.26s	0.36s	−0.10s
Imbalance	0.38s	—	+0.38s
Total	5.70s	4.61s	+1.09s

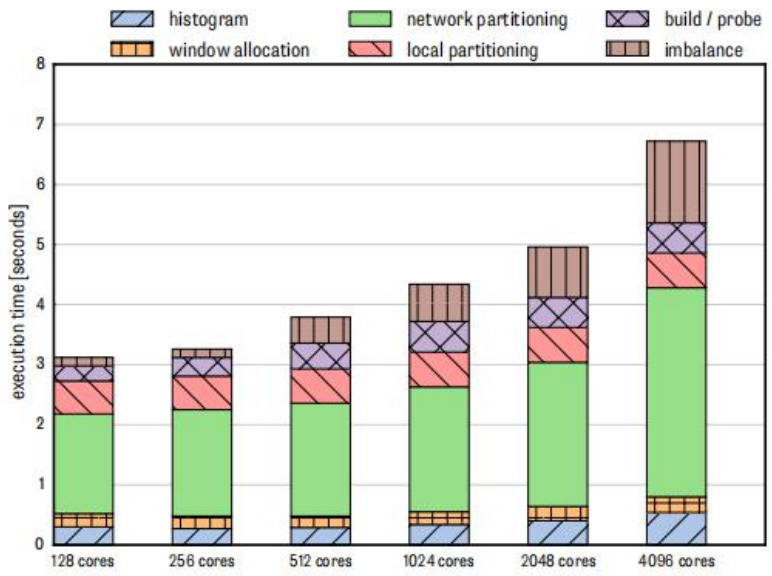
Parameters [million tuples per second]

RHJ: $P_{\text{scan}} = 225$, $P_{\text{Part}} = 120$, $P_{\text{net}} = 1024$, $P_{\text{build}} = 120$, $P_{\text{probe}} = 225$

SMJ: $P_{\text{part}} = 78$, $P_{\text{sort}} = 75$, $P_{\text{net}} = 1024$, $P_{\text{merge}} = 45$, $P_{\text{scan}} = 225$

在实验的具体过程中，作者也严格遵守控制变量法进行对比分析，如：同时增加核数与数据量比较算法运行时间、控制输出不变对比输入的关系大小与运行

时间的关系、控制输入的关系大小不变对比输出的匹配过程与运行时间的关系等。



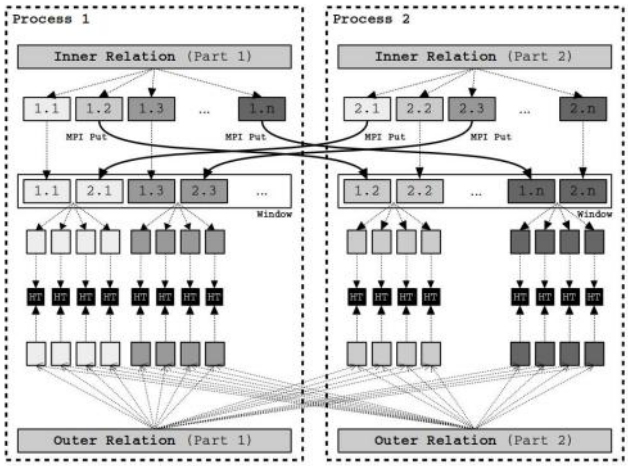
Workload		Radix hash j.		Sort-merge j.	
Input	Output	Time	95% CI	Time	95% CI
40M/40M	40M	4.34s	±0.15s	5.70s	±0.14s
20M/40M	40M	3.45s	±0.15s	4.67s	±0.23s
10M/40M	40M	2.88s	±0.29s	3.83s	±0.27s
10M/40M	20M	2.92s	±0.10s	3.75s	±0.25s
10M/40M	10M	2.91s	±0.18s	3.87s	±0.41s

另外，在 4.1.2 的网络的划分中，为了降低数据的大小，还采用了前序合并的压缩思想。

3.基本算法的描述。

本文中共实现了两个算法，分别为：Radix Hash Join（基数散列连接）算法和 Sort-Merge Join（排序合并连接）算法。

Radix Hash Join（基数散列连接）



(1) Histogram & Assignment Computation（柱状图和分配计算）

分布式基数散列连接计算全局直方图，以便确定通信缓冲区和存储器窗口的大小。使用该直方图，该算法可以确定（i）分区到节点的分配和（ii）在每个进程可以专门写入的存储器窗口内的一组偏移，从而减少在连接操作期间所需的同步量。

计算直方图 T_{hist} 所需的时间取决于输入关系（R 和 S）的大小以及每个 p 进程可以扫描数据的速率 P_{scan}。

$$T_{\text{hist}} = \frac{|R| + |S|}{p \cdot P_{\text{scan}}}$$

(2) Multi-Pass Partitioning（多通道分区）

分区阶段的目标是创建内部关系的小缓存分区。将一个大的分区扇出从而创建大量的小分区。需要注意的是：如果分区的数量大于 TLB 大小，则随机访问大量位置会导致 TLB 未命中的显着增加，并且如果分区的数量大于可用高速缓存行的数量，则会导致数据丢失。故采用多遍分区策略来解决这个问题：通过在每次传递中使用不同的散列函数，使得每个通道中，前一遍的分区被细化，使得每个步骤中的分区扇出 FP 不超过 TLB 和缓存条目的数量。传递次数取决于内部关系 R 的大小。

$$d = \lceil \log_{FP} (|R|/\text{cache size}) \rceil$$

分区阶段可以与网络上的数据重新分配交织。 如果分区将在本地处理，则将关系划分为本地缓冲区；如果已将其分配给远程节点，则将关系划分为 RDMA

缓冲区。远程写入操作以规则的间隔执行，以便交错计算和通信。 每个进程的分区速率 P_{net} 受到进程 P_{part} 的分区速度（计算限制）或可用网络带宽 BWnode 的限制，可用网络带宽 BW_{node} 在相同节点上的所有 t 个进程之间共享（带宽限制）。

$$P_{\text{net}} = \min\left(P_{\text{part}}, \frac{BW_{\text{node}}}{t}\right)$$

后续分区传递在本地执行，不进行任何网络传输。 因此，每个过程可以以分区速率 P_{part} 分配数据。分割数据所需的总时间等于迭代数据 d 次所需的时间。

$$T_{\text{part}} = \left(\frac{1}{p \cdot P_{\text{net}}} + \frac{d-1}{p \cdot P_{\text{part}}}\right) \cdot (|R| + |S|)$$

在此阶段结束时，数据已在所有进程之间进行分区和分配。 匹配的元组已分配给同一分区。

(3) Build & Probe（构建与探索）

在构建阶段，为内部关系的每个分区 R_p 创建一个哈希表。 由于哈希表适合处理器缓存，因此可以在高速率 P_{build} 中执行构建操作。 生成的分区数取决于分区扇出 FP 和分区传递次数 d。

$$T_{\text{build}} = (F_P)^d \cdot \frac{|R_p|}{p \cdot P_{\text{build}}} = \frac{|R|}{p \cdot P_{\text{build}}}$$

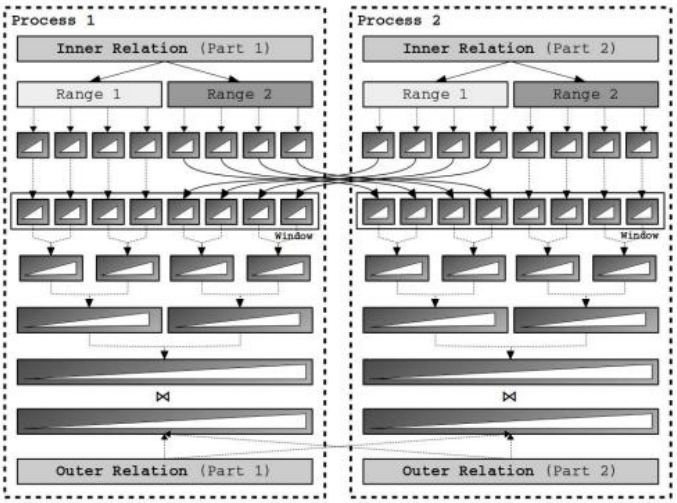
来自外部关系的对应分区 Sp 的数据用于探测哈希表。 探测高速缓存中的哈希表需要在外部关系 S 上进行单次传递。

$$T_{\text{probe}} = (F_P)^d \cdot \frac{|S_p|}{p \cdot P_{\text{probe}}} = \frac{|S|}{p \cdot P_{\text{probe}}}$$

(4) 性能模型

$$T_{\text{rdx}} = T_{\text{hist}} + T_{\text{part}} + T_{\text{build}} + T_{\text{probe}}$$

Sort-Merge Join（排序合并连接）



(1) Sorting（排序）

在排序操作的第一阶段，每个线程对输入数据进行分区。 使用范围分区来确保将两个关系中的匹配元素分配给同一台机器进行处理。 因为使用了连续的密钥空间，所以可以将输入关系分成相同大小的范围。 对数据进行范围分区所需的时间取决于输入关系的大小，进程数 p 和分区速率 P_{part}。

$$T_{\text{part}} = \frac{|R| + |S|}{p \cdot P_{\text{part}}}$$

数据分区后，将创建固定大小 l 的单个 runs。 运行总数取决于两个输入关系中每个输入关系的大小以及每个 run l 的大小。

$$N_R = \frac{|R|}{l} \quad \text{and} \quad N_S = \frac{|S|}{l}$$

一个 run 被排序，然后异步传输到目标节点。 在进行网络传输的同时，该过程可以继续对下一轮输入数据进行排序，从而交叉处理和通信。 算法的性能可以受到可以对 run 进行排序（计算限制）的速率 P_{run} 或可用于同一节点上的所有 t 进程的网络带宽 BW_{node} 的限制。

$$P_{\text{sort}} = \min\left(P_{\text{run}}(l), \frac{BW_{\text{node}}}{t}\right)$$

将输入元组排序为小排序 runs 所需的总时间主要取决于输入大小。

$$T_{\text{sort}} = (N_R + N_S) \cdot \frac{l}{p \cdot P_{\text{sort}}} = \frac{|R| + |S|}{p \cdot P_{\text{sort}}}$$

在进程对其输入数据进行排序后，它会等待，直到它从其他节点收到其范围的所有已排序运行。 一旦收到所有数据，算法就开始使用 m 路合并来合并排序的运行，m 路合并将多个输入运行组合成一个排序的输出。 在两个关系完全排序之前，可能需要对数据进行多次迭代。

$$d_R = \lceil \log_{F_M}(N_R/p) \rceil \quad \text{and} \quad d_S = \lceil \log_{F_M}(N_S/p) \rceil$$

从两个合并树的深度，我们可以确定合并两个关系的 runs 所需的时间。

$$T_{\text{merge}} = d_R \cdot \frac{|R|}{p \cdot P_{merge}} + d_S \cdot \frac{|S|}{p \cdot P_{merge}}$$

在排序阶段之后，两个关系在所有节点之间被分区。 在每个分区中，元素都是完全排序的。

(2) Joining Sorted Relations（连接排序好的关系）

为了计算连接结果，每个进程将内部关系的已排序分区与外部关系的对应分区合并。

$$T_{\text{match}} = \frac{|R| + |S|}{p \cdot P_{\text{scan}}}$$

(3) 性能模型

$$T_{\text{sm}} = T_{\text{part}} + T_{\text{sort}} + T_{\text{merge}} + T_{\text{match}}$$

4.算法分析的结论。

（i）实现最高性能需要在计算和通信容量之间取得适当的平衡。 向计算节点添加更多核心并不总能改进性能，也有可能会使性能恶化。

（ii）虽然两种连接算法都可以很好地扩展到数千个核心，但是通信效率低下会对性能产生重大影响，这表明需要做更多工作才能有效地利用大型系统的全部容量。

（iii）散列和排序合并连接算法具有不同的通信模式，这导致不同的通信成本，使得计算节点之间的通信的调度成为关键组件。

（iv）作者的性能模型表明排序合并连接实现实现了其最大性能。 另一方面，基数散列连接远远超出其理论最大值，但仍然优于排序合并连接，证实了先前基于散列和排序的连接算法比较的研究。

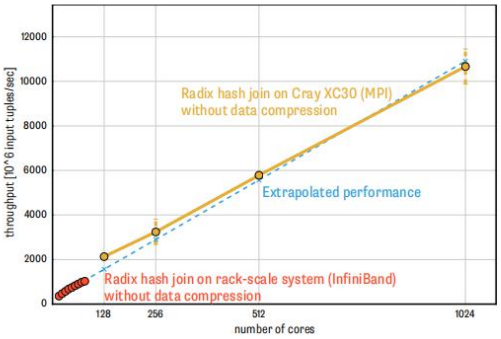
5.用一个例子说明相关算法。

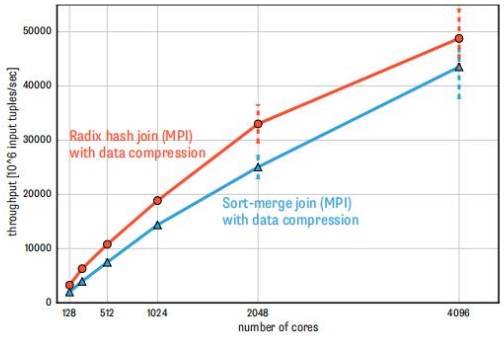
4096 个 CPU 核进行连接算法的例子现实中少见，故此处引用本文中的例子。

作者实验例子中，使用的 Cray XC30 [11]有 28 个计算机柜，实现了分层架构：每个机柜最多可以安装三个机箱。一个机箱最多可以容纳 16 个计算刀片，其中由四个计算节点组成。 整个系统最多可提供 5,272 个可用计算节点。计算节点是具有 32 GB 主内存的单插槽 8 核（Intel Xeon E5-2670）计算机。它们通过 Aries 路由和通信 ASIC 以及 Dragonfly 网络拓扑连接，峰值网络二分带宽为 33 TB / s。Aries ASIC 是一种片上系统设备，包括四个 NIC 和一个 Aries 路由器.NIC 为同一刀片的所有四个节点提供网络连接。 每个 NIC 通过 16x PCI Express 3 接口连接到计算节点。 路由器连接到机箱背板并通过它连接到网络结构。用于实验的第二台机器是 CrayXC40 机器。 它具有与 XC30 相同的架构，但在节点设计上有所不同：每个计算节点有两个 18 核（Intel Xeon E5-2695 v4）处理器和每个节点 64 GB 的主内存。

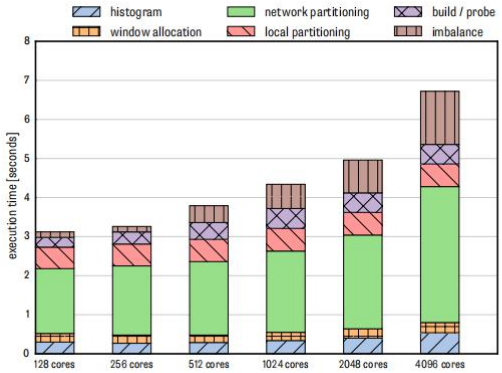
在作者的例子中，一个进程每个关系提供多达 4000 万个元组，这导致 4,096 个核心上的 4.8 TB 输入数据。 内部和外部关系的相对大小在 1 比 1 和 1 比 8 之间。为了创建输入关系，分配每个节点以生成一系列键。 节点确保生成的密钥是随机顺序的。 接下来，输入被分成块。 在每对过程之间交换这些块。 在混洗操作之后，每个节点都拥有整个值范围内的密钥。 对数据的最终传递将元素置于随机顺序中。

结果：

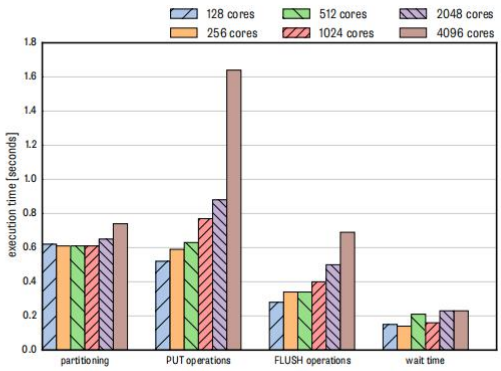




Radix Hash Join：

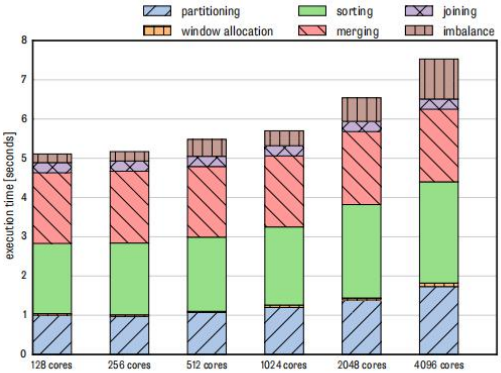


(a) Total join execution

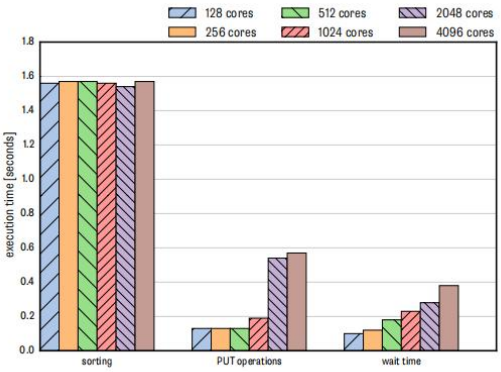


(b) Network partitioning pass

Sort-Merge Join：



(a) Total join execution



(b) Sorting phase

二、小综述（加分项）

1. 已有研究工作。

（1）Hash join 散列连接是 CBO 做大数据集连接时的常用方式，优化器使用两个表中较小的表（通常是小一点的那个表或数据源）利用连接键（JOIN KEY）在内存中建立散列表，将列数据存储到 hash 列表中，然后扫描较大的表，同样

对 JOIN KEY 进行 HASH 后探测散列表，找出与散列表匹配的行。现代硬件架构下的并行 Hash Join 算法主流有两类，分别是不关心硬件参数(Hardware-Oblivious)的哈希连接算法，和对硬件参数敏感(Hardware-Conscious)的哈希连接算法。

（2）Sort-Merge Join 是先将关联表的关联列各自做排序，然后从各自的排序表中抽取数据，到另一个排序表中做匹配。

（3）利用多核处理器的并行资源对内存数据库中哈希连接算法进行优化。

（4）用 DSVM（分布式共享虚拟存储器）和消息传递两种方式分别实现并行哈希连接操作。

2.各工作简介。

（1）Hardware-oblivious hash join（不关心硬件参数的哈希连接算法）：

Hardware-oblivious hash join 顾名思义就是其不需要根据硬件环境的变化来调整算法参数。no partitioning hash join 就属于这一类算法，不分区的哈希连接算法分为两个阶段：build 和 probe。在 build 阶段，所有线程同步构建一张哈希表。在 probe 阶段，所有线程共同工作，在哈希表中找到与 S 表中元组相匹配的桶。

Hardware-conscious hash join（硬件参数敏感的哈希连接算法）

与 hardware-oblivious hash join 相反，hardware-conscious hash join 对硬件环境较为敏感，在不同参数下的算法性能可能差别较大，因此往往需要根据硬件参数仔细调整算法参数。Radix Hash Join 算法就属于这一类算法。有分区的哈希连接算法过程分为 3 个阶段: partition、build 和 probe 阶段。partition 阶段将 R 表的数据划分到不同的分区上，build 阶段在各个分区上构建哈希表，probe 阶段先将 S 表中的元组映射到对应的分区上，再进行连接操作。

（2）Sort-Merge Join ：

排序合并连接没有驱动表一说，两个表/行源是对等关系。排序合并连接原理是先对两个表/行源根据 JOIN 列进行排序(当然了排序的时候要踢出不符合 where 条件的列)，然后再进行连接。排序合并连接可以处理非等值 JOIN。对于非等值 JOIN，一般都要使用 SORT-MERGE JOIN 算法。

（3）利用多核处理器的并行资源对内存数据库中哈希连接算法进行优化：

利用线程级并行和数据级并行优化哈希连接算法。提出了基于多核MapReduce 模型的并行哈希连接算法,包括非划分哈希连接和划分哈希连接。其次,为并行哈希连接算法提出了一种改进的 Cuckoo 哈希表,该表由传统 Cuckoo 哈希表和链式哈希表组成,通过提升哈希表的读写性能来提升并行哈希连接算法的性能。基于上述成果,同时提出了几种优化方法,包括:利用 SIMD 指令优化、多步划分优化、负载平衡优化。最后,通过实验验证了本研究中提出的优化方法可行有效,实验表明:(1)基于多核 MapReduce 模型的并行哈希连接算法与前期算法相比,取得较好的效果;(2)在多核处理器环境下,划分哈希连接大部分情况下都优于非划分哈希连接,且当线程数量较大时内存存取成为性能瓶颈;(3)影响哈希连接算法性能的因素包括:哈希表的结构、划分数量、划分次数、线程数量、数据集等。

（4）用 DSVM（分布式共享虚拟存储器）和消息传递两种方式分别实现并行哈希连接操作：

采用了并行 GRACE 哈希连接算法,并对其进行了改进：并行 GRACE 哈希连接算法实现了子连接内部的并行计算, 即多个站点共同完成一个子连接分片的连接;改进后的并行 GRACE 哈希连接算法是子连接间的并行计算,也就是说,各个站点分别完成不同的子分片的连接工作。

在此基础上实现的第一种算法是基于 Shusse-Uo 系统的 DSVM 机制的并行 GRACE 哈希连接算法;另两种是基于消息传递的并行 GRACE 哈希连接算法,根据使用的存储方式不同又分为 :①使用 Shusse-Uo 系统提供的挥发堆作为存储对象和各种数据结构的空间,但算法中仅使用各个站点的本地堆, 即不使用 DSVM 机制映射的远地堆 ,靠应用程序完成站点间的消息传递来进行并行工作;②完全使用操作系统提供的虚存(Virtual Memory)作为存储空间,既不使用 DSVM 机制, 也不使用堆管理, 也就是说 ,除从原始数据堆中取对象外 ,可完全脱离原系统。

3.比较各工作的算法。

（1）两种哈希连接算法的比较：

两种算法的可扩展性都较好，radix 哈希连接算法可扩展性更强。在硬件环境一样的情况下，radix 哈希连接几乎总是优于 no partitioning 哈希连接，但当 R 表和 S 表元组数量差距较大时对不分区的算法有利。因此，现代的硬件架构并不能很好地隐藏 cache 丢失带来的影响，hardware-oblivious 连接算法性能通常不如使用适合参数的 hardware-conscious 连接算法。另外，在某些情况下，hardware-conscious 算法具有更好的稳定性。因此在设计哈希连接算法时应当仔细考虑硬件环境，调整算法的参数。

（2）哈希连接算法和排序连接算法的比较：

哈希散列连接适用于较小的表完全可以放于内存中的情况，这样总成本就是访问两个表的成本之和。但是在表很大的情况下并不能完全放入内存，这时优化器会将它分割成若干不同的分区，不能放入内存的部分就把该分区写入磁盘的临时段，此时要有较大的临时段从而尽量提高 I/O 的性能。因为 sort-merge join 需要做更多的排序，所以消耗的资源更多。 通常来讲，能够使用 sort-merge join 的地方，hash join 都可以发挥更好的性能,即散列连接的效果都比排序合并连接要好。然而如果行源已经被排过序，在执行排序合并连接时不需要再排序了，这时排序合并连接的性能会优于散列连接。

（3）多核处理器的并行资源对哈希连接算法进行优化后与优化前的比较：

改进的 Cuckoo 哈希表对替换路径设置了上限，避免了较多的替换操作;链式哈希表作为辅助哈希表，避免了插入失败情况的发生;充分利用了多核处理器线程级并行和数据级并行，且较好的解决冲突问题，提高了并行插入效率。改进的 Cuckoo 哈希表能够支持多线程读写、可变长度变量，并获得了较好的总体性能。

划分哈希连接算法可以通过多线程并行执行有效地提升算法效率 SIMD 优化提高了划分哈希操作中探查过程的效果，但由于探查过程并不是划分哈希连接中的主要过程，所以 SIMD 优化带来的效果不及非划分哈希连接。总的说来优化后的划分哈希连接算法与前期的算法相比具有优越性。

（4）用 DSVM（分布式共享虚拟存储器）和消息传递两种方式分别实现并行哈希连接操作的三种算法比较：

改进的并行 GRACE 哈希连接算法分为 3 步:分片, 建立哈希表和匹配哈希表。整个连接过程由一个客户进程控制完成, 初始条件是进行连接的内外聚集分别位于两个不同的站点 ,以下两点分析了 3 种算法的主要不同之处：

①分片阶段：分片操作是为了在多机上并行处理某一操作 ,而将操作对象分为若干部分 ,使各个部分在多机上并行运行 • DSVMA 算法在此阶段中 ,存放内外聚集的两个站点并行地遍历内外聚集中的对象 ,进行分片操作。根据参加并行连接的站点总数 N , 分别在两个站点打开 N 个挥发堆,将所有的子连接分片 Round-Robin 的分配方式存放到 N 个挥发堆中 •这样在连接阶段 , N 个站点可分别从两个分片站点的 N 个堆中依次取出所需完成的内外子连接分片, 进行并行连接操作,所有子连接的连接结果之并即为两聚集的连接结果。分片结束后, 将内外子分片对象的 OID 传给所有参与连接的站点 ,各站点靠子分片对象的 OID 可取得相应子连接分片的对象。

而基于消息通信的两种算法中 ,两个分片站点边遍历内外聚集, 边根据对象的分片属性值 ,以基于消息通信的方式把对象传递到相应的站点,并存储到本地挥发堆(VH-MP)或虚存(VM-MP)中。这样在分片结束时,各个站点所必须完成的子连接分片的所有对象已实际位于该站点上。

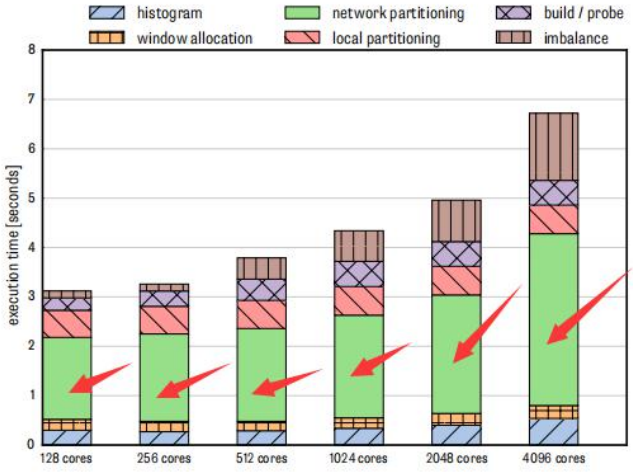
②建立哈希表和匹配哈希表阶段：这些阶段, N 个站点并行工作完成连接操作。3 种算法的主要不同之处在于:DSVMA 和 VH-MP 把哈希表建立在本地挥发堆中,整个搜索哈希表和匹配过程是在堆中完成的 ;而 VM-MP 算法, 把哈希表建立在虚存中,其相应过程是在虚存中完成的。3 种算法的最后连接结果存放到各站点的挥发堆中。通过检测这阶段的时间可分析该数据库系统和实际操作系统的存储管理性能相差多少。

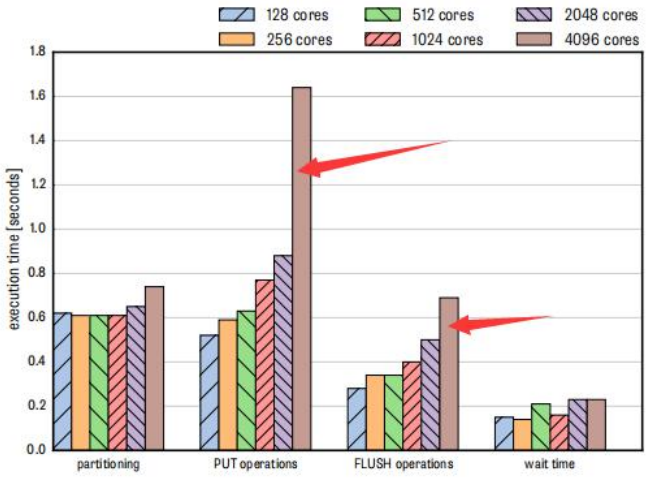
三、描述已有算法的不足（加分项）

不足之处：

在 Radix Hash Join 算法中，随着数据与处理数据的 CPU 核数的同步增加，the

network partitioning phase （网络分区阶段）耗费的时间显著增加。在 the network partitioning phase （网络分区阶段）的各部分中，可以观察到，对数据进行分区所需的时间保持不变，最多可达 1,024 个核心。 从 1,024 个核心开始，分区扇出必须增加到超出其最佳设置，这会导致较小的性能损失。 大部分额外时间增加于 MPI Put 和 MPI Flush 操作。





原因分析：

MPI Put 和 MPI Flush 操作，这两个操作产生传输数据的请求，分别确保数据传输已完成。 这种增加是由管理大量缓冲区和缺乏任何网络调度的额外开销引起的。散列连接交错分区和网络通信。 为此，它分配一个写入数据的临时缓冲区空间。 一旦缓冲区满足，就会生成 MPI Put 请求，并使用新缓冲区继续处理。 因为每个分区的缓冲区空间量相同并且使用了统一数据，所以将调度分区缓冲区以在类似的时间点进行传输，从而导致网络上的时间热点。 拥有更多的

处理器机器会加剧这种情况。 由于硬件具有有限的请求队列，因此在尝试对请求进行排队时将阻止进程，从而导致 MPI Put 调用中的时间花费显着增加。 随着分区扇出的增加，这个问题进一步增强。 在网络分区阶段，每个进程同时与系统中的每个其他进程通信。 拥有更多活跃的通信渠道会产生巨大的开销。

四、针对算法的不足提出的改进想法（加分项）

针对三中所描述的 Radix Hash Join 算法随着数据与处理数据的 CPU 核数的同步增加，the network partitioning phase （网络分区阶段）耗费的时间显著增加的不足，本人主要构想了两种可行的改进想法：

1. 针对原因分析中的“ 由于硬件具有有限的请求队列，因此在尝试对请求进行排队时将阻止进程，从而导致 MPI Put 调用中的时间花费显着增加。”，**对底层的硬件进行改进，增大连接网络中的带宽**，使硬件能容纳更多的请求队列，避免在尝试对请求进行排队时将阻止进程所导致的 MPI Put 调用中的时间花费的增加。

2. 针对原因分析中的“因为每个分区的缓冲区空间量相同并且使用了统一数据，所以将调度分区缓冲区以在类似的时间点进行传输，从而导致网络上的时间热点。 拥有更多的处理器机器会加剧这种情况。”**对 Radix Hash Join 算法进行改进，改进算法中的网络调度技术，**可采用与 Sort-Merge Join 类似的，在交错分区和执行 MPI Put 调用之间交替在发送器端创建均匀的传输速率，尽可能传输的时间，避免网络上的时间热点，以便算法在扩展时仍能保持良好的性能。

五、本次阅读论文的一些心得体会（自己添加）

1. 对论文的认知感触：

（1）ABSTRACT 是对论文的简短、概括性的摘要，简洁而有力地叙述了本文的研究背景、研究内容，是我们快速了解论文情况的一项有力工具（特别针对此次阅读论文的任务要从数目较大的论文中选择一篇自己感兴趣的阅读的情况，通

过 ABSTRACT 快速了解就显得尤为重要！）

（2）为了让读者能够更加客观全面地了解论文内容，如果实验中有用到某些特殊的工具，需要增加 BACKGROUND 进行说明：本文中就对其使用的 RMDA、RMA、MPI 进行了相应的背景介绍。

（3）为了让读者能对实验结果有更加直观的了解，论文中在需要的地方可以使用表格、堆积柱形图、扇形图等多种方式来表示实验结果，辅助文字说明。

（4）论文中对于实验的结果，要有对现象的分析，如：实验结果和理论预测有哪些误差，为什么会出现这些误差；并且对实验中所暴露出的不足进行反思，甚至可以提出有哪些可行的改进方向。

2. 此次论文阅读总结出来的一些经验与小技巧：

（1）正如上面所述的 ABSTRACT 的作用，可以通过阅读相应的 ABSTRACT 快速了解论文大致内容，以筛选得到阅读的论文。

（2）阅读论文时，可以边读边梳理所读的内容，整理出一个小提纲，否则可能会出现读到后面把前面的内容给忘了、导致无法整体把握论文的逻辑、架构的恶劣情况。

（3）正如报告中的小综述所要求我们做的那样，阅读论文前，可以搜索其它与本论文相关的研究、论文，了解一下本论文之前已有的那些研究，这样能帮助我们更清晰地了解作者为什么会提出这个研究内容以及作者的思路。

（4）最后最后，英语能力对阅读论文非常之重要！！一定要掌握一些专业相关的名词，对论文阅读非常有帮助！（本次论文选择时我就把 join 算法误以为是已学过的归并算法！最后没想到是数据库中才学到的连接算法，而后不得不自学相关的算法知识.....）