尊敬的评审专家：

您好！首先，衷心感谢评审专家提出的宝贵意见。针对各位专家提出的问题，我们进行了认真的修改，详细修改说明如下（**修改文字详见论文中高亮部分**）：

修改意见 1：当 MapReduce 程序的 Map 任务不是 I/O 密集型时，在 Reduce 阶段依然会产生大量 I/O 操作，此时 Reduce 程序的负载均衡问题就不能再简单地用文中所提方法，作者的方法需要改进。

修改结果：已修改

本文提出的负载均衡方法本质上解决的是 shuffle 阶段 partition 函数分区不均导致的 Reduce 任务负载不均衡、从而影响连接查询的整体执行效率问题。对于评审专家所描述的“当 Map 任务不是 I/O 密集型时，在 Reduce 阶段依然会产生大量 I/O 操作”的情况，本文提出的负载均衡方法基于的 Reduce Join 就是符合这种情形的一种连接算法，但原文没有描述清楚。因此，在修改文中我们做了进一步的描述，以便读者能够清楚了解本文提出的负载均衡方法。修改文字如下（对应文中标蓝部分）：

Reduce Join[14]的 Map 阶段仅负责将参与连接的数据表中的记录解析成 Key-Value 对（此时 I/O 操作很少），并通过 Shuffle 阶段传输到对应的 Reduce 任务中，而真正的连接操作是在 Reduce 阶段中完成的（此时会产生大量 I/O 操作）。考虑到 I/O 代价是影响连接查询的主要因素，我们对产生大量 I/O 操作的 Reduce 阶段进行读写分析，综合考虑 Reduce 任务的输入和输出代价及其对应的读写权重，最终基于这一综合代价给出了 Reduce 任务的负载均衡方法。

另外，本文以等值连接为例对负载均衡方法进行描述，但该方法并不仅限于此，还适用于其他连接，也可以扩展到连接以外的其它类型作业。例如，进行近似连接时，只需将连接属性值 a(键值对中的键)替换为一个满足近似条件(如$|a_1-a_2|\leqslant\delta$)的二元组$<a_1, a_2>$，并将负载贡献值的计算公式中 f_{1a} 和 f_{2a} 分别替换为 R_1 中连接属性值为 a_1 的元组个数以及 R_2 中连接属性值为 a_2 的元组个数。执行 Replicated Join 时，键值对中的 *key* 将会变成多个连接属性构成的多元组。对于连接以外的其他作业，我们可以分别对 Reduce 任务输入数据和输出数据的处理代价函数进行适应性的更改（论文 3.2 节中已有相应描述）。

[14] Blanas S, Patel J M, Ercegovac V, et al. A comparison of join algorithms for log processing in MaPreduce [C]// Proceedings of the ACM SIGMOD International Conference on Management of Data. Indianapolis, IN, US, 2010: 975-986.

修改意见 2：I/O 代价函数依赖于 C_1，C_2 和 C_3，即作者所提及的本地、网络和 HDFS 的 I/O 代价权重，但没有给出确定这三个权重的方法，以及他们的大小关系。根据实践经验，C_3 一般远大于 C_1，并且通常配置的 HDFS 的 replicated block 都大于 1，因此 C_3 会远大于 $C_1*n+C_2*(n-1)$（即本地存储一份，然后通过网络传输到至少 n-1 个 datanode 备份保存 1 次），而 C_2 与 C_1 的大小关系要根据具体的硬件配置而定。作者可以根据常见硬件配置、HDFS 配置参数等确定的因子，给出 C_1，C_2 和 C_3 的参数，在实践可以针对特定环境提供更准确的优化。

修改结果：已修改。

针对评审专家提出的建议，我们增加了 C_1，C_2 和 C_3 的确定方法以及在本文实验环境下的具体值，详细修改说明如下（对应文中标紫部分）：

I/O 代价函数中的三个系数 C_1、C_2 和 C_3 分别为算法的本地 I/O、网络 I/O 和 HDFS I/O 的代价权重，三者均为系统硬件相关的参数，其中 C_3 还与 HDFS 的副本个数设置有关。因此，我们可以事先通过文件读写实验测出这三个参数的具体值，而后带入 I/O 代价函数中进行计算。在我们的实验环境中，HDFS 的副本个数设为 3（常用副本个数），通过文件读写实验测得的三个参数值分别为 C_1=3.67(s/GB)、C_2=8.93(s/GB)和 C_3=13.37(s/GB)，三者之间的比值近似为 1:2.4:3.6。

从中可以看出，C_3 确实如评审专家所说，远大于 C_1 和 C_2。另外，我们还测试了副本个数为 10 时的 C_3 值：C_3=14.63(s/GB)，从中可以看出，C_3 的确与副本个数有关，但是因为 HDFS 进行写操作的时候副本之间的复制是通过管线进行流式传输的[1]，因此副本个数对 HDFS I/O 的代价权重 C_3 的影响程度是随着副本个数的增多而逐渐减小的，并非线性增加。综上所述，在实际应用中，我们可以针对特定环境的系统硬件配置以及 HDFS 的配置参数等（如副本个数）来通过文件读写实验确定 C_1，C_2 和 C_3 更精确的参数值。

[1] White T. Hadoop 权威指南(第二版)[M]，北京：清华大学出版社，2011，p66.

修改意见 3：实验中的 R_1，R_2 数据量略显不足，对于大数据应用，应该在 R_1，R_2 超过 10 亿条数据量的情形下进一步验证算法的有效性。

改结果：已修改。

评审专家的意见非常准确，文中原来给出的数据量对于大数据应用来讲的确略显不足，对此我们已将数据表 R_1 和 R_2 的数据量分别增大到 10 亿条和 20 亿条，而后重新设计实验，并在 4.2 节中给出相应的实验结果及分析（文中标绿部分）。通过实验结果可以看出，在大数据应用下，文中提出的优化方法依然有效，不但能够很好地均衡所有 Reduce 任务的负载，从而提高连接查询的整体效率，同时还能够确定最佳的 Reduce 任务个数。

修改意见 4：该文理论研究具有较好的创新性，且实验验证结果显示该文的优先方法是很有效的，从而具有很好的应用价值。该文结构合理，可读性良好，语句通顺，但摘要可以稍加简练一些。

修改结果：已修改。

针对您提出的宝贵意见，我们已对本文摘要做了进一步的精炼，对应英文也已做修改（文中标黄部分）。

最后，再次对各位评审老师和编辑部的老师表示衷心的感谢。由于水平有限，我们的文章以及对于问题的回答都可能出现错误和遗漏，请各位老师不吝指正。

祝各位老师工作顺利！

此致

敬礼

文章作者

2014-11-15

ISSN……
XX(X)：XX-XX，2014

基于 MapReduce 的多元连接优化方法

李甜甜[1] 于戈[1] 郭朝鹏[2] 宋杰[2]
[1]东北大学 信息科学与工程学院，沈阳 110819
[2]东北大学 软件学院，沈阳 110819
(litiantian_neu@163.com)

Multi-way Join Optimization Approach based on MapReduce

Li Tiantian[1] Yu Ge[1] and Guo Chaopeng[2] Song Jie[2]
[1](School of Information Science and Engineering, Northeastern University, Shenyang, 110819)
[2](Software College, Northeastern University, Shenyang, 110819)

Abstract Multi-way join is one of the most common data analysis operations, and MapReduce programming model that has been widely used to process large scale data sets has brought new challenges to multi-way join optimization: traditional optimization approaches cannot be simply adapted to fit MapReduce feature; there is still optimization room for MapReduce join algorithm. As to the former, we think I/O is the main cost of join. This paper first proposes an I/O cost based heuristic algorithm to initially determine a join sequence, and conducts further optimization. After the optimization, we also design a parallel execution algorithm to improve the whole performance of multi-way join. As to the latter, we think load balancing can effectively decrease the "buckets effect" of MapReduce. This paper proposes a fair task load allocation algorithm to improve the intra-join parallelism, and also analyzes the method to decide the appropriate number of reduce tasks. Experiments verify the effectiveness of the proposed optimization approaches. This study contributes to multi-way join applications in big data environment, such as the star-join in OLAP and the chain-join in social network.

Key words multi-way join; execution plan; I/O cost; performance optimization; MapReduce; load balancing

摘要 多元连接是数据分析最常用的操作之一，MapReduce 是广泛用于大规模数据分析处理的编程模型，它给多元连接优化带来新的挑战：传统的优化方法不能简单地适用到 MapReduce 中；MapReduce 连接执行算法尚存优化空间。针对前者，考虑到 I/O 代价是连接运算的主要代价，本文首先以降低 I/O 代价为目标提出一种启发式算法确定多元连接执行顺序，并在此基础上进一步优化，最后针对 MapReduce 设计一种并行执行策略提高多元连接的整体性能。针对后者，考虑到负载均衡能够有效减少 MapReduce 的“木桶效应”，本文通过任务公平分配算法提高连接内部的并行度，并在此基础上给出 Reduce 任务个数的确定方法。最后，通过实验验证本文提出的执行计划确定方法以及负载均衡算法的优化效果。本研究对大数据环境下 MapReduce 多元连接的应用具有指导意义，可以优化如 OLAP 分析中的星型连接，社交网络中社团发现的链式连接等应用的性能。

关键词 多元连接；执行计划；I/O 代价；性能优化；MapReduce；负载均衡

中图法分类号 TP393

连接运算根据连接条件把两个或多个关系中的记录组合为一个结果数据集，包含连接运算的查询简称为连接查询。连接查询在数据分析中很常见，TPC-H 提供的 22 个查询用例中有 16 个涉及到此类查询[1]。当一个连接查询涉及 n 个关系时，称为 n 元连接；当 n>2 时，称为多元连接，多元连接是数据分析

收稿日期：2014-11-15
基金项目：国家自然科学基金重大项目（No.61433008）和青年基金（No. 61202088）；中国博士后科学基金面上项目（No. 2013M540232）；中央高校基本科研业务费专项资金（No. N120817001）；教育部博士点基金（No. 20120042110028）。

中最常用的操作之一。此外，在当今的大数据环境下，MapReduce[2]编程模型被广泛用于大规模数据集的分析处理。目前，MapReduce 中数据分析的优化工作包括索引、数据布局、查询优化、迭代处理、公平负载分配以及交互式处理等方面[3]。基于此，我们分析 MapReduce 给多元连接的优化带来的新挑战。

首先，多元连接查询依赖良好的执行计划。传统的执行计划确定方法[4]不满足 MapReduce 特性，无法通过简单的适应性更改应用到现有的优化中。另外，现有基于 MapReduce 的执行计划确定方法[5, 6]复杂度较高，应用范围受限。因此，亟需提出一种新的满足 MapReduce 特性且复杂度较低的执行计划确定方法。此外，我们还注意到，无论是传统研究还是现有研究，其执行计划都只确定了连接的执行顺序，并未考虑无依赖关系的连接操作间的并行执行策略。

其次，良好的执行计划固然重要，对执行框架的优化同样可以有效地提高多元连接的性能，这一点在分布式环境中尤为重要。一种公平的并行任务负载分配方法可以有效地减少 MapReduce 中的“木桶效应”，从而提高连接操作内部的并行度。然而，就我们所知，目前没有针对连接运算的 MapReduce 负载均衡方法，且现有的通用方法[7-9]仅考虑了任务的输入代价，不适用于连接运算，因为它的输出代价也不可忽略。

本文研究 MapReduce 环境下多元连接的优化方法，基于上述分析，我们提出以下问题：①多元连接执行计划的解空间很大，短时间内很难找到最优解，那么能否通过某个复杂度较小的算法快速找到一个近似最优解；②连接运算属于 I/O 密集型运算，I/O 为主要代价，那么能否针对 MapReduce 特性提出 I/O 代价模型，并选择代价最小的执行计划；③连接顺序确定后，不存在依赖关系的连接操作可以并行执行，那么当存在多个满足并行执行的连接操作时该如何选择；④执行框架的优化中，负载均衡能够有效地减少“短板效应”，那么此处的连接负载又该如何定义。这些问题的求解存在一定程度的挑战，就我们目前所知，尚未发现能够完全解决上述问题的研究工作。

本文首先通过分析多元连接执行计划解空间的缩减方法、MapReduce 连接算法的 I/O 代价模型、Replicated Join[1]的优劣以及 MapReduce 作业的并行执行特性，最终确定了执行计划的优化方法；接着，结合 MapReduce 框架分析连接运算的特性，提出负载均衡模型及其对应的均衡算法，并在此基础上提出 Reduce 任务个数的确定方法。大量实验验证了本文提出的优化方法的有效性。

本文组织结构如下：继引言之后第 1 节介绍了本文的相关工作；第 2、3 节分别给出了多元连接执行计划的优化方法以及基于 MapReduce 框架的连接负载均衡方法；第 4 节通过大量实验对本文提出的优化方法进行验证；第 5 节对本文进行了总结。

1. 相关工作

现有多元连接的优化研究可归为以下三类：执行计划的优化、连接算法的优化和执行框架的优化。

对于第一类研究，文献[4]将 n 元连接拆分为 n-1 个二元连接，每个二元连接对应一个 MapReduce 作业(后文如不特殊指明，作业均为 MapReduce 作业)，然而该方法针对的是传统的多处理器计算环境，不适用于 MapReduce，且该方法在 n 较大时会导致较高的作业初始化代价以及中间结果的存储代价。文献[10]针对链式连接提出使用平衡二叉树(AVL)的方式来执行多元连接，但其并未给出平衡二叉树的构建规则。文献[11]在一个作业中完成所有的连接运算，然而该方法在 n 较大时会因为数据需要传输到多个 Reducer 而导致较高的网络 I/O 代价。文献[5, 6]将 n 元连接划分为若干个组，每组涉及若干个关系并由一个作业完成，而后采用 Replicated Join 连接所有组生成的中间结果，然而该方法因为要穷举所有可能的 $m(m<n)$元连接作为候选集而导致算法复杂度较高，从而使其应用范围较窄。此外，上述所有研究确定的执行计划都只确定了连接的执行顺序，并未考虑无依赖关系的连接操作间的并行执行策略。

本文首先基于 MapReduce 特性提出多元连接顺序的确定方法，该方法复杂度较低且能够很好地均衡中间结果的存储代价与网络传输代价。确定连接顺序后，本文还给出一种算法来确定无依赖关系的连接操作间的并行执行顺序，该算法通过对节点资源的充分利用来提高多元连接的执行效率。

对于第二类研究，文献[12]针对 theta-join 提出一种随机算法 1-Bucket-Theta 以及它的一个扩展算法 M-Bucket；文献[13]对文献[12]中提出的算法进行下界分析，并通过聚类方法提高了 M-Bucket 算法的效率。文献[14, 15]总结现有实现算法为 Map Join、Reduce Join、Semi Join 等，这些算法分别适用于不同的查询场景，如 Map Join 仅适用于数据量较小的关系能够装入内存的查询。MapReduce 连接算法的优化研究相对比较成熟，不在本文的研究范围之内。因此，不失一般性，本文采用没有任何约束条件的通用的 Reduce Join 作为研究对象。

[1] Replicated Join：在一个 MRJ 中执行多个连接操作。此时，同一个 Key-Value 对需要被复制到多个 Reducer 上，因此称为 Replicated Join。该算法牺牲部分网络 I/O 代价来换取 MRJ 的初始化以及 HDFS 读写代价。

连接运算的执行效率依赖于实现算法和运行环境，由此衍生出第三类研究。文献[16]基于 MapReduce 提出一个改进的执行框架 Map-Reduce-Merge，新添加的 Merge 阶段为 Reduce Join 的执行节省了一次作业。文献[17]对 MapReduce 框架进行修改，允许不同操作间的数据管道式传输，支持在线聚集以及持续查询，然而改进后的框架使得失效恢复(Fail Recovery)机制变得非常复杂，且对于批处理性能的提高也很不明显。文献[7-9]给出了通用的 MapReduce 负载均衡方法，但他们都只考虑了任务的输入代价，不适用于连接运算，因为它的输出代价也不可忽略。本文提出的 MapReduce 负载均衡方法着重考虑连接任务的负载特性，与传统的负载均衡方法有所不同。具体来讲，本文提出的负载均衡方法基于的 Reduce Join 的 Map 阶段仅负责将参与连接的数据表中的记录解析成 Key-Value 对（此时 I/O 操作很少），并通过 Shuffle 阶段传输到对应的 Reduce 任务中，而真正的连接操作是在 Reduce 阶段中完成的（此时会产生大量 I/O 操作）。考虑到 I/O 代价是影响连接查询的主要因素，我们对产生大量 I/O 操作的 Reduce 阶段进行读写分析，综合考虑 Reduce 任务的输入和输出代价及其对应的读写权重，最终基于这一综合代价给出了 Reduce 任务的负载均衡方法。

2. 连接执行计划

多元连接执行计划的解空间很大，短时间内很难找到最优解。本节首先通过查询树模型确定解的一个子空间，而后通过复杂度较小的启发式算法从中找到一个近似最优解，并在此基础上做进一步的优化以均衡中间结果的存储代价与网络传输代价。最后，根据 MapReduce 框架特性给出一种算法来确定无依赖关系的连接操作间的并行执行顺序，该算法通过对节点资源的充分利用来提高多元连接的效率。

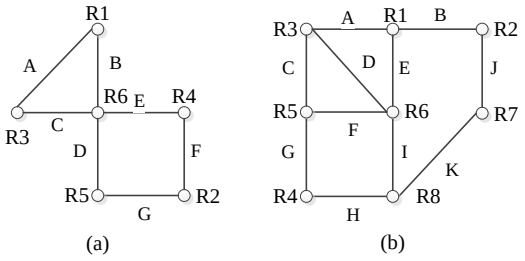


图 1. 多元连接查询示例

Fig.1 Example Queries of Multi-way Join

2.1 查询树模型

多元连接可以用一个查询图 $G=<V, E>$来表示[4-6]，其中 V 是节点的集合，每个节点代表一个关系(记为 R_i)，E 是边的集合，每条边$<R_i, R_j>$连接两个之间存在连接属性(A、B、C 等)的关系，如图 1 所示。图 1-(a)所示的查询图包含 6 个关系，为 6 元连接；同理，图 1-(b)所示的查询图为 8 元连接。

多元连接执行计划的最优解确定是个 P 完全问题[6]，传统优化方法通常采用查询树模型限定解的一个子空间，并设计算法从中获取一个最优解。如图 2 所示，主流的查询树模型有 Left-deep Tree、Right-deep Tree、Zigzag Tree 以及 Bushy Tree[18]四种，其中前三种为顺序执行，最后一种为并行执行。很多研究工作[4, 5]显示，并行执行的 Bushy Tree 更适用于分布式环境。从图 2 中也可以看出，只有基于 Bushy Tree 确定的连接顺序中不同连接操作间不是完全的依赖关系，是可以并行的。因此，本文选取 Bushy Tree 作为 MapReduce 多元连接的查询树模型。

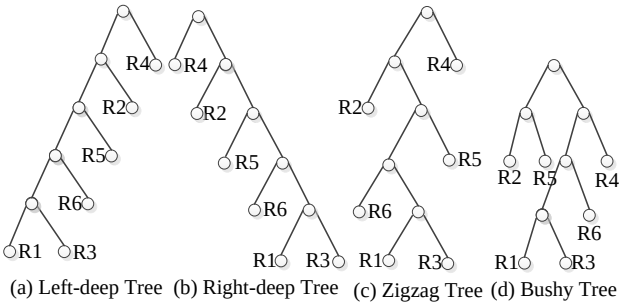


图 2. 查询树模型

Fig.2 Query Trees

2.2 查询树模型

一个 n 元连接的执行方式有两种：①将其拆分为 n-1 个二元连接分别执行，每个二元连接对应一个作业；②在一个作业中执行所有连接操作。然而，当 n 较大时，第一种方式的作业初始化代价以及中间结果的存储代价也随之增大，第二种方式也因为数据的多次传输而产生较大的网络代价。为解决该问题，本文首先基于 Bushy Tree 初步确定多元连接的执行顺序，而后根据是否受益将部分二元连接合并成多元连接。

（1）基于 I/O 代价的连接顺序确定方法

通过 Bushy Tree 确定多元连接的执行顺序首先需要给出树的构建规则。考虑到连接运算属于 I/O 密集型计算，连接代价以 I/O 代价为主，本文针对 MapReduce 特性给出连接运算的 I/O 代价模型，并选择 I/O 代价最小的连接顺序。

正如相关工作中的描述，本文选择 Reduce Join 作为连接算法，下面以二元连接为例对其进行简单介绍。Reduce Join 由 Map 阶段和 Reduce 阶段组成。Map 阶段主要完成如下操作：①Map 任务读取(通常为本地读)参与连接的两个关系；②以连接属性为键、

记录为值，按键排序后输出键值对到本地磁盘；③将中间结果通过网络传输给 Reduce 任务。Reduce 阶段主要完成如下操作：①接收来自 Map 任务的键值对并按键排序；②执行连接操作，并将结果写入分布式文件系统(HDFS)。基于该分析，我们给出关系 R_i 和 R_j 进行 Reduce Join 的 I/O 代价计算方法(见公式(1))。

$$\begin{aligned} Cost^{Map} &= C_1\times 4(|R_i|+|R_j|)+C_2\times(|R_i|+|R_j|)\\ Cost^{Reduce} &= C_1\times(1+\lambda)(|R_i|+|R_j|)+C_3\times|R_i\bowtie R_j|\\ Cost &= Cost^{Map}+Cost^{Reduce} \end{aligned} \tag{1}$$

其中，C_1、C_2 和 C_3 分别为本地、网络和 HDFS 的 I/O 代价权重，三者均与系统硬件相关(其中，C_3 还与 HDFS 的副本个数设置有关)，其值均可事先通过文件读写实验测出(在本文的实验环境中，副本个数为 3，通过实验测得三个参数值分别为 C_1=3.67(s/GB)、C_2=8.93(s/GB)和 C_3=13.37(s/GB)，三者之间的比值为 1:2.4:3.6)；$|R_i|$代表关系的基数。另外，Map 阶段的第②个操作首先需要溢出写文件，而后读取并排序输出，因此共需 3 次读写操作；Reduce 阶段的第①个操作使用内存和磁盘进行混合式排序，因此我们用参数 λ 表示该混洗比例，其值可通过经验设定。

通过 Bushy Tree 确定连接顺序时，我们每次从查询图 G=<V, E>中选择 I/O 代价最小的连接操作执行，而后更新图 G 及其对应的关系的特征参数，直到执行完所有连接运算(见算法 I)。算法 I 的复杂度为 log(|V||E|)，小于文献[5, 6]的复杂度 log(|V|^2|E|)。

算法 I. P_{MC}./*基于最小代价的连接顺序确定算法*/

输入：G = (V, E)，query profile/*包括关系的基数、连接属性的基数等相关参数*/

输出：Bushy Tree

1. Repeat until |V|=1
2. Choose $R_p\bowtie R_q$ from G = (V, E) satisfying that:
   $Cost(R_p, R_q)=MIN_{RiRj\in E}\ Cost(R_i, R_j)$
3. Merge R_p and R_q to $R_{min(p,\ q)}$ and update the profile;

算法 P_{MC} 中计算最小代价时，$|R_i|$和$|R_j|$均已知，$|R_i\bowtie R_j|$未知，需要我们计算。目前关于$|R_i\bowtie R_j|$的计算方法通常假设 R_i 和 R_j 在连接属性 A 上均匀分布[4, 9]，具体计算方法见公式(2)。

$$|R_i\ \triangleright\!\triangleleft\ R_j|=|R_i|\times|R_j|/|A| \tag{2}$$

其中，$|A|$为连接属性的基数。然而，事实上 R_i 和 R_j 通常不满足均匀分布这一假设，因此在实际应用中，我们应该考虑数据倾斜因素。设 F_i 和 F_j 为 A 在 R_i 和 R_j 中出现的频数分布，则定义倾斜度如下：

定义 1. 倾斜度. 定义倾斜度δ为频数分布 F_i 和 F_j 偏离均匀分布的程度，表达式见公式(3)。

$$\delta=\sum_1^{|A|}(f_{i1}-\overline{F_1})\times(f_{i2}-\overline{F_2})/|A| \tag{3}$$

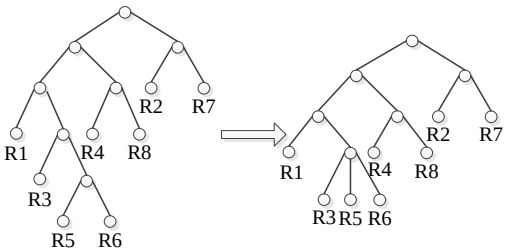
在实际计算中，倾斜度可以通过采样获取。有了倾斜度，$|R_i\bowtie R_j|$的计算方法可以表示为公式(4)。

$$\begin{aligned} &\because \sum_{i=1}^{|A|}(f_{i1}-\overline{F_1})\times(f_{i2}-\overline{F_2})=\sum_{i=1}^{|A|}f_{i1}\times f_{i2}-|A|\times\overline{F_1}\times\overline{F_2}\\ &\therefore |R_1\ \triangleright\!\triangleleft\ R_2|=\sum_{i=1}^{|A|}f_{i1}\times f_{i2}=|A|\cdot\delta+|R_i|\times|R_j|/|A| \end{aligned} \tag{4}$$

考虑倾斜度因素计算出的$|R_i\bowtie R_j|$更精确，同时基于最小代价的算法 P_{MC} 确定的连接顺序也更优。

（2）基于 Replicated Join 的优化

通过 Bushy Tree 确定的执行顺序仅包含二元连接，这样的执行计划会导致较高的作业初始化代价和中间结果的存储代价。考虑到算法 I 在复杂度上的优越性，我们保留由它确定的执行顺序，并在此基础上做进一步的优化。



(a) 算法P_{MC}确定的连接顺序　(b)基于Replicated Join的优化

图 3. 基于 Replicated Join 的优化

Fig.3 Optimization based on Replicated Join

解决上述问题的直观想法为减少作业个数，也即增加每个作业执行的连接操作个数。Replicated Join 满足该需求，但该算法中同一个键值对需要被复制到多个 Reduce 任务上，网络代价较高。为此，本文分别计算查询图采用 Replicated Join 以及采用多个二元连接这两种执行方法的 I/O 代价，定义“受益(*Benefit*)”为二者的代价差。当受益为正时，合并这些二元连接，如图 3 所示。二元连接的 I/O 代价见公式(1)，下面给出 Replicated Join 的 I/O 代价计算方法。

设查询图 G=<V, E>，关系集合 V={R_1, R_2, …, R_n}，E 关联的连接属性集合为 $\bar{E}$，R_i 关联的连接属性集合为 $\bar{E}_i$，Replicated Join 的 I/O 代价计算见公式(5)。

$$\begin{aligned} &Cost(G)\\ &=C_1\times(5+\lambda)\sum_{R_i\in V}|R_i|+C_3\times|R_1\bowtie R_2\bowtie\cdots\bowtie R_n|+\\ &C_2\times\sum_{R_i\in V}|R_i|\cdot\prod_{a_i\in\bar{E}}a_i\Big/\prod_{a_i\in\bar{E}_i}a_i \end{aligned} \tag{5}$$

基于 Replicated Join 的优化需要对算法 P_{MC} 确定的 Bushy Tree 进行遍历，以合并所有可能的二元连接。然而，这样做的代价很高，本文考虑到对无依赖关系的连接操作执行 Replicated Join 明显会导致较高的网络 I/O 代价，因此仅判定具有依赖关系的连接操作(也即图 3(a)中只能顺序执行的子树)。另外，如果

一个顺序执行的子树进行 Replicated Join 时受益为负，那么包含该子树的顺序执行子树的受益也为负。通过以上方法能够大大降低树的遍历代价。基于 Replicated Join 的优化算法见算法 II。

算法 II. OPT_B.//下标 B 为 Benefit 的缩写

输入：$G = (V, E)$，query profile/*G 为 P_{MC} 确定的 Bushy Tree*/

输出：Optimized Bushy Tree

1. Node R; Graph G_0, G_1;
2. Repeat until all leaf nodes are visited
3. Find the next leaf node R_i;//采用深度优先搜索方式
4. Set R_i to be visited; $G_0 = G_1$ = null; R=R_i;
5. IF (R.bro is a leaf node)
6. Set R.bro to be visited;
7. WHILE (R.parent.bro is a leaf node)
8. Set R.parent.bro to be visited;
9. G_0.root=R.parent;//G_0 为 G 的子图;
10. G_1.root=R.parent.paren;//G_1 为 G 的子图;
11. IF (computeBenefit(G_1)>0)
12. R=R.parent;
13. ELSE
14. G_1=G_0; Break;
15. IF (|G_1.leafNodes|>2)
16. Merge all the joins related with the leaf-nodes in G_1;
17. move pointer upward along the tree to the first node whose brother node is not a leaf node;

OPT_B 的算法复杂度小于算法 P_{MC} 确定的 Bushy Tree 中所有最大顺序执行子树的高度之和。又考虑到顺序执行子树的最大高度为 n-1，故 OPT_B 的算法复杂度为 O(n)。

2.3 并行执行顺序

很多关于多元连接执行计划的优化研究[4-6]都只确定了连接的执行顺序，并未考虑 MapReduce 环境下无依赖关系的连接操作间的并行执行策略。

若图 3(b)为 2.2 节中最终优化的多元连接顺序，那么连接操作 $R_2 \bowtie R_7$、$R_4 \bowtie R_8$ 和 $R_3 \bowtie R_5 \bowtie R_6$ 之间无依赖关系，可以并行执行。下面结合 MapReduce 特性对并行执行的优势进行分析。

MapReduce 集群中每个节点最多可执行的 Map 任务个数是预设的，因此最多可并行执行的 Map 任务数也是确定的。对于连接运算，Reduce Join 中的 Map 任务负责将关系中的元组按照连接属性值进行分区，其执行时间仅取决于处理数据量。又因为每个 Map 任务处理一个固定大小的分片，我们可以认为同时分配的 Map 任务同时结束。设 MapReduce 每次最多可并行的 Map 任务个数为 M，M 个任务的并行执行称为一轮[2, 19, 20]。若每轮 Map 任务的执行时间为 T，作业 J_i 需要的 Map 任务个数 $M_i = a_i*M + b_i$，其中 a_i、$b_i \in N$，$b_i \in (0, M)$，那么可以认为 J_i 的执行时间为 $(a_i+\lceil b_i/M \rceil)T$。若作业 J_j 需要的 Map 任务个数 $M_j = a_j*M + b_j$，那么当 $b_i + b_j \le M$ 时，两个作业的并行执行时间为 $(a_i+a_j+1)T$，而串行执行时间为 $(a_i+a_j+2)T$，此时并行执行可节省 T 时间；当 $b_i + b_j > M$ 时，并行执行时间和串行执行时间同为 $(a_i+a_j+2)T$。综上，当 $b_i + b_j \le M$ 时，作业 J_i 和 J_j 并行执行的效率高于串行。

当有多个作业满足这一条件时，我们选取 b_i+b_j 最大的两个作业并行执行，从而充分利用计算资源。下面给出具体的算法实现：

算法 III. P_{EE}.//高效的并行执行算法

输入：$G = (V, E)$，query profile/*G 为连接顺序优化图*/

输出：Parallelized Execution Sequence

1. Repeat until there are no join to be executed;
2. Find all of the independent joins in G and denote the set as J;
3. IF (|J|>2)
4. FOR EACH J_i in J;
5. computeMapTasks(J_i);
6. Select J_p and J_q to execute in parallel satisfying that:

$$b_p + b_q = MAX_{bi + bj <M}(b_i + b_j);$$

7. ELSE
8. Execute joins in J in parallel;
9. Delete all the leaf nodes involved and update the profile;

显然，算法 P_{EE} 仅遍历无依赖关系的连接操作，算法复杂度为 O(n)。

2.4 小结

通过 2.2 节和 2.3 节给出的优化算法，我们最终动态确定多元连接的执行计划。图 4 以流程图的方式描述了执行计划的优化步骤。

给定查询图 G，首先通过 P_{MC} 算法初步确定查询树(Bushy Tree)，而后分别通过算法 OPT_B 和 P_{EE} 进行优化，直到更新后的树中叶子节点的个数小于 3。

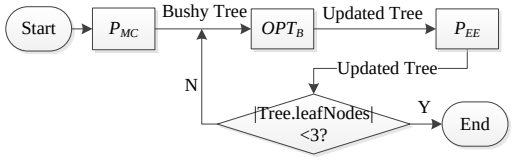


图 4. 执行计划优化流图

Fig.4 Optimization Flow of the Execution Plan

3 负载均衡

良好的执行计划固然重要，但对执行框架的优化同样可以有效地提高多元连接的执行效率，这一点在分布式环境中尤为重要。一种公平的并行任务负载分

配方法可以有效地减少 MapReduce 中的“短板效应”，从而提高连接操作内部的并行度。

连接运算的 MapReduce 实现算法有很多，分别适用于不同的查询场景。不失一般性，本文选择没有任何约束条件的 Reduce Join 作为连接执行算法。该算法包括 Map 和 Reduce 两个阶段，Map 阶段只负责将关系中的元组按照连接属性值进行分区以输出到不同的 Reduce 任务，运算完全相同，因此 Map 任务的负载完全取决于处理数据量。又因为 MapReduce 中每个 Map 任务只负责处理一个数据分片(Split，默认 64MB)，所以 Map 阶段各个 Map 任务是负载均衡的。很多研究工作也都做出 Map 任务均衡的假设，如文献[6]和[21]。因此，本文仅研究连接运算的 Reduce 任务负载均衡。

Reduce Join 算法在 Reduce 阶段执行连接运算，连接属性值的不均匀分布将会导致由默认 Hash 分区函数确定的 Reduce 任务负载不均衡。为提高 Reduce 任务间的并行度，本小节给出一种针对连接运算的负载均衡优化方法。

3.1 负载均衡模型

设 R_1 和 R_2 为参与连接的两个关系，连接属性值的集合记为 A，A 在 R_1 和 R_2 中的频数分布分别为 F_1 和 F_2。在计算 Reduce 任务的负载之前，我们先给出连接属性值 $a \in A$ 的负载贡献定义如下。

定义 3. 负载贡献. 连接属性值 $a \in A$ 的负载贡献(LC_a)是指执行该连接操作的代价，计算表达式见公式(6)。

$$LC_a = \omega_1 \cdot (f_{1a} + f_{2a}) + \omega_2 \cdot (f_{1a} \times f_{2a}) \quad (6)$$

上式中，f_{1a} 和 f_{2a} 分别为 R_1 和 R_2 中连接属性值为 a 的元组个数；ω_1 和 ω_2 为 Reduce 任务输入数据和输出数据的处理代价权重，输入数据为网络 I/O，输出数据为写到 HDFS 上，二者的比值是由运行多元连接的分布式集群系统决定的。此处，我们认为连接运算代价中 I/O 占主导地位，CPU 处理代价可以忽略，文献[5, 6]中也有同样结论。文献[8]给出的当前最好的负载均衡方法中采用的代价模型仅考虑输入数据对 Reduce 任务负载的影响，而事实上对于连接运算输出数据的代价不容忽略。

通过对 MapReduce 运行机制的分析可知，Reduce 任务的负载取决于分区函数。分区函数将连接属性值划分成若干个组，每个组对应一个 Reduce 任务。设分区函数将连接属性值的集合 A 划分为 R 个组，分别记为 A_1，A_2，…，A_R，那么组 A_i 的处理代价 $Load(A_i) = \sum_{a \in A_i} LC_a$

第 i 个 Reduce 任务的负载 $Load(R_i) = Load(A_i)$，Reduce 任务负载均衡这一目标可以等价表示如下：

$$\text{For } \forall 1 \le i, j \le R \text{且} i, j \in N,\ Load(R_i) = Load(R_j)$$

负载均衡模型中最关键的是获取连接属性 A 在 R_1 和 R_2 中的频数分布 F_1 和 F_2。获取这两个分布，最精确的方法是对不同键值进行频数统计[22]，但当键值个数很多时，会耗费大量存储，且在汇总各个 Mapper 的统计信息时还会带来很高的网络传输代价。针对该问题，文献[5, 7-9]对键值进行哈希从而降低统计信息的规模。然而，文献[9]在 Map 任务执行的同时对频数信息进行统计，这样会导致第二轮 Map 任务无法执行，还会造成数据到 Reduce 的传输延迟，因为必须等到根据频数信息确定 Partition 函数后才能进行传输。文献[5, 7, 8]则单独开启一个作业进行频数统计，避免了上述问题。基于以上分析，本文可以采用类似的方法获取连接属性值的频数信息，并据此确定 Reduce 任务的个数以及 Partition 函数。

3.2 负载均衡算法

文献[6]指出 Reduce 任务的负载均衡是一个 NP 难问题，不能够在多项式时间内获取最优解，因此我们仅专注于寻找尽可能接近最优解的近似解。由于连接属性值的不可分割性，拥有相同连接属性值的键值对必须发送到同一 Reduce 任务节点进行连接运算，Reduce 任务的负载均衡问题可以转换成尽可能降低 Reduce 任务的最大负载 $Max\{Load(R_i)\}$。

理想情况下，所有 Reduce 任务的负载完全相同，此时 $Max\{Load(R_i)\}=Avg\{Load(R_i)\}$。然而，这种情况不总发生，本文给出一种朴素的均衡算法来获取近似解。该算法首先对 A 中所有连接属性值的负载贡献值按降序排序，而后每次将连接属性值为 a 的键值对分配给当前负载最小的 Reduce 任务(详见算法 IV)。

算法 IV. *LBA*//Load Balancing Algorithm

输入：F_1 和 F_2 /*A 在 R_1 和 R_2 中的频数分布*/

输出：$A_1, A_2, \ldots, A_R$ /*A 的一个划分，A_i 对应于 R_i*/

1. Compute each LC_a in A;
2. Sort LC_a in decreasing order;
3. FOR EACH LC_a
4. Add a to A_i whose load equals Min$\{Load(A_i)\}$;
5. Update $Load(A_i)$;

为评估该算法，我们将由该算法获取的 Reduce 任务负载最大值 Max 与最优算法获取的最大值 Max^* 进行对比，并得出 Max 的上界如下：$Max \le 1.5Max^*$。

证明 1. $Max \le 1.5Max^*$.

证明：记 A 中具有最大负载值的划分组为 A_i，那么 $Max = Load(A_i)$。

①当 A_i 只包含一个连接属性值时，很容易证明 $Max = Max^* \le 1.5Max^*$；

②当 A_i 包含至少两个连接属性值时，设 a 为分配

给 A_i 的最后一个连接属性值，那么：

首先，易知 $Max^* \geq Avg\{Load(R_i)\}$；

其次，将 a 分配给 A_i 时，A_i 的负载更新前的值 $(Load(A_i) - LC_a)$应该是最小的，所以：

$$R \times (Load(A_i) - LC_a) \leq \sum_1^R Load(A_i)$$
$$(Load(A_i) - LC_a) \leq Avg\{Load(A_i)\} \leq Max^*$$

最后，因为连接属性值是按照递减的顺序分配的，所以 Max^*应该比第$(R+1)$个连接属性值的 2 倍大，而第$(R+1)$个连接属性值又比 LC_a 大，所以：

$$Max^* \geq 2LC_a,\ LC_a \leq Max^* / 2$$

综上，$Max = Load(A_i) - LC_a + LC_a \leq 1.5Max^*$。

本文给出的负载均衡方法是以等值连接为例进行描述的，它还适用于其他连接，也可以扩展到连接以外的其它类型作业。例如，进行近似连接时，只需将连接属性值 a(键值对中的键)替换为一个满足近似条件(如$|a_1-a_2|\leq\delta$)的二元组$<a_1, a_2>$，并将负载贡献值的计算公式中 f_{1a} 和 f_{2a} 分别替换为 R_1 中连接属性值为 a_1 的元组个数以及 R_2 中连接属性值为 a_2 的元组个数。执行 Replicated Join 时，键值对中的 *key* 将会变成多个连接属性构成的多元组。对于连接以外的其他作业，我们可以将 Reduce 任务输入数据的处理代价函数(频数的加和)以及输出数据的处理代价函数(频数的乘积)进行适应性的更改。

3.3 Reduce 任务个数的确定

现有通用的 Reduce 端负载均衡的方法[7-9]均未考虑 Reduce 任务个数的确定方法，本文根据获取的键值频数统计信息给出一种简单的确定规则。

设 Map 任务的输出中不同键值的个数为 k，所有键值的负载贡献加和为 Sum，其中键值的最大负载贡献为 LC_{max}，Reduce 任务个数为 R，通过优化算法获取的 Reduce 任务最大负载为 Max。当 $LC_{max} \geq Sum/R$ 时，也即 $R \geq Sum/LC_{max}$ 时，Max 的取值不再下降，始终为 LC_{max}，这意味着作业的性能不再提高，而它的资源消耗却随着 R 的增加而增大。因此，有必要找到 R 的一个临界值使得连接的执行效率最高。另外，考虑到 Max 的取值还与 Sum 有关，本文给出均衡算法的度量函数 $g(Max, R)$的表达式如下：

$$g(Max, R) = \frac{Max}{Sum} \times R^{\alpha}$$

其中，α 是性能与能耗之间的权重比，度量值越小，均衡效果越好。函数 $g(Max, R)$应该存在一个极小值点 R_0，使得在该点处性能与资源消耗达到一个很好的折中，且该值有可能比 Sum/LC_{max} 小。4.2 节中通过大量实验得出，当 α=0.05 时，函数 $g(Max, R)$的极小值点正好就是 Max 不再下降的临界值。

另外，考虑到不同键值的个数可能会很大，这将导致频数统计信息不能存入内存，针对该问题，我们可以采用文献[8]中提出的 optimal sketch packing 算法，该方法通过 Hash 函数将键值(这里是负载贡献值)进行哈希后再进行均衡分配，从而降低键的规模，节省统计信息占用的内存。本质上，该方法是牺牲精确度来降低统计信息的存储空间。

4 实验与范例分析

本节设计实验对提出的连接执行计划优化方法以及连接负载均衡方法进行验证和分析。

4.1 执行计划的优化效果分析

以图 1 中的查询图为例对本文提出的执行计划优化方法进行效果分析，相应的特征参数见表 1 和 2。

表 1 图 1(a)对应的关系特征参数

Table 1. Cardinalities of the Relations in Fig.1(a)

关系	R_1	R_2	R_3	R_4	R_5	R_6
基数	235	204	212	200	262	240
属性	*A*	*B*	*C*	*D*	*E*	*F*	*G*
基数	19	15	17	19	16	15	18

表 2 图 1(b)对应的关系特征参数

Table 2. Cardinalities of the Relations in Fig.1(b)

关系	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
基数	100	85	93	106	102	90	101	94
属性	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*
基数	9	8	7	9	9	10	9	7	7	10	8

首先，为了验证本文提出的连接顺序确定算法 P_{MC} 的优化效果，本文将其与最优解(可用分支限定法获取)进行对比。图 5 中，P_{OPT} 代表最优解，从中可以看出 P_{MC} 的代价比最优解稍高，二者的比值分别为 1.003 和 1.034。由此可见，P_{MC} 算法能够确定一个很好的连接顺序，从而降低连接的 I/O 代价。

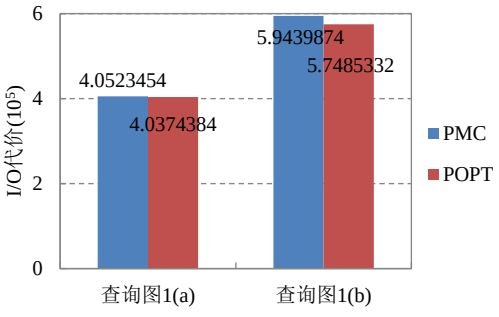


图 5. 算法 P_{MC} 与最优解的 I/O 代价对比

Fig.5 I/O Cost Comparison between P_{MC} and the Optimal Result

接着，我们分析了两种查询图下算法 OPT_B 和 P_{EE} 的优化效果。从表 3 中可以看出，算法 OPT_B 能够找出可以合并为多元连接的二元连接，一定程度上降低了 I/O 代价；算法 P_{EE} 能够找出最大限度使用集群计算资源的可并行连接操作，从而节省多元连接的整体运行时间。

表 3 算法 OPT_B 和 P_{EE} 的优化效果（表中提到的循环为图 4 中的循环优化）

Table 3. Optimization Results of Algorithms OPT_B and P_{EE} (The loops here refer to the optimization loops in Fig.4)

	OPT_B	P_{EE}
图 1(a)	无优化	第一轮循环确定 $R_1 \bowtie R_3$ 和 $R_5 \bowtie R_6$ 并行，共节省 1T 时间。
图 1(b)	第四轮循环合并 $R_1 \bowtie R_4$ 和 $R_1 \bowtie R_2$ 为 $R_1 \bowtie R_2 \bowtie R_4$，I/O 代价降低了 30226.18。	第一轮循环确定 $R_1 \bowtie R_3$ 和 $R_2 \bowtie R_7$ 并行，第三轮循环确定 $R_1 \bowtie R_5$ 和 $R_4 \bowtie R_8$ 并行，共节省 2T 时间。

4.2 负载均衡方法验证

文献[5, 7-9]中均提到采用模拟实验验证其提出的负载均衡方法，假设键值服从 *Zipf* 分布。不失一般性，本文也采用该方法来验证第 3 节中针对连接运算设计的负载均衡方法。以等值二元连接为例，假设参与连接的两个关系表 R_1 和 R_2 服从相同的 *Zipf* 分布，数据条数分别为$|R_1|$和$|R_2|$，不同连接属性值的个数 k，则 R_1 和 R_2 中第 i 个最频繁出现的连接属性值的出现概率 $p_i = 1/(i^z \times H_k)$，其中 z 代表数据的倾斜度，H_k 是调和系数，那么第 i 个连接属性值的负载贡献值计算如下：

$$LC_i = \omega_1 \cdot (|R_1| \times p_i + |R_2| \times p_i) + \omega_2 \cdot (|R_1| \times |R_2| \times p_i^2)$$

具体测试用例设计为：$|R_1|$=10 亿，$|R_2|$=20 亿，k=3 万、30 万和 300 万，z 的取值范围为[0, 1]。这里，需要指出的是，z 仅代表关系 R_1 和 R_2 中连接属性值的倾斜程度，并不代表负载贡献值集合的倾斜度(记为δ)，但δ与 z 值是成正相关的，且比 z 大。

首先，我们对比不同因素下负载均衡算法的效果，采用 Imbalance Ratio(简记为 *IR*)来度量，它是所有 Reduce 任务中的最大负载(*Max*)与平均负载(*Avg*=*Sum*/*R*)之间的比值。从图 6 中可以看出，*IR* 的影响因素有δ(受 z 影响)、k 和 R，与δ和 R 成正相关，与 k 成负相关。从图 6 中我们还可以看出，*IR* 的值早在 z=0.5、R=64 时已经超过 1.5，这是因为 *IR* 是 *Max* 与 *Avg* 之间的比值，而通过最优负载均衡算法获取的 *Max**通常会因为δ较大而远大于 *Avg*。例如，当一个键的频数占总频数的 80%时，因为键的不可分割性，*Max* 和 *Max**会远大于 *Avg*。

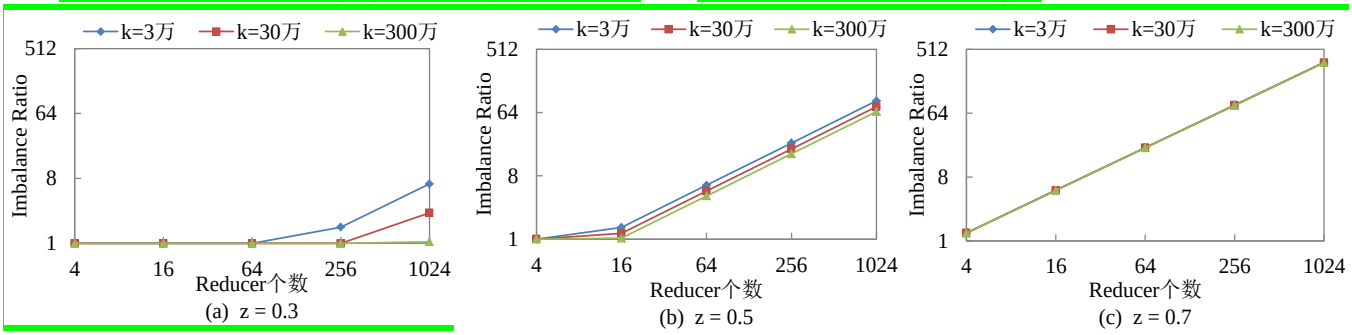


图 6. 不同倾斜度下负载均衡算法的 Imbalance Ratio

Fig.6 Imbalance Ratios of the load balancing algorithm under different skew degrees

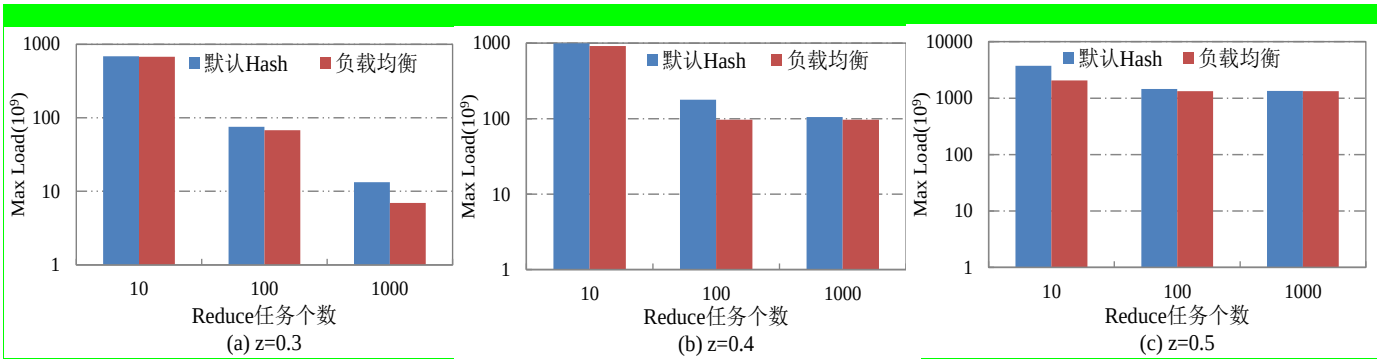
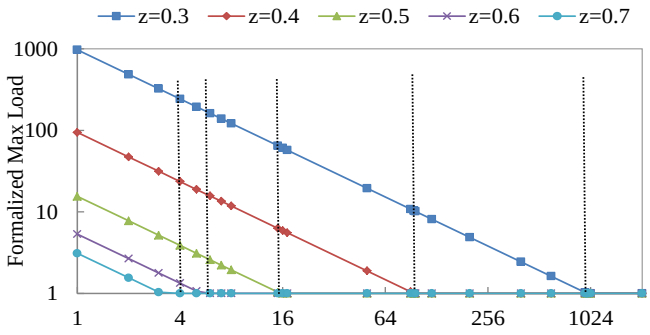
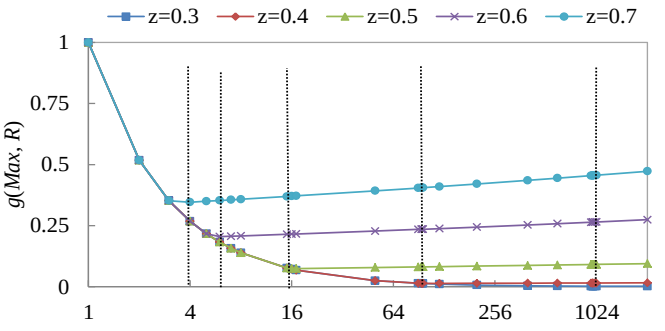


图 7. k=300 万时不同倾斜度下负载均衡算法与默认 Hash 的最大负载对比

Fig.7 Max load comparisons between the load balancing algorithm and the default hash under 3 million keys



(a) *Max*变化曲线



(b) *g*(*Max*, *R*)变化曲线

图 8. *k*=300 万时正规化的最大负载 *Max* 以及函数 *g*(*Max*, *R*)随 *R* 的变化曲线

Fig.8 Relationships between the formalized *Max*, *g*(*Max*, *R*) and *R* under 3 million keys

其次，为了验证本文设计的负载均衡算法的有效性，我们将其得到的最大负载值与默认 Hash 函数得到的进行对比。从图 7 中可以看出，在三种倾斜度、三种 Reduce 个数下，我们的算法均比默认 Hash 的好。从图 7（a）可以看出，随着 Reduce 个数的增加，负载均衡算法的优势越来越明显；而在图 7（b）和图 7（c）中，Reduce 个数为 100 和 1000 时，均衡算法得到的最大负载值均未发生变化，这是因为此时数据太过倾斜而产生了“二八现象”，最大负载值分别在 Reduce 个数大于 95 以及 16 后不再发生变化（从图 8 中可以看出），这也验证了我们在 3.3 节中的理论分析。另外，在图 7（b）和图 7（c）中，虽然默认 Hash 得到的最大负载值依然随着 Reduce 个数的增加而降低，但该值由于数据倾斜以及键值不可分割等的原因而不会低于均衡算法得到的值。

最后，为了验证 3.3 节中提出的 Reduce 任务个数的确定方法，我们分析了正规化后的最大负载 *Max* 与负载均衡算法的评估函数 *g*(*Max*, *R*)随 Reduce 任务个数 *R* 的变化情况。通过大量实验，我们发现当函数 *g*(*Max*, *R*)中的 α=0.05 时，它的极小值点正好就是 *Max* 不再下降的临界值 R_0。另外，在实验过程中，我们发现 *z* 值越大，临界值 R_0 的下降趋势越不明显，因此，为直观起见，本文选取了其中 5 个具有代表性的 *z* 值进行展示。从图 8(a)中可以看出，随着 *R* 的增长，*Max* 不断减小，最终趋向平稳值。图中 5 个 *z* 值对应的临界 *R* 值分别为 974、95、16、6 和 4，这与我们通过均衡算法度量函数 *g*(*Max*, *R*)得到的极小值点是完全吻合的(见图 8(b))。

5 总结

本文基于 MapReduce 研究多元连接的优化方法，主要从以下两部分展开研究：连接的执行计划和连接的负载均衡。

针对前者，本文首先分析现有主流的查询树模型，确定适合本文研究环境的查询树(Bushy Tree)；随后通过白盒分析给出 MapReduce 连接算法的 I/O 代价模型，并选择 I/O 代价最小的连接顺序作为初步的执行计划；接着对执行计划做进一步的优化，根据是否受益将查询树中的二元连接合并成 Replicated Join，以降低多个作业引起的中间结果代价；最后结合 MapReduce 特性提出一种作业并行执行算法，以提高集群资源的使用率。

针对后者，本文首先分析连接运算的特性，给出连接负载的定义以及负载均衡目标；接着给出具体的均衡算法，并证明该算法的上界；最后在实验中分析 Reduce 任务个数的确定与性能之间的关系。

实验证明，本文提出的连接执行计划以及负载均衡的优化算法是有效的。本研究对大数据环境下 MapReduce 多元连接的应用具有指导意义，可以优化如 OLAP 分析中的星型连接，社交网络中社团发现的链式连接等应用的性能。

参 考 文 献

[1] Han Xixian, Yang Donghua, Li Jianzhong. Approximate join aggregate on massive data [J]. Chinese Journal of Computers, 2010, (10): 1919-1933(In Chinese).
(韩希先, 杨东华, 李建中. 海量数据上的近似连接聚集操作[J]. 计算机学报, 2010, (10): 1919-1933)

[2] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters [J]. Communications of the ACM, 2008, 51(1): 107-113.

[3] Doulkeridis C, Norvag K. A survey of large-scale analytical query processing in MapReduce [J]. The VLDB Journal, 2013: 1-26.

[4] Chen M S, Yu P S, Wu K. Optimization of parallel execution for multi-join queries [J]. IEEE Transactions on Knowledge and Data Engineering, 1996, 8(3): 416-428.

[5] Wu S, Li F, Mehrotra S, et al. Query optimization for massively parallel data processing [C]// Proceedings of the 2nd ACM Symposium on Cloud

Computing. Cascais, Portugal, 2011.

[6] Zhang X F, Chen L, Wang M. Efficient multi-way theta-join processing using MapReduce [J]. Proc. VLDB Endow, 2012, 5(11): 1184-1195.

[7] Gufler B, Augsten N, Reiser A, et al. Load balancing in MapReduce based on scalable cardinality estimates [C]// Proceedings of the International Conference on Data Engineering. Arlington, VA, US, 2012: 522-533.

[8] Yan W, Xue Y, Malin B. Scalable and robust key group size estimation for reducer load balancing in MapReduce// Proceedings of the IEEE International Conference on Big Data. Santa Clara, CA, US, 2013: 156-162.

[9] Gufler B, Augsten N, Reiser A, et al. Handling data skew in MapReduce [C]// Proceedings of the 1st International Conference on Cloud Computing and Services Science. Noordwijkerhout, Netherlands, 2011: 574-583.

[10] Zhou M Q, Zhang R, Zeng D D, et al. Join Optimization in the MapReduce environment for column-wise data store [C]// Proceedings of the Sixth International Conference on Semantics Knowledge and Grid. Los Alamitos, CA, USA, 2010: 97-104.

[11] Afrati F N, Ullman J D. Optimizing multiway joins in a map-reduce environment [J]. IEEE Transactions on Knowledge and Data Engineering, 2011, 23(9): 1282-1298.

[12] Okcan A, Riedewald M. Processing theta-joins using MapReduce [C]// Proceedings of the ACM SIGMOD International Conference on Management of Data Athens. Athens, Greece, 2011: 949-960.

[13] Koumarelas I K, Naskos A, Gounaris A. Binary theta-joins using MapReduce: efficiency analysis and improvements [C]// Proceedings of the Workshops of the EDBT ICDT 2014 Joint Conference. 2014.

[14] Blanas S, Patel J M, Ercegovac V, et al. A comparison of join algorithms for log processing in MaPreduce [C]// Proceedings of the ACM SIGMOD International Conference on Management of Data. Indianapolis, IN, US, 2010: 975-986.

[15] Luo G, Dong L. Adaptive join plan generation in Hadoop [R]. Duke University, Durham NC, USA, 2010.

[16] Yang H, Dasdan A, Hsiao R, et al. Map-reduce-merge: simplified relational data processing on large clusters [C]// Proceedings of the ACM SIGMOD international conference on Management of data. Beijing, China, 2007: 1029-1040.

[17] Condie T, Conway N, Alvaro P, et al. Online aggregation and continuous query support in MapReduce [C]// Proceedings of the ACM SIGMOD International Conference on Management of Data. Indianapolis, IN, US, 2010: 1115-1118.

[18] Aljanaby A, Abuelrub E, Odeh M. A survey of distributed query optimization [J]. Int. Arab J. Inf. Technol, 2005, 2(1): 48-57.

[19] Agrawal P, Kifer D, Olston C. Scheduling shared scans of large data files [J]. Proceedings of the VLDB Endowment, 2008, 1(1): 958-969.

[20] Li F, Ooi B C, Ozsu M T, et al. Distributed data management using mapreduce [J]. ACM Computing Surveys, 2014, 46(3).

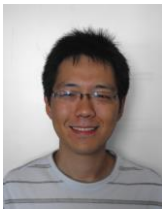
[21] Nykiel T, Potamias M, Mishra C, et al. MRShare: sharing across multiple queries in mapreduce [J]. Proceedings of the VLDB Endowment, 2010, 3(1): 494-505.

[22] Ibrahim S, Jin H, Lu L, et al. LEEN: Locality/fairness-aware key partitioning for MapReduce in the cloud [C]// Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science. Indianapolis, IN, US, 2010: 17-24.

Li Tiantian, born in 1989, PhD candidate. Student member of China Computer Federation (CCF). Her research interests include energy efficient computing, and data intensive computing.

Yu Ge, born in 1962. Professor and PhD supervisor in Northeastern University. His main research interests include database theory and data flow.

Guo Chaopeng, born in 1990, master candidate. His research interests include iterative computing, and data intensive computing.



Song Jie, born in 1980. PhD and associate professor in Northeastern University. His research interests include cloud computing, data intensive computing and big data.