文章编号: 1000-5641(2017)05-0001-10

基于分布式系统 OceanBase 的并行连接

徐石磊, 王 雷, 胡卉芪, 钱卫宁, 周傲英

(华东师范大学 计算机科学与软件工程学院, 上海 200062)

摘要: 随着应用数据的飞速增长以及分布式数据库系统的不断涌现, 数据存储在物理独立的节点已经成为一种趋势. 在这种情况下, 当应用需要进行复杂 join 查询时, 就会不可避免地产生非常多的网络传输代价. 所以, 如何提高分布式系统中 join 查询的效率成为研究热点. 本文在分析分布式数据库系统 OceanBase 执行 nested loop join、Hash join、semi-join 等算法的基础上, 提出了合理利用硬件资源采用多线程并行执行 join 操作的优化思想, 并在 OceanBase 数据库中分别对 nested loop join、Hash join、semi-join 等算法进行了并行改造. 实验结果表明, 在一定线程数内 join 算法执行效率与并行度呈正相关.

关键词: 查询; semi-join; OceanBase; 并行连接

中图分类号: TP392 **文献标志码**: A **DOI:** 10.3969/j.issn.1000-5641.2017.05.001

Parallel join based on distributed system OceanBase

XU Shi-lei, WANG Lei, HU Hui-qi, QIAN Wei-ning, ZHOU Ao-ying

(*School of Computer Science and Software Engineering, East China Normal University, Shanghai* 200062, *China*)

Abstract: With the rapid growth of application data and the continued development of distributed database systems, data storage in physical independent nodes has become a trend. In this trend, when the application needs to perform complex join queries, it inevitably generates a lot of network traffic. Therefore, improving the efficiency of join query in distributed system is a hot topic. Based on the analysis of the nested loop join, Hash join, semi-join in the OceanBase, this paper puts forward the optimization idea of using hardware resources reasonably and using multithread to execute join operations in parallel. We implement experiment on OceanBase with nested loop join algorithm, Hash join algorithm, semi-join algorithm respectively. The experimental results confirm that the efficiency of join algorithm is positively related to parallelism in a certain number of threads.

Key words: query; semi-join; OceanBase; parallel join

收稿日期: 2017-06-19

基金项目: 2017年上海市青年科技英才扬帆计划(17YF1427800)

第一作者: 徐石磊, 男, 硕士研究生, 研究方向为数据存储与数据挖掘. E-mail: xsl118857@sina.com.

通信作者: 胡卉芪, 男, 助理研究员, 研究方向为数据库. E-mail: hqhu@dase.ecnu.edu.cn.

0 引 言

OceanBase[1]是阿里巴巴公司开发的分布式数据库, 其设计目标是支持数百 TB 的数据量以及数十万 TPS、数百万 QPS 的访问量. 因此, 如何实现对如此庞大数据的快速查询服务, 给我们的工作提出了一个巨大的挑战.

通过对分布式数据库系统 OceanBase 的架构分析, 可以知道, 虽然 OceanBase 有多个数据存储节点和多个查询处理节点, 但是, 目前各个查询处理节点在处理 join (连接)工作时尚且无法实现协同工作. 因此, 无法通过将任务分解给多个查询节点协同工作的方式实现高效快速查询. 此外, 在查询处理节点中, 对于 join 查询操作, OceanBase 只是简单地实现了串行化执行, 并没有合理利用硬件资源和数据冗余存储的特性, 由此给查询带来了极大的网络传输和 join 时间, 导致系统在处理大表数据 join 查询时效率非常低下. 针对这一缺点, 我们提出了一种即时有效地并行连接优化方案: 将数据进行范围切分, 根据数据多重备份特性, 设置多线程读取数据; 每个线程独立读取数据, 数据达到后独立执行 join 操作. 从而实现了各个查询节点中 join 操作的并行化执行, 极大地提高了 OceanBase 做复杂 join 查询的效率. 本文的研究重点是 nested loop join、Hash join、semi-join 等 join 算法的并行化设计以及数据分片的切分方式.

论文的内容组织如下: 第 1 节简要介绍OceanBase 数据库的整体结构及传统的 nested loop join, merge sort join、Hash join 等 join 算法和 semi-join、分布式 join 算法; 第 2 节介绍OceanBase 中传统 join 算法的查询原理和基于 semi-join 的传统 join 算法查询原理; 第 3 节介绍在 OceanBase 中对 nested loop join、Hash join、semi-join等 join 算法的并行优化设计; 第 4 节从并行度对查询效率的影响、并行度对各 join 算法执行效率的影响,以及并行度对基于 semi-join 算法的各 join 算法执行效率的影响等 3 方面进行全面的实验验证; 第 5 节对本文进行总结.

1 相关工作

OceanBase 分布式数据库整体架构主要分为 4 个模块: 主控服务器 RootServer(以下简称为 RS)、数据存储服务器 ChunkServer(以下简称为 CS)、增量数据服务器 UpdateServer(以下简称 UPS)以及查询处理服务器 MergeServer(以下简称为 MS). RS 负责管理集群中的所有服务器, 一般设置一主一备两个 RS; UPS 主要负责处理系统的增量数据更新, 一般也设置一主一备两个 UPS; MS 则负责接收和解析用户的 SQL 请求, 经过词法分析、语法分析、查询优化等一系列操作后转发给相应的 CS 或者 UPS; CS 主要负责存储系统的基准数据, 基准数据一般存储两到三份, 可配置.

传统的连接算法有嵌套循环连接(nested loop join)算法、归并排序连接(merge sort join)算法以及哈希连接(Hash join)算法. 这 3 种算法的提出都有特定的问题背景和适用情况. 伴随着分布式数据库的发展和普及, Bernstein 等[2]又提出了 semi-join (半连接)算法, 此算法可大幅减少 join 过程中数据的传输代价. 此外, 在分布式系统中, 伴随着 MapReduce、Spark 等系统的发展, 越来越多的分布式系统开始采用类似 Map/Reduce[3]的计算模型, 这类系统旨在通过任务分解的方式, 减小 join 计算代价. 下面简要介绍这些算法的特点以及研究发展情况.

1.1 nested loop join 算法

Blasgen 等[4]在 1977 年提出了 nested loop join 算法. 该算法适合于两表数据量较小并

且内存可以存放的情况. 后来, 在共享内存架构下, Zhou 等[5]又提出了利用 SIMD 技术优化嵌套循环连接的3种方式: 复制外层循环; 复制内层循环; 旋转方式. 在无共享架构下, Spark 结构中采用广播形式将数据传到其他节点上[6].

1.2 merge sort join 算法

针对嵌套循环连接处理大数据时效率低的问题, Blasgen 等[4]提出了 merge sort join 算法. 该算法首先对两张表排序, 然后进行顺序扫描; 当其中一张表扫描结束后, 算法也立即结束. 这曾经被认为是最好的连接算法[7]. 在之后对归并排序连接算法的研究中, Kim 等[8]指出了归并排序连接算法的效率与 SIMD 宽度有关.

1.3 Hash join 算法

Babb[9] 在 1979 年第一次提出了以哈希(Hash)函数为基础的连接算法. 当小表数据量较小可放在内存中且小表的连接列具有非常好的选择性时, 效率很好. 随着硬件技术的更新换代, Boncz 等[10]提出了 Radix 连接算法, 算法充分利用了硬件资源, 极大地优化了连接算法在内存中的 Cache 缺失和 TLB 缺失.

1.4 semi-join 算法

Bernstein 在 1981 年提出了 semi-join 算法. 该算法针对一张小表和一张超大表的内连接, 目的在于利用小表的连接列对大表进行过滤. 但是 semi-join 算法需要构建过滤表达式以及一次额外的数据传输——将过滤表达式传到大表所在存储节点. 因此, 其适应于大表数据过滤后只有少量数据的情况.

1.5 分布式 join 算法

相比传统数据库处理 join 查询, 分布式数据库处理复杂 join 查询时, 可将join 操作分解为多个 join 任务分配给多台机器独立执行, 最后再将各部分 join 结果汇总. 例如, 文献 [3] 研究了如何将 join 操作分解成多个任务在 Map/Reduce 模型中执行. 这类分布式 join 算法原理都是将一个完整的 join 操作分解成多个小的 join 操作, 放在多个查询节点上并行执行, 最终将网络传输代价、join 计算代价分摊给多个查询处理节点.

2 分布式系统 OceanBase 中 join 算法的工作原理

2.1 nested loop join、merge sort join、Hash join 算法工作原理

图 1 为 nested loop join、merge sort join、Hash join 流程图, 其中, R 表作为驱动表, S 作为被驱动表, MS 是查询处理节点, CS 是数据存储节点.

join 操作符左右各有一个 Sort 操作符和一个 RpcScan 操作符, 其中 RpcScan 负责向 CS 存储节点请求数据. 在存储节点 CS 上的 Project、Filter 操作符, 用于过滤存储节点数据. 当 join 查询产生时, join 算法的左右两个子操作符 RpcScan 同时向 CS 请求数据, 如果表数据量很大且没有附带充分的过滤条件, 将会产生非常大的数据网络传输代价. 请求得到的数据在 join 算子处进行逐行 join 操作.

2.2 基于 semi-join 的 join 工作原理

图 2 为基于 semi-join 的 join 流程图, 其中, R 表作为驱动表, S 作为被驱动表, MS 是查询处理节点, CS 是数据存储节点.

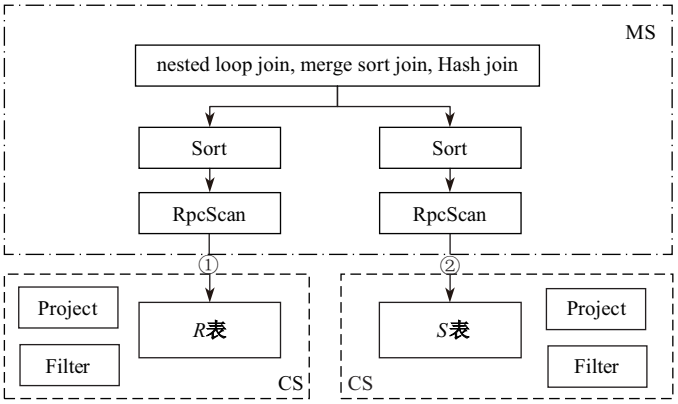

图 1 nested loop join、merge sort join、Hash join 执行计划

Fig. 1 Nested loop join, merge sort join, Hash join execution plan

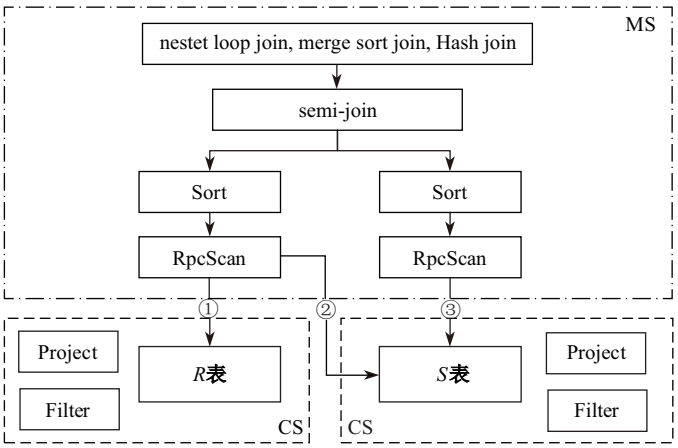

图 2 基于 semi-join 的 join 执行计划

Fig. 2 Execution plan of join based on semi-join

基于 semi-join 的 join 算法步骤如下.

(1) 向 R 表所在的 CS 并行发出请求, 获得数据, 这里获得的数据是经过 Project 和 Filter 过滤的、连接列上的、符合条件的数据.

(2) 将(1)中所得到的数据构造成过滤表达式发送给 S 表所在的 CS, 之后会在 S 表所在的 Server 上执行这个过滤操作, 过滤掉 S 表中不符合条件的数据.

(3) 右表的 RpcScan 操作符开始拉去过滤后的 S 表数据.

(4) R 表和 S 表过滤后的数据到达 join 操作符后, 进行串行化逐行 join.

3 并行优化设计

基于第 2 节描述的串行化逐行 join 情况, 我们提出并行 join 优化方案. 下面我们将对 OceanBase 数据库中的 Hash join、nested loop join 及 semi-join 进行并行优化设计.

3.1 Hash join 的并行优化设计

首先, 分配线程分别对两张表的数据调用 Hash 函数进行分区; 然后根据分片数申请处

理线程, 每个线程读取一组对应分片数据后做连接操作; 随后将输出结果发送给连接算子. 具体流程如图 3 所示.

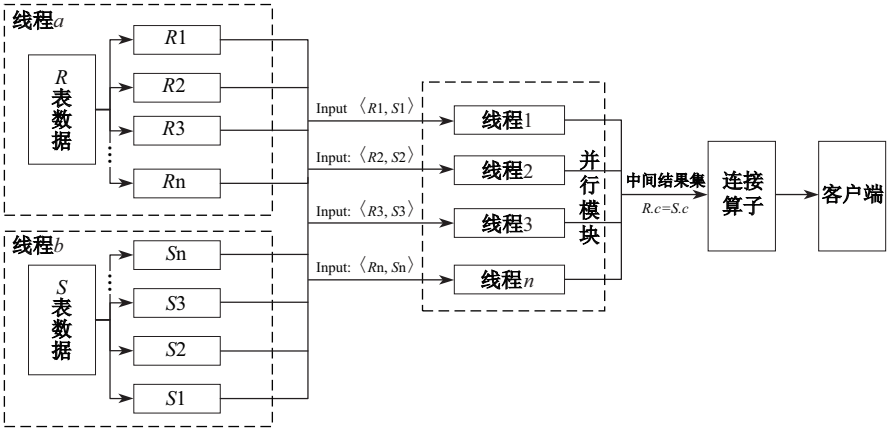

图 3　Hash join 的并行设计

Fig. 3　Parallel design of Hash join

1. 首先分配两个线程, 线程 a 和线程 b, 使用相同的 Hash 函数, 并行地对 R 表和 S 表数据根据连接列进行分区处理, 将 R 表分区成 $R1$, $R2$, $R3$,$\cdots$,Rn 等多个结果集, 同样将 S 表分成 $S1$, $S2$, $S3$,$\cdots$,Sn, 分区个数取决于具体的 Hash 函数. 由于 R 表与 S 表的数据集大小不同, 分区执行的时间也不同. 因此, 要等到两个线程分区工作都完成才能继续. 如果出现两张表大小相差很大时, 可以进一步分配多个线程对大表进行分区. 这样, 可以有效减少分区时间.

2. 由于使用的 Hash 函数相同, 对于分区后的结果集, 我们将 $R1$ 与 $S1$ 对应, $R2$ 与 $S2$ 对应, 其他类似.

3. 并行模块设置多个处理线程, 线程 1, 线程 2, 线程 3,$\cdots$,线程 n, 分别对应 2 中的一组分区. 每个线程并行地去各个存储节点拉取对应范围数据, 随后进行 join 处理, 例如, 线程 1 负责$\langle R1,\ S1\rangle$. 多个线程同步执行, 可以极大减少处理时间. 处理线程具体步骤如下, 以线程 1 为例.

(1) 对 $R1$ 分区的连接属性列, 使用 Hash 函数构造 Hash 表 T.

(2) 对 $S1$ 分区中的每一个连接列数据, 使用相同的 Hash 函数对(1)中生成的 Hash 表 T 进行检测.

(3) 如果 $S1$ 分区中的连接列数据落在 Hash 表 T 中, 则将对应行数据进行连接操作, 生成中间结果集, 并将结果集发送给连接算子操作符.

(4) 处理线程结束, 释放内存资源, 清空环境信息, 并重新挂起线程.

4. 连接算子操作符接收处理线程发送过来的中间结果集, 存在操作符内部的缓存区. 直到所有数据集全部处理完, 并接收到所有中间结果集, 然后发送给客户端.

以上为 Hash join 的并行实现流程. 并行设计体现在: 首先, 对于连接表数据读取进行并行操作; 然后处理线程并行执行连接操作.

3.2　nested loop join 的并行优化设计

与 Hash 类似, 首先, 分配线程对 R 表的数据进行切分; 然后根据分片数申请线程, 每个线程分别读取数据后将数据与 S 表做连接操作; 随后将输出结果发送给连接算子. 与 Hash

join 的区别在于无需对 S 表进行切分. 具体流程如图 4 所示.

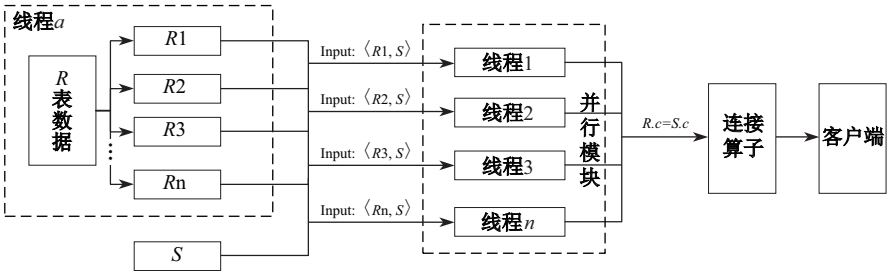

图 4 nested loop join 的并行设计

Fig. 4 Parallel design of nested loop join

1. 首先分配一个线程, 将 R 表分为等分的多个部分 $R1$, $R2$, $R3, \cdots, Rn$, 具体分为多少部分由 R 的结果集和可用的线程数确定.

2. 将 1 中 R 表等分产生的 $R1$, $R2$, $R3, \cdots, Rn$ 分别与 S 对应起来,例如 $\langle R1, S\rangle, \langle R2, S\rangle$.

3. 并行模块设置多个线程 $R1$, $R2$, $R3, \cdots, Rn$, 分别对应 2 中的一组分区. 每个线程并行地去各个存储节点拉取对应范围数据, 随后进行 join 处理, 例如, 线程 1 负责$\langle R1, S\rangle$, 多线程同步执行. 处理线程具体步骤如下, 以线程 1 为例.

(1) $R1$ 作为外层循环表, S 表作为内层循环表.

(2) 将 $R1$ 中的记录逐一与 S 表中的所有记录对比. 如果对应连接列值匹配, 则生成新的元祖作为中间结果集, 执行这一操作, 直到 $R1$ 表中数据全部遍历完, 并将结果集发给连接算子操作符.

(3) 处理线程结束, 释放内存资源, 清空系统环境, 重新挂起线程.

4. 处理算子操作符接收处理线程发送过来的中间结果集, 存在操作符内部的缓存区. 直到所有数据集全部处理完, 并接收到所有中间结果集, 然后发送给客户端.

以上为 nested loop join 的并行实现流程. 并行设计体现在: 首先, 多个线程并行地从存储节点读取相应范围的数据; 然后处理线程并行执行连接操作.

3.3 semi-join 的并行优化设计

首先, 分配线程对 R 表的数据进行切分; 然后根据分片信息申请线程, 每个线程分别读取数据并构造过滤表达式, 随后将输出结果发送给半连接算子. 具体流程如图 5 所示.

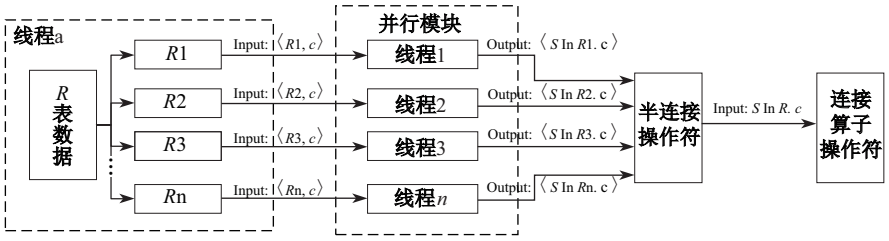

图 5 semi-jion 并行设计

Fig. 5 Parallel design of semi-join

1. 将 R 表结果集连接属性列等分为 $R1$, $R2$, $R3, \cdots, Rn$, 具体分为多少份取决于系统资源以及 R 表结果集大小.

2. 并行模块为 R 表结果集每个分片提供一个线程, 各个处理线程分别拉取对应范围数

据, 并根据连接属性列信息并行执行过滤操作. 此时的输出并非 join 结果集, 而是 S 表中符合过滤条件的元组. 随后提交给半连接操作符. 处理线程具体步骤如下.

(1) 根据输入 R 表分片信息构造过滤表达式. 这里可以根据数据输入的 R 表结果集大小考虑构造 In 表达式或者 Between 表达式, 具体构造哪种表达式取决于 R 表输入信息.

(2) 将过滤表达式发送给 S 表所在存储节点. S 表所在存储节点用过滤表达式根据连接属性列过滤 S 表数据.

(3) S 表所在存储节点将符合条件的数据作为输出结果集发送到半连接操作符.

3. 半连接接收处理线程陆续发来的结果集, 并将结果集存在操作符内部的缓存区. 当所有数据处理结束后, 半连接算子操作符将缓存中的数据发送给连接算子操作符. 此后, 连接算子操作符才开始真正的 join 操作.

以上为 semi-join 的并行设计方案. 并行设计体现在: 根据 R 表输入并行构造过滤表达式; 多线程并行读取各分片对应过滤表达式信息, 随后并行用过滤表达式过滤 S 表数据; 各连接算子并行做 join 操作.

4　实验分析

首先, 测试不同并行度下, 单表读取数据的效率; 其次, 测试在不同并行度下传统 join 算法的执行效率; 最后, 测试不同并行度下基于 semi-join 算法的传统 join 算法执行效率.

4.1　实验软件环境

本文选择 OceanBase 系统作为实验系统环境. 实验使用 5 台服务器, OceanBase 的实验版本为 0.4.2, 其中主控节点 RS 与 UPS 共用一台服务器, 其余 4 台服务器分别部署 CS 与 MS. 整个 OceanBase 数据库集群中 3 台 CS, 1 台 MS. 实验环境的OceanBase 集群物理拓扑如图 6 所示.

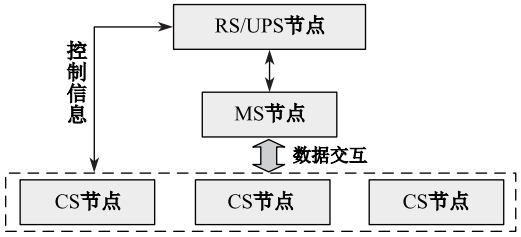

图 6　OceanBase 实验环境物理拓扑

Fig. 6　Physical topology of OceanBase experimental environment

4.2　实验硬件环境

本文测试所用硬件环境如表 1 所示, 其中磁盘为 SSD(Solid State Drive).

表 1　集群服务器配置

Tab. 1　The cluster server configuration

角色	CPU	内存/GB	磁盘/TB	网络
CS	6 核 12 线程(Intel(R)Xeon(R)CPU E5-2620 V2@2.10 GHz)*2	64	3	千兆网
UPS/RS MS	6 核 12 线程(Intel(R)Xeon(R)CPU E5-2620 V3@2.30 GHz)*2	165	1.5	千兆网

4.3　实验数据集

本文测试用到的所有数据表的模式如表 2 所示.

表 2 测试表的模式

Tab. 2 The schema of the test table

属性名称	是否为主键	数据类型	数据大小/Byte
ID	是	Int	4
Col 1	否	Varchar	64
Col 2	否	Varchar	64

所有实验数据都是由 Sysbench 的数据生成器生成. 实验一共涉及 7 张表, 数据分布以及数据量如表 3 所示.

表 3 测试数据表信息

Tab. 3 Test data table information

表名	数据分布(主键)	数据量(行数)/万
$R1$	连续	10
$R2$	连续	100
$R3$	连续	1 000
$S1$	连续	10
$S2$	连续	100
$S3$	连续	1 000

以 $R1$ 为例, 数据分布的连续性是指 $R1$ 表的主键列 ID 的数据是按照升序排列的.

4.4 不同并行度下的连接查询响应时间

实验目的: 单表情况下并行度以及数据过滤方式对查询响应时间的影响.

数据设置: 使用表 $R2$ 作为测试表, 结果集为 100 万, 并发度分别为 1、5、15、20、25、30、35, 数据过滤方式为主键定位、In 表达式、Between 表达式.

实验结果如图 7 所示.

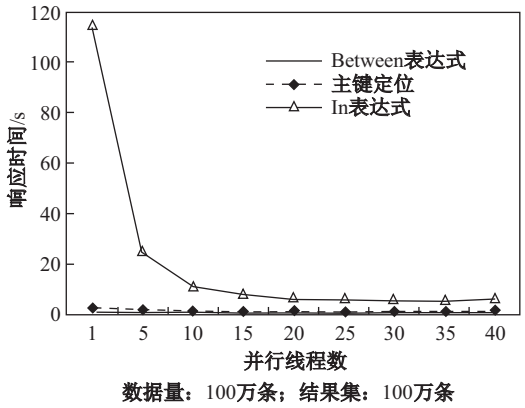

图 7 不同并行度单表数据查询的响应时间

Fig. 7 Response time for single table query with different parallel number

从实验结果可以得出, 随着并行度的增加, 响应时间总体呈下降趋势. 因此通过并行的方式来提高数据的过滤速度, 对减少连接查询的响应时间是有效的.

4.5 连接算子的执行效率

图 8 所示为在不同的并行度下, 各连接算法的执行效率. 在并行度为 1 的情况下, nested loop join 在处理 100 万条数据的连接计算时花费的时间在 78 s 左右. 由于与其他连接算子的

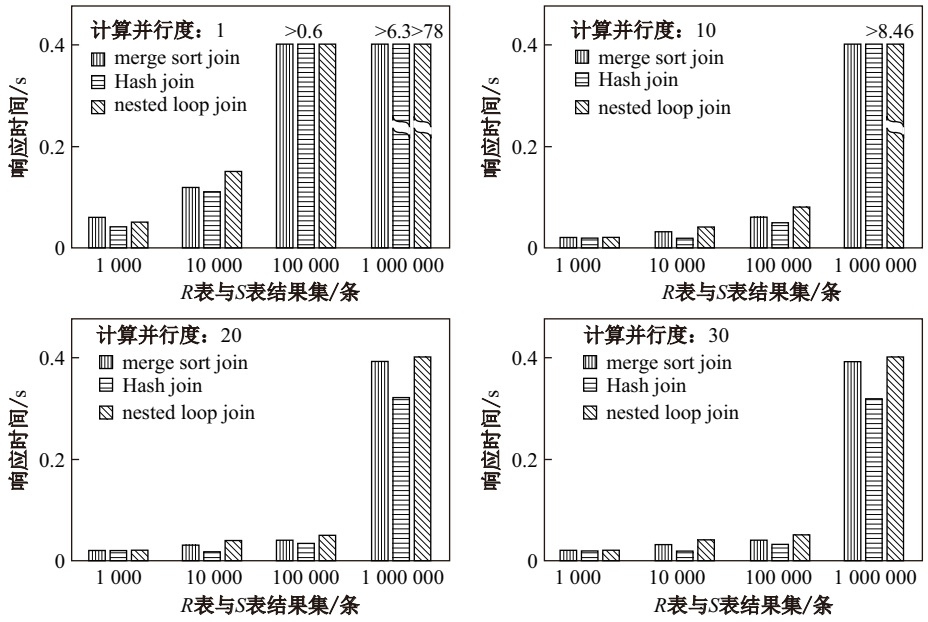

图 8　不同并行度下连接算法的执行效率

Fig. 8　Execution efficiency of join algorithms with different parallel number

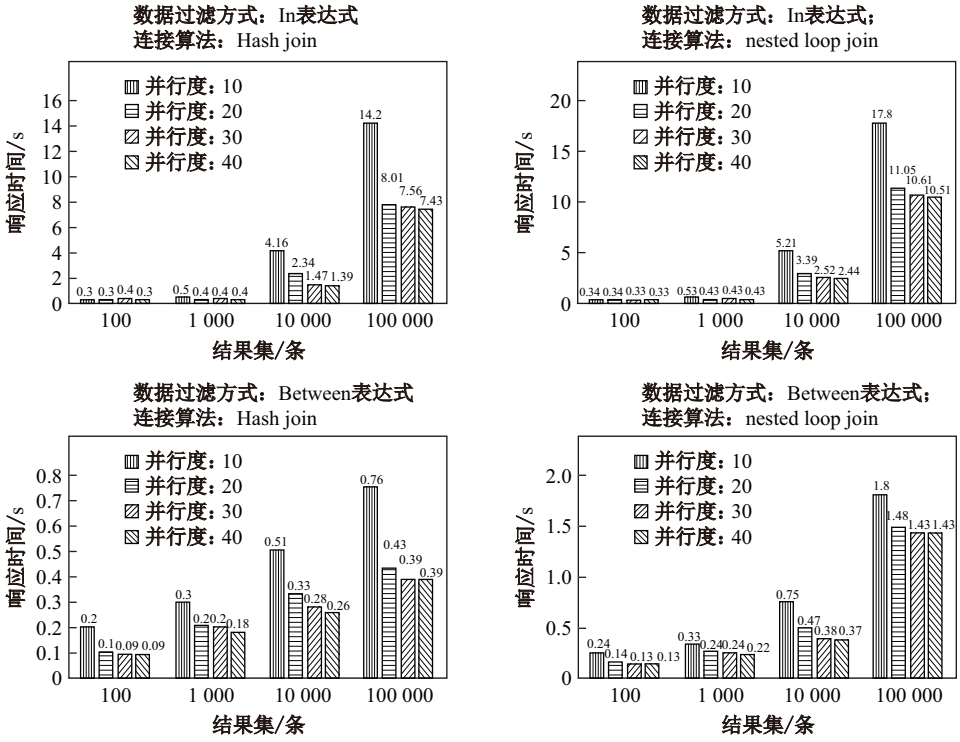

图 9　不同并行度下基于 semi-join 的 join 执行效率

Fig. 9　Execution efficiency of join based on semi-join in different parallel number

响应时间相差太大, 因此没有在图中完全显示. 随着并行度的增加, 各连接算子的响应时间都有所降低.

4.6 semi-join 下各 join 算法的查询效率

如图 9 所示,随着并行度的提高, Hash join、nested loop join 的响应时间都在逐渐降低. 特别地, 当结果集不断变大时响应时间的下降速度也随之加快. 但是当并行度超过 20 后, 响应时间的变化就变得不是非常明显, 原因在于数据库系统的计算能力受制于服务器 CPU 的核心数目; 而用于本文实验的服务器, 在采用超线程技术后, 可用的核心数目为 24 个, 当并行度超过 20 后就有明显的调度以及资源争用问题, 原因是, 一方面受 CPU 核心数目的限制, 另一方面本文也没有将任务线程与相应的 CPU 核心进行绑定, 因此可能出现多个任务线程共用一个核心的情况.

5 总 结

本文提出了一种在分布式数据库系统 OceanBase 中对传统 Hash join、nested loop join 等算法以及 semi-join 算法的并行连接优化方案, 并且在 OceanBase 中进行了实验验证. 该优化方案充分利用了分布式数据库多数据节点和服务器多核的特点. 实验结果验证了并行连接优化的有效性: 在服务器线程总数内, 传统 join 算法和基于 semi-join 的传统 join 算法执行效率都有所提高. 但是本文方案还有进一步优化的空间. 如何根据数据分布信息动态选择使用何种 join 算法、如何实现线程资源的极大化利用、如何实现不同查询处理节点间的交互、如何实现多个查询处理节点间的协同工作, 这些都将是以后研究的重点.

[参 考 文 献]

[1] 杨传辉. 大规模分布式存储系统[M]. 北京: 机械工业出版社, 2013.

[2] BERNSTEIN P A, GOODMAN N, WONG E, et al. Query processing in a system for distributed databases (SDD-1)[J]. ACM Transactions on Database Systems, 1981, 6(4): 602-625.

[3] ZHANG X F, CHEN L, WANG M. Efficient multi-way theta-join processing using MapReduce[J]. Proceedings of the VLDB Endowment, 2012, 11(5): 1184-1195.

[4] BLASGEN M W, ESWARAN K P. Storage and access in relational databases[J]. IBM Systems Journal, 1977, 16(4): 363-377.

[5] ZHOU J R, ROSS K A. Implementing database operations using SIMD instructions [C] //Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. 2002: 145-156.

[6] ZAHARIA M, CHOWDHURY M, FRANKLIN M J, et al. Spark: Cluster computing with working sets[C/OL]//Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. (2010-06-25) [2017-04-01]. https://www.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia. pdf?CFID=973306186&CFTOKEN=67460167.

[7] MERRETT T H. Why sort-merge gives the best implementation of the natural join[J]. ACM Sigmod Record, 1983, 13(2): 39-51.

[8] KIM C, PARK J, SATISH N, et al. CloudRAMSort: Fast and efficient large-scale distributed RAM sort on shared-nothing cluster[C]// ACM SIGMOD International Conference on Management of Data. ACM, 2012: 841-850.

[9] BABB E. Implementing a relational database by means of specialzed hardware[J]. ACM Transactions on Database Systems, 1979, 4(1): 1-29.

[10] BONCZ P A, ZUKOWSKI M, NES N. MonetDB/X100: Hyper-pipelining query execution[C/OL]//Proceedings of the 2005 CIDR Conference on Innovative Data Systems Research. 2005: 225-237 [2017-04-01]. https://www.researchgate.net/publication/45338800_MonetDBX 100_Hyper- Pipelining_Query_Execution.

(责任编辑: 李 艺)