文章编号:1000-5641(2014)05-0263-08

# Spark 上的等值连接优化

卞昊穹[1,2], 陈跃国[1,2], 杜小勇[1,2], 高彦杰[1,2]
(1. 数据工程与知识工程教育部重点实验室(中国人民大学);
2. 中国人民大学 信息学院, 北京 100872)

**摘要**: 等值连接作为数据分析中最常用、代价最高的操作之一,在 Spark 上的实现和优化与传统并行数据库有很大的差别,传统并行数据仓库中基于数据预划分的连接算法在 Spark 上难以实现,而目前被广泛采用的 Broadcast Join 和 Repartition Join 性能较差,如何提高连接性能成为基于 Spark 的海量数据分析的关键. 本研究将 Simi-Join 与 Partition Join 的优势相结合,并基于 Spark 上的特性提出了一种优化的等值连接算法. 代价分析和实验表明本算法比现有基于 Spark 的数据分析系统中的连接算法性能提升 1～2 倍.

**关键词**: Spark; SQL; 大数据分析; 等值连接; 内存计算
**中图分类号**: TP392 **文献标识码**: A **DOI**:10.3969/j.issn.1000-5641.2014.05.023

## Equi-join optimization on spark

BIAN Hao-qiong[1,2], CHEN Yue-guo[1,2], DU Xiao-yong[1,2], GAO Yan-jie[1,2]
(1. *Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE, Beijing* 100872, *China*;
2. *School of Information, Renmin University of China, Beijing* 100872, *China*)

**Abstract**: Equi-join is one of the most common and costly operations in data analysis. The implementations of equi-join on Spark are different from those in parallel databases. Equi-join algorithms based on data pre-partitioning which are commonly used in parallel databases can hardly be implemented on Spark. Currently common used equi-join algorithms on Spark, such as broadcast join and repartition join, are not efficient. How to improve equi-join performance on Spark becomes the key of big data analysis on Spark. This work combines the advantages of Simi-Join and Partition Join and provides an optimized equi-join algorithm based on the features of Spark. It is indicated by cost analysis and experiment that this algorithm is 1-2 times faster than algorithms used in state-of-the-art data analysis systems on Spark.

**Key words**: Spark; SQL; big data analysis; equi-join; in-memory computation

## 0 引 言

Spark 是继 Hadoop 之后出现的通用高性能并行计算平台,应用于大规模的分布式数

收稿日期:2014-06
基金项目:中国人民大学科学研究基金(中央高校基本科研业务费专项资金资助)(10XNI018)
第一作者:卞昊穹,男,博士研究生,研究方向为数据库. E-mail: bianhaoqiong@gmal.com.
通信作者:陈跃国,男,副教授,硕士生导师,研究方向为数据库、信息检索. E-mail: chenyueguo@gmail.com.

据处理. 在存储方面,Spark 采用了基于分布式共享内存的弹性分布式数据集(Resilient Distributed Datasets, RDD)[1]作为数据结构,同时兼容 HDFS,可将 HDFS 中已经存在的大量现有数据作为数据源加载到 RDD 中进行处理,并将必要的数据在内存中进行容错的缓存以支持高性能的迭代计算. 在计算模型方面,Spark 与 MapReduce 相似,但更加灵活,在 RDD 上提供了更加丰富的操作符及操作符间的组合方式.

由于 Spark 的高可扩展性和较高的数据处理性能[2],在大规模 SQL 查询分析正在被研究和应用. 目前基于 Spark 的开源 SQL 查询分析系统有 Shark[3]和 Spark SQL[4]. 在存储方面,无论 HDFS 还是 Spark RDD,为了实现良好的可扩展性,都采用了较简单的存储模型,将数据以大数据块的方式分布式存储在集群的各个节点上,不支持传统并行数据仓库中的数据预划分和数据索引;同时为了简化应用程序开发,Spark 与 Hadoop 等平台的存储分布性对上层应用透明,这使得基于这类平台的数据分析系统亦无法干预数据的分布.

等值连接是结构化数据分析中最为常见、代价最高的操作之一,在传统并行分析型数据库中对等值连接操作的优化大多基于数据预划分,无法在 Spark 中实现. 在 Shark 和 Spark SQL 中使用最多是 Broadcast Join 和 Repartition Join. Broadcast Join 局限性大,网络和内存开销大,计算复杂度较高;Repartition Join 则要求在查询执行时对参与连接的两表作重新划分,网络和内存开销也很大. 在结构化大数据分析中,事实表和维表的数据量可能都很大,为了提高数据分析的实时性,需要对现有的 Spark 上的等值连接算法进行优化.

本文针对大数据分析中事实表与维表,尤其是大维表的等值连接提出了基于 Spark 优化的连接算法. 该算法首先将事实表在连接属性上进行以数据块为单位的去重,然后将按块去重后的事实表与维表进行预连接,预连接通过将去重后的事实表和维表基于一致性哈希的原理进行划分,使之在集群中并行完成,避免了可能的数据倾斜问题,连接结果与事实表再进行一次组装得到最终连接结果. 经过理论分析和实验验证,本文算法较目前 Spark 上性能最好的 SQL 数据分析系统 Spark SQL[3]的连接性能提高了 1 - 2 倍,且具有很好的可扩展性.

## 1 算法总体描述

基于 Spark/MapReduce 的大数据分析中,常用的连接算法有 Simi-Join 及其变种(如 Per-split Simi-Join)、Broadcast Join、Repartition Join 等[5-6]. 其中 Simi-Join 和 Broadcast Join 局限性较大,通常性能较差,Repartition Join 适用性最好,在绝大多数情况下具有最好的性能. 但 Repartition Join 在 Spark 上具有如下的缺点.

(1) 需要在查询时对数据进行动态的重划分,通信量较大,尤其在宽表的情况下;

(2) 通过 Hash 函数划分到同一节点的很多事实表元组在外键上具有相同值,内存和计算资源消耗较大.

针对以上问题,本文提出了一种对现有的 Spark 上的连接的优化算法. 本文中所用到的符号描述如表 1.

**定义 1** 连接并行度:连接时连接属性哈希分区的个数.

算法的执行流程包括如下两个阶段.

阶段一:动态优化与预连接

**表 1　符号列表**

Tab. 1　Frequently used symbols

| 符号 | 描述 |
|---|---|
| *Fact* | 来自事实表的参与连接的元组集合 |
| *Dim* | 来自维表的参与连接的元组集合 |
| *Key* | *Fact* 与 *Dim* 做等值连接的连接属性值 |
| *Pd* | 预连接并行度 |
| *FactUK* | *FactRdd* 的连接属性值 *Key* 的无重集 |
| *JoinedUK* | 预连接、即 *FactUK* 与 *Dim* 连接的结果 |
| *getPd* | *Pd* 的计算函数 |
| $\alpha$、$\beta$、$\omega$ | *getPd* 中所用的三个参数 |
| $\varphi$ | 预连接时计算数据划分区间的采样率 |
| *sampleUK* | 预连接时计算数据划分区间的样本 |
| *NetCost* | 算法的网络 I/O 代价 |
| *MemCost* | 算法的内存空间代价 |
| *Size*… | 数据量 |

该阶段根据 *Fact* 和 *Dim* 的元组数量计算出连接的并行度 *Pd*，并计算出 *Fact* 的连接属性值 *Key* 的无重集 *FactUK*，*FactUK* 中的每一个元素除了包含 *Key* 外，还记录了 *Key* 对应于 *Fact* 元组的存储位置. 将 *FactUK* 与 *Dim* 根据并行度 *Pd* 进行哈希连接，得到结果 *JoinedUK*，这一过程为预连接. 由于 *FactUK* 与 *Dim* 的数据量都远小于 *Fact*，因此预连接的网络 I/O、内存开销和 CPU 开销远小于 *Fact* 和 *Dim* 的 Hash 连接. 本文中预连接所用的哈希函数借鉴了 Dynamo、Cassandra 等 NoSQL 数据库中用于数据分布的一致性哈希算法的思想[7-8]，避免了现有系统中所采用的简单哈希函数受到数据倾斜影响较大、造成连接时负载不均衡的问题，且查询时动态计算出的并行度更加合理，使得预连接的效率得到了进一步的提高. *JoinedUK* 中同样包含 *Key* 对应于 *Fact* 元组的存储位置.

阶段二：组装连接结果

该阶段根据 *JoinedUK* 中记录的 *Fact* 中元组的存储位置将 *JoinedUK* 与 *Fact* 进行组装，得到最终的连接结果. 这一阶段无须再进行网络 I/O 和内存、CPU 开销巨大的 Hash 连接. *JoinedUK* 的数据量不大于 *FactUK*，在部分实际的查询中，*JoinedUK* 的数据量远小于 *FactUK*，这一阶段仅需将 *JoinedUK* 根据 *Fact* 中元组的存储位置与 *Fact* 进行组装，代价很小.

## 2　算法细节描述

本文的等值连接算法在 Spark 中可以高效地实现. 下面对算法各阶段中的一些过程作细节描述.

### 2.1　事实表去重

*Fact* 中的元组需要在连接属性 *Key* 上进行去重(投影). 但是由于 *Fact* 是分布式存储在集群中的，需要对 *Fact* 进行 shuffle 和 reduce 才能完成去重，*Fact* 数据量巨大，这样去重的代价很高. 实验分析发现，由于实际数据分析中采用的数据块通常较大(64MB 整数倍)，块内包含的元组通常很多，加之数据的分布通常具有一定的局部性，因此重复数据以数据块为单位进行去重也可以达到较好的效果. 所以本算法的去重分为两个阶段，第一阶段先以数据块为单位进行去重，其结果参与预连接，在预连接的过程中，重复的连接数据会划分到相同的分区中，从而进一步去重. Spark 的 mapPartitions 操作[2]用于实现以数据块为粒度的

计算,可用于实现数据块级别的去重.

2.2 连接并行度设定

连接并行度是 Repartition Join 的一项重要的参数,该参数设置过大或过小都会一定程度地影响连接的性能.研究者在 Shark 的研究实现中提出了动态的参数优化[9],在 Hive 等 SQL-on-Hadoop 系统中,查询优化是查询执行之前基于数据的静态统计信息进行的[10],Shark 中在查询运行时的动态参数调整给 Shark 带来了性能的提升.本文中借鉴了动态优化的思想,将连接并行度的设定在连接过程中完成.因为在连接执行之前难以估计参与连接的实际数据量,在查询执行前的静态查询优化阶段难以准确地设定这一参数.本文中的连接并行度通过式(1)所示的函数 *getPd* 计算得出.

$$Pd = getPd(FactUK, Dim) = \min(FactUK.partitionNum + Dim.partitionNum, \omega) \quad (1)$$

其中,*partitionNum* 表示数据块的个数,$\omega$ 用于控制并行度的最大值,在实际的数据分析系统中,往往有多种数据分析和处理负载共享系统中的资源,控制并行度可以防止当连接操作对资源的过度侵占.实验发现,当集群中没有其他负载时,$\omega$ 取分配给 Spark 集群的 CPU 核数的 2 倍左右可以得到最好的连接性能;在 Spark 中数据集(RDD)的 *partitionNum* 可以快速获取.

2.3 基于一致性哈希的数据划分

经过连接属性上的去重之后,*FactUK* 中的元素个数和数据量都远小于 *Fact*,但 *FactUK* 和 *Dim* 的数据量仍然可能超过单机处理能力,且仍可能存在数据倾斜.在预连接中,本文借鉴了一致性哈希[7]的思想对 *FactUK* 和 *Dim* 进行划分,具体流程为如下.

(1) 对 *FactUK* 和 *Dim* 中的 *Key* 取并集(不去重),并根据采样率 $\varphi$ 进行采样,得到样本 *sampleUK*.

(2) 计算 *sampleUK* 中 *Key* 的 Hash 值(如果 *Key* 为整型,Hash 值可以是其本身),并将 Hash 空间划分为 *Pd* 个区间,保证每个区间中的样本数基本相等.

(3) 将 *FactUK* 和 *Dim* 按照(2)中计算出的区间 Hash 划分到各个区间中,这些区间的物理位置分布在集群的多个节点上.

Spark 中支持 RDD 上的 union 和 sample[2].其中 union 是两个元素类型相同的 RDD 上的并操作,该操作并不会移动两个 RDD 的数据,也不会进行去重,只是将 RDD 的元数据合并;sample 是 RDD 上的采样操作,在各个 partition 上并行进行.这两个操作代价都很低,可快速完成(1);而利用 Spark 中的 RangePartitioner[2]可以快速完成(2)和(3).

2.4 预连接

*FactUK* 和 *Dim* 一致性哈希划分之后,相关的集群节点并行地对各个分片进行连接,得到连接结果 *JoinedUK*,其中每个元素包含了一个 *Key* 和 *Dim* 中对应于该 *Key* 的元组,以及该 *Key* 对应的 *FactRdd* 分区号的列表.由于划分后的数据缓存在各个集群节点的内存中,且充分利用了多节点和节点上多核的并行计算能力,预连接的速度很快.预连接的目的是进一步去重并排除没有连接结果的 *Key*.在实际的大规模数据分析中,相当一部分表连接查询中,都存在大量的 *Key* 上没有连接结果,浪费了大量网络 I/O 和计算资源.

Spark 中的 cogroup 操作可以实现将两个数据集按照 *Key* 快速分组,对分组结果集作简单的过滤操作即可保留有连接的 *Key*,完成预连接.

2.5 连接结果组装

预连接结果 *JoinedUK* 和 *Fact* 再做一次连接即可得到最终的连接结果.但这样需要广

播 *JoinedUK*,代价较高. 本文根据 *JoinedUK* 中包含的 *Key* 对应的 *Fact* 的分区号,将 *JoinedUK* 按照 *Fact* 的分区号做重新的划分,由于 *JoinedUK* 的数据量较小,划分代价较低. 划分后的结果和 *Fact* 具有相同的分区数且各分区和 *Fact* 一一对应,因此组装的过程以分区为粒度在集群中并行完成,效率很高. 组装完成后即得到最终的连接结果,可继续在连接结果上完成查询计划中的其他操作.

在 Spark 中通过自定义的 Partitioner 可以快速将 *JoinedUK* 按照 *Fact* 的分区号进行划分,划分得到的结果集具有和 *Fact* 相同的分区(数据块)数,之后通过 Spark 中的 zipPartitions 操作将 *Fact* 和 *JoinedUK* 做快速的组装,这一过程无需在网络上传输 *Fact* 中的数据.

## 3　算法分析

Spark 是基于分布式共享内存的分布式计算平台,在数据的处理过程中,数据可以始终保持在内存中. 本文假设参与连接的事实表和维表上的中间结果数据可以被缓存在内存中,因此,连接的代价主要来自网络 I/O、内存空间占用和 CPU 计算,这也是内存计算中性能考量的三个主要方面.

### 3.1　代价模型

本文的代价模型从网络 I/O 代价、计算复杂度、内存空间代价三个方面建立的.

1. 网络 I/O 代价

本文的连接算法中,事实表去重和计算连接并行度过程中几乎没有网络 I/O 的开销,网络 I/O 开销来自预连接过程的数据划分和连接结果的组装,如式(2)所示.

$$NetCost = NetCost_{jp} + NetCost_{a} \tag{2}$$

其中 $NetCost_{jp}$ 是预连接数据划分(RangePartition)的网络 I/O 代价,用数据量表示为式(3).

$$NetCost_{jp} = Size(FactUK) + Size(Dim) \tag{3}$$

由于去重后的 *FactUK* 中元组个数少于 *Fact*,且每个元组仅包含一个 *Key* 值及其对应的分区号,因此 *FactUK* 的数据量远小于 *Fact*,即 $Size(FactUK) = \alpha \cdot Size(Fact)$,$0 < \alpha \ll 1$. *Dim* 的数据量也远小于 *Fact*,即 $Size(Dim) = \beta \cdot Size(Fact)$,$0 < \beta \ll 1$.

$NetCost_{a}$ 是结果组装的网络 I/O 代价,其中包括将预连接进行一次划分和在组装过程中跨节点读取对应分区的通信代价,而这两个子过程的通信代价不会超过 $NetCost_{jp}$,通常由于预连接之后相当于对 *Fact* 在连接属性上做了全局的去重且排除了不能连接的元组,$NetCost_{a}$ 会远低于 $NetCost_{jp}$. 假设参与连接的节点数为 $N$,则总的通信量将由这些节点分摊,因此本文算法的网络 I/O 代价估算结果如式(4)所示.

$$NetCost \approx NetCost_{jp} \leqslant \frac{(\alpha+\beta) \cdot Size(Fact)}{N}, \quad 0 < \alpha+\beta \ll 1 \tag{4}$$

2. 计算复杂度

算法中计算分区、Hash 探索、判断等值等基本运算的执行次数都是和数据规模呈线性关系的. 假设 *Fact* 中的元组数为 $n$,*Dim* 中的元组数为 $m$,由于 $n \gg m$,则算法的计算复杂度为 $O(n)$.

3. 内存空间代价

算法中,除了事先缓存在内存中的参与连接的数据外,还需要缓存一些中间结果以加快计算,且连接过程中,需要将参与连接的数据在内存中建立 Hash 表、进行哈希探索,故需要

一定的内存空间消耗.

由于 *FactUK* 需要用于采样以确定预连接时的 Hash 空间划分并且要参与预连接,在 Spark 中对多次使用的数据进行缓存可以提高计算的效率,因此在算法实现中对 *FactUK* 进行了缓存. 此外在预连接中需要将划分后的 *FactUK* 和 *Dim* 在内存中建立 Hash 表,完成连接. 虽然这两个过程是依次进行的,但由于 Spark 的机制问题,在预连接开始之前,被缓存的 *FactUK* 数据难以被及时释放,故算法的内存代价应为两者之和. *FactUK* 通常比 *Fact* 小很多,所以算法的内存代价如式(5)所示.

$$\begin{aligned} MemCost &= 2 \cdot Size(FactUK) + Size(Dim) \\ &\leqslant 2\alpha \cdot Size(Fact) + Size(Dim) 0 < \alpha \ll 1 \end{aligned} \tag{5}$$

3.2 对比分析

基于 Spark/MapReduce 的大数据分析常用的连接算法中,Simi-Join 及其变种性能较差,适用性较差,仅在特殊的条件下才使用,目前在 Spark 上没有其系统实现,因此本文不做与 Simi-Join 的对比分析. 而 Broadcast Join 也仅适用于维表很小的情况下,在 Spark SQL 等基于 Spark 的大数据分析系统中并不常用. 假设事实表 *Fact* 中的元组数为 $n$,每个数据块中存储的元组数为 $b_f$,维表 *Dim* 中的元组数为 $m$,每个数据块中存储的元组数为 $b_d$,其中 $b_f$ 和 $b_d$ 是常数. 则有:

(1) Broadcast Join:的每个节点的平均网络通信量为 $Size(Dim)$,远大于本文算法;计算复杂度为 $O\left(\frac{mn}{b_f}\right) \rightarrow O(mn)$;所需内存空间等于维表的数据量乘以事实表的数据块数,即 $\frac{Size(Fact) \cdot Size(Dim)}{bs}$,其中 $bs$ 为事实表数据块大小,通常为 64 MB 整数倍.

(2) Repartition Join:在查询执行时需要将维表和事实表的数据进行划分,所以每个节点的网络通信量为 $\frac{(Size(Fact) + Size(Dim))}{N}$;计算复杂度同样为 $O(n)$;所需内存空间为 $Size(Fact) + Size(Dim)$.

本文算法与 Broadcast Join、Repartition Join 的对比如表 2 所示.

**表 2 等值连接算法代价对比**

Tab. 2 Cost comparison of equi-join algorithms

| 等值连接算法 | 网络 I/O 代价 | 计算复杂度 | 内存空间代价 |
|---|---|---|---|
| Broadcast Join | $Size(Dim)$ | $O(mn)$ | $\frac{Size(Fact) \cdot Size(Dim)}{bs}$ |
| Repartition Join | $\frac{(Size(Fact) + Size(Dim))}{N}$ | $O(n)$ | $Size(Fact) + Size(Dim)$ |
| 本文算法 | $\leqslant \frac{(\alpha + \beta) \cdot Size(Fact)}{N}$ | $O(n)$ | $\leqslant 2\alpha \cdot Size(Fact) + Size(Dim)$ |

可见,网络 I/O 代价方面,由于 $N$ 为参与连接的节点数,通常与集群规模有关,是一个较大的正整数,而 $0 < \alpha + \beta \ll 1$,所以本文算法网络 I/O 代价远低于 Broadcast Join 和 Repartition Join;计算复杂度方面,本文算法和 Repartition Join 为同一数量级,低于 Broadcast Join;内存空间代价方面,由于 $\frac{Size(Fact)}{bs}$ 为事实表数据块数,通常是一个较大的正整数,所以本文算法的内存空间代价远低于 Broadcast Join 和 Repartition Join.

## 4　实　　验

本文在 Spark 上实现了所述的连接算法，并以 TPC-DS[11-12] 作为测试基准来测试连接算法的性能，与 Spark SQL 和 Shark 进行了对比.

### 4.1　实验环境

实验的软件环境如表 3 所示.

**表 3　软件环境**

Tab. 3　Software environment

| 操作系统 | Hadoop | Spark | Shark | JDK |
|---|---|---|---|---|
| CentOS 6.4 - x64 | Apache Hadoop 2.2 | Apache Spark 1.0(本文算法和 Spark SQL)<br>Apache Spark 0.8(Shark) | 0.8 | OpenJDK 1.7 |

注：由于本文实验时，Shark 最高的稳定版本为 0.8，只能稳定运行在 Spark 0.8 上

本文的实验在实验室的云平台上完成，所用虚拟机集群的配置如表 4 所示.

**表 4　集群配置**

Tab. 4　Cluster settings

| CPU | 内存 | 磁盘 | 网络 | 节点数 |
|---|---|---|---|---|
| 4 核 2.5 GHz(其中 3 个分配给 Spark) | 16 GB(其中 14 GB 分配给 Spark) | 500 GB | 1GBi Ethernet | 8 / 16 |

所用的测试数据为 TPC-DS 100GB 和 300GB 数据集中最大的一张事实表 Store_Sales 和最大的一张维表 Customer，存储在 HDFS 上. 测试所用的两张表的数据量占数据集总数据量的 40%左右，如表 5 所示. Store_Sales 中的外键 ss_customer_sk 与 Customer 中的主键 c_customer_sk 可以连接，连接时每个表除联接属性外，各使用了 6 个 32 位整型列.

**表 5　测试表的数据量**

Tab. 5　Data volume of testing table

| 数据集 | Store_Sales | Customer |
|---|---|---|
| TPC-DS(100 GB) | 288 M 行、38.15 GB | 2M 行、257.03 MB |
| TPC-DS(300 GB) | 864 M 行、115.86 GB | 5M 行、647.34 MB |

### 4.2　实验结果

**定义 2**　*Fact* 与 *Dim* 连接选择率：*Dim* 中能与 *Fact* 产生连接结果的元组数占 *Dim* 总元组数的比率.

在实验中，取连接选择率为 70%，采用 TPC-DS 300GB 数据集，集群节点数为 16，每个节点给 Spark 分配 3 个核，测试连接并行度 *Pd* 对连接执行时间的影响，结果如图 1 所示.

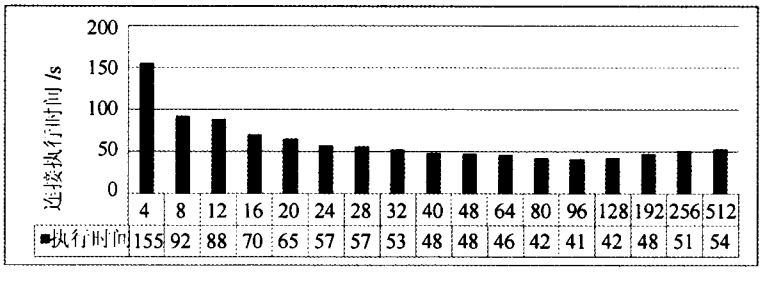
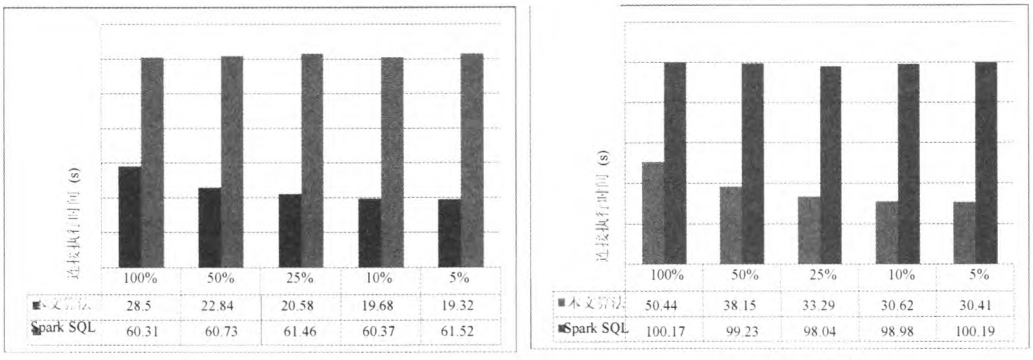



图 1　连接并行度对连接执行时间的影响

Fig. 1　Impact of parallelism degree on join execution time

可见当 $Pd$ 取分配给 Spark 集群的 CPU 核数的 2 倍,即 96 时可以得到最好的连接性能,并且 $Pd$ 在该值周围时,查询执行时间变化不大.

在实验中,取 $Pd$ 为分配给 Spark 集群的 CPU 核数的 2 倍.通过人为加入随机因数改变两表的连接选择率来验证在不同的连接选择率下连接算法的性能.实验结果如图 2 所示.




(a) 8节点、100GB数据集

(a) 8 nodes, 100GB dataset

(b) 16节点、300G数据集

(b) 16 nodes, 300GB dataset

图 2 连接执行时间对比

Fig. 2 Join execution time comparison

图 2 中的执行时间是 5 次执行取得的平均值,单位为 s,保留小数点后两位.根据查询计划分析,Spark SQL 中使用的是 Repartition Join 算法,而 Shark 中使用的是 Broadcast Join 算法.实验中测试了不同连接选择率下的连接操作执行时间,Shark 的执行时间全部超过 1 800 s 或者报错,因此执行时间没有标注在图中. Spark SQL 使用的 Repartition Join 对连接选择率不敏感.本文中的连接算法随着连接选择率的下降,执行时间有所下降,并趋向一个稳定值,这个稳定值是算法执行过程中预连接的执行时间.在最坏情况下,即连接选择率为 100%时,本文连接算法的执行速度仍然比目前性能最好的 Hash Join 高出 1 倍.

通过图 2 (a)和(b)的对比,当集群中节点数量增加 1 倍(8 个增加到 16 个),数据量增加 2 倍(288 M 行到 864 M 行),扩展性最理想情况下执行时间增加 50%,本文连接算法的平均执行时间(5 种连接选择率下执行时间的平均值)增加 63.7%,而 Spark SQL 的 Repartition Join 平均执行时间增加 63.1%,可见本文的算法和 Repartition Join 都具有较好的可扩展性.

## 5 结 论

随着 Spark 等大规模集群式的内存计算框架在大数据分析中的普及,交互式的大数据分析成为必然的趋势.连接性能是交互式大数据分析的主要瓶颈.本文对 Spark/Hadoop 上现有的等值连接算法进行了分析研究,提出了一种改进的等值连接算法.该算法具有很好的适用性,在事实表和大维表的连接中表现出良好的性能,比现有系统中的连接算法性能高出 1～2 倍.并且随着连接选择率的降低,算法的性能会进一步提高.

本文的算法可以作为等值连接的操作符加入到现有的 Spark SQL 等数据分析系统中,提高现有数据分析系统的性能.

(下转第 280 页)

Systems (PDCS 2009). Innsbruck, Austria (February 2009).
[7] 申德荣，于戈，王习特，等. 支持大数据管理的 NoSQL 系统研究综述[J]. 软件学报，2013，8：008.
[8] POKORNY J. NoSQL databases: a step to database scalability in web environment[C]//The 13th International Conference on Information Integration and Web-based Applications & Services (iiWAS). 2011: 278-283.
[9] BROWN A, WILSON G. The Architecture of Open Source Applications[M]. Lulu. com, 2011.

(责任编辑 李 艺)

(上接第 270 页)

## [参 考 文 献]

[1] ZAHARIA M, CHOWDHURY M, FRANKLIN M J, et al. Spark: cluster computing with working sets[C]// HotCloud2010. USENIX Association Berkeley, CA:[s. n.], 2010: 10-10.
[2] Spark[OL]. http://spark. apache. org/.
[3] Shark[OL]. http://shark. cs. berkeley. edu/.
[4] Spark SQL[OL]. http://spark. apache. org/sql/.
[5] BLANAS S, PATEL J M, ERCEGOVAC V, et al. A comparison of join algorithms for log processing in MaPreduce[C]//SIGMOD2010. New York: ACM, 2010: 975-986.
[6] SAKR S, ANNALIU, FAYOUMI A G. The Family of MapReduce and Large-Scale Data Processing Systems[J]. ACM Computing Surveys (CSUR), 2013, 46(1).
[7] KARGER D, LEHMAN E, LEIGHTON T, et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide Web[C]//STOC97. New York :ACM, 1997: 654-663.
[8] DECANDIA G, HASTORUN D, JAMPANI M, et al. Dynamo: Amazon's highly available key-value Store[C]// SOSP2007. New York: ACM, 2007: 205-220.
[9] XIN R S, ROSEN J, ZAHARIA M, et al. Shark: SQL and rich analytics at scale[C]//SIGMOD2013. New York: ACM, 2013: 13-24.
[10] THUSOO A, SARMA J S, JAIN N, et al. Hive :a warehousing solution over a map-reduce framework[J]. PVLDB, 2009, 2(2): 1626-1629.
[11] OTHAYOTH R, POESS M. The making of TPC-DS[C]//VLDB2006. New York: ACM , 2006: 1049-1058.
[12] POESS M, NAMBIAR R O, WALRATH D. Why you should run TPC-DS:a workload analysis[C]//VLDB2007. New York: ACM , 2007: 1138-1149.

(责任编辑 李 艺)