软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(6):1438−1456 [doi: 10.13328/j.cnki.jos.004586]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

MapReduce 连接查询的 I/O 代价研究*

宋 杰[1], 李甜甜[2], 朱志良[1], 鲍玉斌[2], 于 戈[2]

[1](东北大学 软件学院,辽宁 沈阳 110819)

[2](东北大学 信息科学与工程学院,辽宁 沈阳 110819)

通讯作者: 宋杰, E-mail: songjie@mail.neu.edu.cn, http://www.neu.edu.cn

摘 要: 数据的指数级增长给数据管理和分析带来了严峻的挑战.连接查询是数据分析中一种常用运算,而MapReduce是一种用于大规模数据集并行处理的编程模型,研究基于MapReduce的连接查询代价评估和查询优化,有着学术意义和应用价值.MapReduce连接查询算法的性能主要取决于I/O代价(包括本地和网络I/O),而I/O代价与数据集以及连接运算的特征参数相关,通过对二元连接的I/O代价评估可以优化多元连接执行计划.基于此,首先提出了二元连接查询的I/O代价模型;随后,对现有二元连接算法进行形式化定义和简单扩展,归纳出6种基于MapReduce连接查询算法,并通过算法白盒分析定义它们的I/O代价函数;最后,提出一种多元连接最优执行计划的选择算法.通过实验表明I/O代价模型的正确性且能够准确地反映算法的性能优劣.

关键词: 连接查询;MapReduce;I/O代价模型;查询优化

中图法分类号: TP311

中文引用格式: 宋杰,李甜甜,朱志良,鲍玉斌,于戈.MapReduce 连接查询的 I/O 代价研究.软件学报,2015,26(6):1438−1456. http://www.jos.org.cn/1000-9825/4586.htm

英文引用格式: Song J, Li TT, Zhu ZL, Bao YB, Yu G. Research on I/O cost of MapReduce join. Ruan Jian Xue Bao/Journal of Software, 2015,26(6):1438−1456 (in Chinese). http://www.jos.org.cn/1000-9825/4586.htm

Research on I/O Cost of MapReduce Join

SONG Jie[1], LI Tian-Tian[2], ZHU Zhi-Liang[1], BAO Yu-Bin[2], YU Ge[2]

[1](Software College, Northeastern University, Shenyang 110819, China)

[2](School of Information and Engineering, Northeastern University, Shenyang 110819, China)

Abstract: The exponential growth of data has posed serious challenges to the data management and analysis. Join query is a common data analysis operation, and MapReduce is a programming model implemented for parallel processing on large-scale datasets. Therefore the research on MapReduce based join algorithms and its cost model has a certain academic significance and application value. This study believes that the I/O (including the network and the local I/O) cost is the main factor affecting the performance of MapReduce based join algorithm. Furthermore, as the I/O cost is determined by the feature of both datasets and join operation, the executed plan of multi-ways join could be optimized by evaluating the I/O cost of two-ways join. In the study, an I/O cost model of two-ways join is proposed and then formally defined as a simple extension to the existing MapReduce based join algorithms, resulting in six join algorithms and their I/O cost functions through write-box analysis. In addition, an selection algorithm to find the best executed plan of multi-ways join is presented. The correctness and accuracy of the I/O cost model are validated through a series of experiments. The experiment results suggest that the I/O cost can accurately reflect the algorithm performance.

Key words: join; MapReduce; I/O cost model; query optimization

* 基金项目: 国家自然科学基金(61433008, 61202088, 61402090); 教育部高等学校博士学科点专项科研基金(20130042120006); 中国博士后科学基金面上项目(2013M540232); 中央高校基本科研业务费重大科技创新项目(N120817001); 辽宁省博士启动基金(201403314)

收稿时间: 2013-06-04; 修改时间: 2013-12-03; 定稿时间: 2014-01-21

近年来,随着计算机技术在互联网、传感器以及科学数据分析等领域的应用,数据呈指数级地增长,海量数据给传统的数据管理和分析带来了新的挑战.海量数据具有数据量大(TB-PB 级别)、半结构化或非结构化、异构、实时等特点.海量数据的分析和处理包含了数据高可用性、复杂和时间苛刻的运算等诸多新内涵.《自然》杂志在 2008 年 9 月专门出版了一期 Big Data(大数据)的专刊[1],从互联网技术、超级计算、环境科学、生物医药等多个方面介绍了海量数据带来的技术挑战、现有解决方法以及可以预见的未来发展方向.美国自然科学基金委也是从 2008 年开始支持数据密集型计算的研究,旨在推动该技术的发展,实现那些依赖传统的单一数据源、准静态数据库无法实现的应用[2].在这种背景下,产生了多种海量数据的管理和分析技术,以 Google GFS[3] 和 Hadoop HDFS[4]为代表的分布式文件系统和 MapReduce[5]分布式并行编程模型在学界和业界最为流行.MapReduce 编程模型是本文研究的重点,它是分布式环境中并行处理海量数据算法所参照的编程模型.MapReduce 应用程序采用迁移计算而非迁移数据的理念,能够在大量普通配置计算机上并行地处理海量数据.

与传统数据分析类似,海量数据分析中最常见的操作为连接查询.现有海量数据基于 MapReduce 的连接研究均从算法角度考虑二元连接(two-ways join)运算的优化,因此提出了很多连接的实现算法,如 Map 端连接、Reduce 端连接、半连接等[6,7].本文从另外一个角度优化连接.我们注意到以下事实:① 尽管现有很多 MapReduce 连接算法,但每一种算法都有其适用范围,某算法在特定环境下性能会最优;② 实际应用环境很少出现简单的二元连接,而更多的是复杂的多元连接(multi-mays join),最常见的为分析型数据仓库中维数据集和事实数据集之间的星型连接,以及社会网络中多个数据集的链式连接,n 元连接可以以多种方式分解成 $n-1$ 个二元连接,每二元连接又可以采用多种算法,因此存在大量执行计划;③ MapReduce 连接算法包含大量 I/O 操作,是一种数据密集型计算,二元连接的运算并不复杂,算法代价以 I/O 为主,CPU 因等待 I/O 而空闲,因此,优化连接性能最有效的方法是减少 I/O 操作.

本文研究连接运算的 I/O 代价,我们假设多元连接均转化成二元连接执行,基于此,我们提出以下问题:

(1) 有多少种本质不同的 MapReduce 算法可以实现二元连接查询,一个多元连接有多少种可行的执行计划;

(2) 对于多元连接,是否可以根据上下文计算每一种执行计划的 I/O 代价,以此选择 I/O 代价最小的执行计划;

(3) 连接查询的 I/O 代价与哪些因素有关,如何定量分析这种关系,是否可以对现有算法的 I/O 代价进行优化.

MapReduce 作业的执行是一个复杂的过程,涉及到多节点之间的协作和复杂的数据交互,I/O 代价又取决于很多因素,如数据量、数据连接率、属性选择率、并行度、Map 任务个数、Reduce 任务个数、网络带宽等,因此,对连接查询的 I/O 代价进行定量分析存在一定挑战.就我们目前所知,尚未发现基于 MapReduce 的连接算法 I/O 代价评估的研究报道,上述问题亟待解决.

本文首先在 MapReduce 连接现有的两种主流算法 Map-Join 和 Reduce-Join 的基础上,定义 MapReduce 连接查询和其关键算法步骤;接着,定义基于 MapReduce 的连接查询的 I/O 代价模型;随后,对现有 Map-Join 和 Reduce-Join 加以整理和扩展,给出 6 种二元连接算法的形式化表达以及 I/O 代价与各影响因素的函数表达,称为 I/O 代价函数;随后,将 I/O 代价模型和函数扩展到星型连接、链式连接等多元连接上,提出一种最优执行计划的选择算法;最后,通过实验证明 I/O 代价函数对 I/O 代价估算的准确性、连接算法的 I/O 代价规律与理论模型的一致性,并证明了 I/O 代价是影响连接算法性能的主要因素.本文工作将对海量数据连接算法的评价和优化,以及基于 MapReduce 的数据管理系统和分析系统的研发起指导作用.

本文第 1 节介绍研究背景和相关工作.第 2 节描述连接查询的定义和已知 MapReduce 实现算法.第 3 节给出 MapReduce 连接查询算法主要步骤的 I/O 代价模型.第 4 节阐述 6 种二元连接算法和其 I/O 代价函数.第 5 节基于二元连接的 I/O 代价函数提出多元连接执行计划的优化选择算法.第 6 节验证 I/O 代价模型的正确性,并分析算法 I/O 代价与性能之间的关系.第 7 节总结全文并提出下一步工作.

1 相关工作

MapReduce[5]编程模型已被广泛运用于海量数据的分析和处理中,它有多种实现:Google MapReduce, Apache Hadoop[8],Map-Reduce-Merge[9],多核和多处理器系统的 MapReduce[10]等.连接查询可以分为查询和连接这两个个关键操作,通过 MapReduce 很容易实现数据查询,因此,连接操作成为研究关注的焦点.有很多学者注意到了 MapReduce 环境下连接算法的多种实现方式,研究了算法细节及其代价模型,这些研究极具参考价值,我们把这些研究工作归纳为 3 类.

第 1 类是对现有 MapReduce 连接算法的总结和评价.

文献[6]总结了等值连接在 Pig[11],Hive[12]以及 Jaql[13]中的执行算法,如 Repartition Join,Broadcast Join,Semi-Join,并对每种算法进行改进,增加了数据预处理阶段;还比较了一系列算法的性能,并以性能作为代价模型得出不同算法的选择决策树.文献[14]提出一种连接算法的代价评估模型,该模型根据数据集的大小、Map 和 Reduce 任务个数确定算法的执行代价,然而该模型并未得到实验验证.文献[7]总结 Hadoop 中连接的实现方式为 Default Join,Map Side Join,Semi Join,Advanced Default Join 和 Advanced Map-Side Join,并提出一种代价模型,该模型综合考虑本地 I/O、本地运算、网络 I/O 和结果集大小,但也仅是基于数据集的大小定性度量了算法的代价,没有定量的分析.文献[15]在研究 Hadoop 多种连接算法的同时也提出了代价模型,该模型考虑的因素有分布式文件个数、网络速度、Map 和 Reduce 任务个数、节点个数以及数据集大小;通过该模型估算了各种连接算法的执行代价,并给出了算法选择策略.但文献[15]的代价模型过于简单,没有考虑到执行连接操作的数据量并非总数据量以及数据分布不均匀对并行程度影响,也没有通过实验证明代价模型的准确性.文献[16]提出了多路连接(multi-way join)模型,并给出了连接代价的粗略计算方法,这是对连接算法的一次深入研究,指出网络数据传输代价是决定 MapReduce 性能的主要因素,认为 Map 和 Reduce 个数会影响网络通信.文中首先定义了网络代价模型,随后对模型进行求解,找到使网络通信最小的 Map 和 Reduce 个数.然而本文认为, Map 和 Reduce 个数不是影响通信代价的唯一因素.本文综合考虑了数据集大小、连接率、查询率、投影率、Map 和 Reduce 任务个数以及网络速度等多种影响因素,对连接查询算法的 I/O 代价(包括本地 I/O 和网络 I/O)进行了准确地定量分析,并通过实验证明了该代价模型的正确性.

第 2 类是对现有 MapReduce 连接算法的改进.

文献[17]对传统的 Repartition Join(reduce join)和 Replication Join(map join)算法做了一定程度上的改进,通过改变连接顺序以及把运算量大的 Reduce 任务分配到计算能力强的节点来提高连接效率.文献[18]提出一种列数据库的基于 MapReduce 的星型连接实现方式,采用分布式索引提高星型连接效率.文献[19]提出一种基于 Bipartite Join Graph 的链式连接(chain join)的 MapReduce 实现方式.文献[16]介绍了 Hadoop 中几种经典的连接实现方法,包括一种使用分布式缓存进行优化的连接实现算法.文献[20]提出一种并发连接算法,把多个数据集的连接查询(如星型模式下的查询)转换成能够并行执行的两个数据集间的连接.本文并非采用预处理、索引、缓存等技术来优化算法,而是把多元连接拆分成多个二元连接,通过连接执行顺序和方法的不同来优化整个过程,大部分针对连接的优化都可以运用到这个执行计划的某一特定步骤上.

第 3 类是对特定连接算法的研究及应用.

文献[21]研究基于 MapReduce 的非等值连接算法;文献[22]提出一种在科学数据领域常用的 Cross-Match 连接方法的 MapReduce 实现,属于 MapReduce 在特定领域的应用;文献[23]提出用于 RDF 数据的 MapReduce 优化连接算法,采用一种称为 All Possible Join 的索引结构来缩短 MapReduce 作业的搜索空间;文献[24]研究了海量数据环境下的近似连接聚集算法;文献[25]提出了一种在私有表上计算连接和聚集操作但不暴露细节数据的办法,采用一种 Sketching 的办法来隐藏细节数据,并未使用 MapReduce 技术.

与本文最为相似的是文献[26],它在多核环境下的并行数据库中对连接算法进行了优化,文中并未使用 MapReduce 模型,主要采用对数据进行分区的方法提高并行性,减少数据传输代价.文献[26]提出了一种基于执行时间的代价模型,并通过实验对模型进行了大量验证.本文借鉴了该文献中的很多方法,如参数的选择,但与其有本质的不同.本文面向的 MapReduce 与并行算法有显著区别,并未从数据分区的角度来优化性能,因为集群

环境下海量数据的分区存在一定困难:按文献[26]的做法,难以找到确定的分区键对数据进行预分区,且海量数据环境下维护这一分区的代价也很高.

综上,本文提出的连接查询算法和 I/O 代价模型借鉴了现有基于 MapReduce 的连接运算研究工作,但与之有明显区别.本从 MapReduce 算法角度讨论连接查询的多种实现;提出的代价模型基于 I/O 代价而非执行时间,综合考虑各种影响因素定量分析 I/O 代价,指导调度程序选择 I/O 代价小的执行算法.

2 连接查询

查询是在数据集中检索满足特定条件的数据并生成结果集的过程,连接运算则把两个或多个数据集中的记录按条件组合为一个结果数据集,我们将包含连接运算的查询统称为连接查询.连接查询在数据分析中非常常见,TPC-H 提供的 22 个查询用例中有 16 个涉及到此类查询.连接运算有多种方式,内连接的连接查询结果集中仅包含满足条件的行,是大部分数据库系统默认的连接方式.根据所使用的比较方式不同,内连接又分为等值连接、自然连接和不等连接等 3 种;交叉连接的连接查询结果集包含两个数据集所有行的组合,又称笛卡尔连接;外连接的连接查询结果集既包含那些满足条件的行,还包含其中某个数据集的全部行,有 3 种形式的外连接:左外连接、右外连接、全外连接.在上述连接种类中,最为常用的是等值连接和自然连接,也是本文研究的重点.

当一个连接操作涉及多个数据集时,我们称该连接为多元连接(multi-ways join).连接运算可采用如图 1(a)~图 1(c)所描述的 3 种连接方式:形如图 1(a)的连接运算是 R 和 S 最简单的二元连接,广泛用于简单的数据管理和数据查询;形如图 1(b)和图 1(c)的连接运算为数据集 S_0 和多个数据集 $S_1 \ldots S_n$ 进行星型连接,这种连接方式等同于事实数据集和多个维数据集的连接,多用于数据分析、OLAP 和数据集成应用;形如图 1(c)的连接运算为多数据集($S_1 \ldots S_n$)逐一链式连接,这种连接方式多用于 Web 数据和社会网络中的查询.

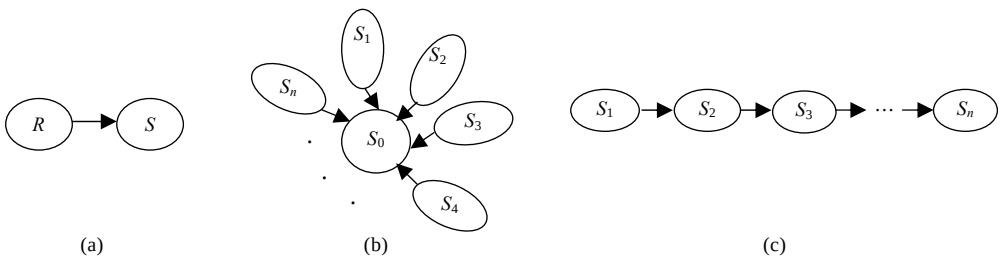


Fig.1 Examples of two-ways join, star join and chain join

图 1 二元连接、星型连接和链式连接示意图

一般的,n 元连接可以转换为 n–1 个 2 元连接并按照特定顺序执行;在特定用例下,还可以利用 MapReduce 让多个 2 元连接并行执行.不失一般性,我们定义二元连接查询中涉及运算的两个数据集为 x 和 y,其中,较大的数据集 x 为主表,较小的数据集 y 为从表.定义连接条件为 $x.a=y.a$,查询条件为 C_x 和 C_y,投影属性为 P.由此,连接定义为 $\sigma_{x.a=y.a}(x \times y)$,投影定义为 $\prod_P(x \times y)$,则二元连接查询可以定义为

$$\prod_P(\sigma_{x.a=y.a \wedge C_x \wedge C_y}(x \times y)) \tag{1}$$

基于此,我们讨论 MapReduce 连接查询的实现算法.Map-Join 和 Reduce-Join 是连接操作基于 MapReduce 的两种实现:前者首先需要将从表分发到每个 Map 节点或各个节点的分布式缓存中,然后在 Map 任务中完成连接操作;后者的 Map 任务负责同时读取主表和从表,并为每条数据打上标签以区别数据来源,Reduce 任务进行连接操作.Reduce-Join 是最常用的 Join 方式,对于给定的两数据集 x 和 y,Map 任务分别读取两数据集的各个部分,从每个记录中按查询条件(C_x 和 C_y)抽取连接属性值($x.a$ 和 $y.a$),作为 Map 任务的 key 值输出,这样,具有相同连接属性值的记录就会汇总到一个 Reduce 实例中,从而在 Reduce 任务中对两个数据集的记录进行笛卡尔积运算,完成 Join 过程.Reduce-Join 具有普适性,当 x 和 y 数据集都很大时,会产生大量网络 I/O.Map-Join 是 Reduce-Join 的一种改进,减少了 Reduce 任务,从而消除了数据从 Map 任务进入 Reduce 任务的网络传输过程,但需要在

Map 任务开始前增加一个分发任务.当数据集 y 较小时,系统将 y 分发至每一个节点内存,或分发至分布式缓存中供每个节点访问.这样,在每个 Map 任务就可以参照 y 完成数据连接.当 x 数据量很大而 y 数据量很小时,Map-Join 方式的效率非常高;但是当 y 的数据量非常大时,分发阶段的 I/O 代价会很高.此外,现有研究在两种常用的连接算法上提出了基于半连接的 Semi-Join 算法.Semi-Join 在连接操作之前提取出用于连接的连接属性数据集($x.a$ 和 $y.a$),然后用此数据集对需要参加连接的数据集进行过滤,从而减少传输数据.半连接的实现方式很多,可以采用普通的 HashSet 来存储连接属性数据集,可以通过 BloomFilter[33]方式用连接属性数据集过滤数据,根据不同的场景可以采用不同的方式.Semi-Join 需要额外的 MapReduce 过程来完成半连接,如果连接属性数据集的记录数和原数据集的记录数相差无几,那么 Semi-Join 的优势将不会明显,且增加了一个 MapReduce 过程和大量额外的计算以及 I/O 操作.为评价 MapReduce 连接算法 I/O 代价,我们将算法划分为多个阶段,每个阶段执行同一类任务的多个实例.

定义 1(阶段和任务(phase and task)). MapReduce 连接算法由不同阶段组成,每个阶段内同一类型任务的实例并行执行.符号 D_i,M_i 和 R_i 分别表示数据分发阶段、Map 阶段和 Reduce 阶段,下标表示该阶段在算法中的次序(i 为自然数).任务则由 $Phrase_i$:$Function(\cdot)$的二元表达式组成,$Phrase_i$ 为阶段,$Function(\cdot)$为任务完成的运算.

分发任务 D:$mem(Input)$表示将 $Input$ 数据分发给每个节点,Map 任务和 Reduce 任务的定义参照 MapReduce 框架.如,M_2:$\sigma(S)$表示在第 2 个阶段中 Map 任务对 S 数据集进行查询,Θ_i 表示第 i 个任务的输出数据集.若用符号 $\rightarrow$分割算法的不同阶段,根据定义 1,Map-Join 可以形式化描述为

$$D_1:mem(y)\rightarrow M_2:\prod\nolimits_{y.b,x.c}(\sigma_{x.a=y.a\wedge C_x\wedge C_y}(x'\times y)) \tag{2}$$

公式(2)中,$mem(y)$表示把数据集 y 分发到每个 Mapper 的内存中,x'为数据集 x 的一个分片.Reduce-Join 可以形式化描述为

$$M_1:\begin{cases}\Theta_{1x}=\prod\nolimits_{x.a,x.c}(\sigma_{C_x}(x))\\ \Theta_{1y}=\prod\nolimits_{y.a,y.b}(\sigma_{Cy}(y))\end{cases}\rightarrow R_2:\sigma_{x.a=y.a}(\Theta_{1x}\times\Theta_{1y}) \tag{3}$$

基于定义 1,第 3 节将研究每个阶段的 I/O 代价特征.

3 I/O 代价模型

衡量算法的效率通常采用时间复杂度和空间复杂度两种指标,即算法执行需要的运算和存储.本文讨论的连接查询是数据密集型计算的一种,该类型计算有以下特点:当数据量超过一个限度时,存储系统难以满足海量数据处理的读写需要,数据传输带宽成为计算的一个瓶颈[27].另外,在数据连接过程中,对数据集的操作是确定的,结果集大小也是确定的,不论采用什么实现算法,其计算代价都近似相等[7].综上,I/O 代价是数据密集型计算的主要运算代价,在 I/O 资源稀缺的环境中,连接查询的效率主要由 I/O 代价决定.因此,本文选取 I/O 代价作为算法的评估指标.在具体介绍 I/O 代价模型前,先给出以下相关定义.

定义 2(连接率(join ratio)). 定义数据集 s 的连接率α_s为数据集 s 中参与连接的记录数与数据集 s 总记录数的比值.

设$|s|$计算数据集(结果集)的记录数,则 x 和 y 连接率可以表示为

$$\alpha_x=\frac{|\sigma_{x.a\ \mathrm{in}\ y.a}(x)|}{|x|},\alpha_y=\frac{|\sigma_{y.a\ \mathrm{in}\ x.a}(y)|}{|y|} \tag{4}$$

定义 3(选择率(selection ratio)). 定义数据集 s 的选择率β_s为满足 s 查询条件的记录数与 s 总记录数的比值:

$$\beta_s=\frac{|\sigma_{C_s}(s)|}{|s|} \tag{5}$$

定义 4(投影率(project ratio)). 定义数据集 s 的投影率γ_s为 s 投影后的数据量与 s 数据量的比值.

若近似认为每个属性大小相等,则 s 的投影率为投影后属性个数与投影前属性个数的比值.因此,对于形如

$\prod_P(\sigma_{C_s}(s))$ 的操作,$Attr(\cdot)$表示数据集的属性个数,则:

$$\gamma_s=\frac{Attr(P)}{Attr(s)} \tag{6}$$

为进一步描述 I/O 代价模型,表 1 描述了下文使用的符号.

Table 1 Description of notations

表 1 相关符号描述

符号	含义描述
N_m	Map 节点数
x	主表(较大的数据集)
y	从表(较小的数据集)
θ	连接查询后的记录条数
θ_x, θ_y	$x(y)$的记录条数
Φ	连接查询后数据量(KB)
ϕ	连接查询后的单条记录大小(KB)
ϕ_x, ϕ_y	$x(y)$的数据量(KB)
I_i, O_i	第 i 个任务输入(I)、以及输出(O)数据量(KB)
τ	单个连接属性大小(KB)
k	参加连接的不同连接属性的个数
α_x, α_y	$x(y)$的连接率
β_x, β_y	$x(y)$的选择率
γ_x, γ_y	$x(y)$的投影率
ε_x, ε_y	$x(y)$连接查询命中数据量与原有数据量之间的比例
λ	Reduce 过程中 sort 阶段磁盘数据的百分比
C_i^L	第 i 个阶段的磁盘 I/O 总量
C_i^N	第 i 个阶段的网络 I/O 总量
C^L	连接查询的磁盘 I/O 总量
C^N	连接查询的网络 I/O 总量

3.1 连接查询结果集大小

我们把连接查询作为一个整体计算结果数据集大小,连接查询结果的数据量Φ等于连接查询后的数据条数θ与连接查询后的单条记录大小ϕ的乘积,即$\Phi=\theta\times\phi$,本节研究其估算方法.在数据库领域现存若干连接查询结果集大小的估算算法[28−30],但这些算法在大数据环境下要么精确度低,要么性能差,难以适用于本文的连接优化算法.文献[28]指出,大部分查询处理代价模型都会做出以下 3 个假设:① 各个表中连接谓词涉及到的不同属性列之间相互独立;② 连接属性的不同值服从均匀分布;③ 参与连接的两个表的连接属性值集合之间符合包含关系.然而,文献[29]指出,基于假设的估算方法虽然速度快,但精确度得不到保证.因此,提出一种基于概率关系模型(probabilistic relational models)的估算方法,相对精确地估计 Key Join(连接属性分别是参与连接的两个表的主键和外键)结果集的大小.文献[30]则首先指出大部分商用查询优化器做出的不同属性间独立的假设通常是导致查询执行计划不是最优的主要原因,而后在没有该假设的情况下,基于图模型提出一种有效的相对精确的连接结果集大小估算方法,且该方法不仅限于 key join,它还适用于所有任意的 theta-join.然而,文献[29,30]中提出的两种方法虽然精确,但都是针对关系型数据库的,在大数据集上难以实现.因此,我们设计一种基于采样的方法对连接查询结果集的大小进行估计.现有采样方法包括随机采样、系统采样、分层采样等,其中:随机采样是最基本也最简单的采样方式,在保证采样的随机性和样本容量的情况下,能够较准确地反映总体数据的特征;其他采样方法大都需要关于数据特征的先验知识,采样代价也相对较高.考虑到本文研究的 MapReduce 连接查询针对的数据量通常是很大的,且事先不知道数据的特征,本文最终决定采用最简单的随机采样,虽然牺牲一定的准确度,但却极大地节省了采样代价.具体方法步骤如下:

第 1 步:对 x 和 y 表进行随机采样,获得采样数据集 x'和 y'.通过数据集 x'和 y 获得连接数据集 x'_α 和连接率 α_x,同理获得 y'_α 和α_y;通过数据集 x'和对 x 的查询条件获得查询数据集 x'_β 和查询率β_x,同理获得 y'_β 和β_y.

第 2 步:对于数据集 x,设其查询属性和连接属性之间的相关度为ω_x($0\leqslant\omega_x\leqslant1$,两者越相关,查询条件选中区

域和连接条件选中区域越重合),在采样数据集 x'上计算相关度ω_x,可以采用信息增益(information gain)计算相关度;同理可以获得ω_y.

第3步:对于任意数据集s,若α_s和β_s已知,设查询和连接选中的数据量与原有数据量之间的比例为ε_s,则最好情况下,连接命中区域和查询命中区域重合,ε_s=min(α_s,β_s);最坏情况下,连接命中区域和查询命中区域不重合,ε_s=0,即查询连接后得不到数据.更一般地,我们认为ε_s 是一个函数,与α_s,β_s 以及二者之间的相关度ω_s(0≤ω_s≤1)有关,函数表示如下:

$$\varepsilon_s=\begin{cases}\omega_s\cdot\min(\alpha_s,\beta_s), & \alpha_s+\beta_s\leqslant 1\\ \omega_s[\min(\alpha_s,\beta_s)-(\alpha_s+\beta_s-1)]+(\alpha_s+\beta_s-1), & \alpha_s+\beta_s>1\end{cases} \tag{7}$$

当α_s+β_s≤1 时,最好情况下,连接命中区域与查询命中区域重合,ε_s=min(α_s,β_s);最坏情况下,连接命中区域与查询命中区域不重合,ε_s=0(如图 2(a)所示).当α_s+β_s>1 时,最好情况下,连接命中区域与查询命中区域重合,ε_s=min(α_s,β_s);最坏情况下,连接命中区域与查询命中区域重合部分最小,ε_s=α_s+β_s−1(如图 2(b)所示).公式(7)中,ω_s 表示查询条件和连接条件的相关度(0≤ω_s≤1),两者越相关,查询条件选中区域和连接条件选中区域越重合.

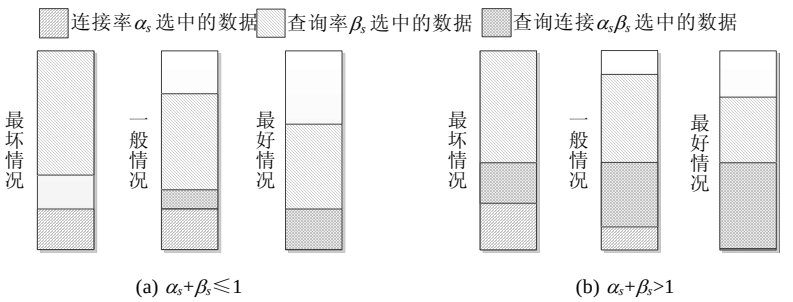


Fig.2 Sketch for explain Eq.(7)

图 2 公式(7)的示意图

通过公式(7)可以计算获得ε_x和ε_y.那么,x 表和 y 表经选择条件和连接条件作用后的数据记录条数分别为

$$\theta'_x=\theta_x\cdot\varepsilon_x,\theta'_y=\theta_y\cdot\varepsilon_y.$$

第 4 步:设 X 为 x 数据集中不同的连接属性值集合,Y 为 y 数据集中不同的连接属性值集合.对于 x 表,采用数据集 x'_β 获得经查询条件过滤后的连接属性的数据分布:对于$\forall t\in X\cup Y$,t 在 x'_β 数据集中出现的概率为 p_x^t:

- 当 $t\in X$ 时, p_x^t 服从该分布;
- 当 $t\notin X$ 时, p_x^t=0.

同理获得 p_y^t .那么,可按公式(8)计算连接查询后的数据集记录条数θ:

$$\theta=\sum_{t\in X\cup Y}(p_x^t\cdot\theta'_x)\cdot(p_y^t\cdot\theta'_y)=(\theta_x\cdot\varepsilon_x\cdot\theta_y\cdot\varepsilon_y)\sum_{t\in X\cup Y}p_x^t\cdot p_y^t \tag{8}$$

第 5 步:计算ϕ的值,连接查询后的单条记录大小ϕ与连接属性是否是投影属性有关,分以下两种情况计算:

$$\phi=\begin{cases}\gamma_x\dfrac{\phi_x}{\theta_x}+\gamma_y\dfrac{\phi_y}{\theta_y}-\tau, & a\in P\\ \gamma_x\dfrac{\phi_x}{\theta_x}+\gamma_y\dfrac{\phi_y}{\theta_y}-2\tau, & a\notin P\end{cases} \tag{9}$$

第 6 步:参照公式(8)和公式(10),计算连接查询后的数据量Φ=θ×ϕ:

$$\Phi=\theta\cdot\phi=\left((\theta_x\cdot\varepsilon_x\cdot\theta_y\cdot\varepsilon_y)\sum_{t\in X\cup Y}p_x^t\cdot p_y^t\right)\cdot\left(\gamma_x\frac{\phi_x}{\theta_x}+\gamma_y\frac{\phi_y}{\theta_y}-2^n\cdot\tau\right),n=\begin{cases}0, & a\in P\\ 1, & a\notin P\end{cases} \tag{10}$$

按上述步骤,可以估算连接查询后数据集的大小.举例说明:如果经查询条件过滤后的数据集上,连接属性的每个值出现的频率服从常见的 Zipf 分布[31],则第 i 个最多出现的连接属性值出现的概率为 $p_i=\frac{1}{i\times H_k}$,其中,$1\leqslant i\leqslant k$,$H_k$ 是调和系数,可近似为$(\gamma+\ln k)$[26],γ=0.57721,为欧拉常数.由此可计算出两数据集查询连接投影后的记录条数:

$$\theta=(\theta_x\cdot\varepsilon_x\cdot\theta_y\cdot\varepsilon_y)\sum_{i=1}^{k}p_x^i\cdot p_y^j=\frac{\theta_x\cdot\varepsilon_x\cdot\theta_y\cdot\varepsilon_y}{(\gamma+\ln k)^2}\sum_{i=1}^{k}\frac{1}{i\cdot j} \tag{11}$$

其中,j 表示 x'_β 中第 i 个最多出现的连接属性值在 y'_β 中的频数排名.此时,如果两个数据集连接属性的分布一致$(i=j)$,那么在海量数据环境中,$\sum_{i=1}^{k}\frac{1}{i^2}$ 可取极限值$\pi^2/6$,则θ计算如下:

$$\theta=\frac{\theta_x\cdot\varepsilon_x\cdot\theta_y\cdot\varepsilon_y}{(\gamma+\ln k)^2}\sum_{i=1}^{k}\frac{1}{i^2}=\frac{\theta_x\cdot\varepsilon_x\cdot\theta_y\cdot\varepsilon_y\cdot\pi^2}{6(\gamma+\ln k)^2} \tag{12}$$

如果数据集的连接属性服从的是其他分布,也可用类似的方法来求解θ值.

3.2 连接查询各阶段I/O代价

本节首先给出连接查询算法各个阶段的 I/O 代价模型,第 4 节则根据具体算法对每个阶段的 I/O 代价进行汇总.I/O 代价的计算基于 MapReduce 框架的数据读写情况,如图 3 所示(MapReduce 的内部运行机制见文献[5]).本文给出的连接查询算法的 I/O 代价仅包括最终结果写入本地磁盘的代价,而不包括其写入分布式文件系统的代价.这是因为对于所有算法,后者的代价是一致的,且与具体的文件系统相关.我们设第 i 个任务的输入(I)以及输出(O)数据量分别为 I_i 和 O_i.

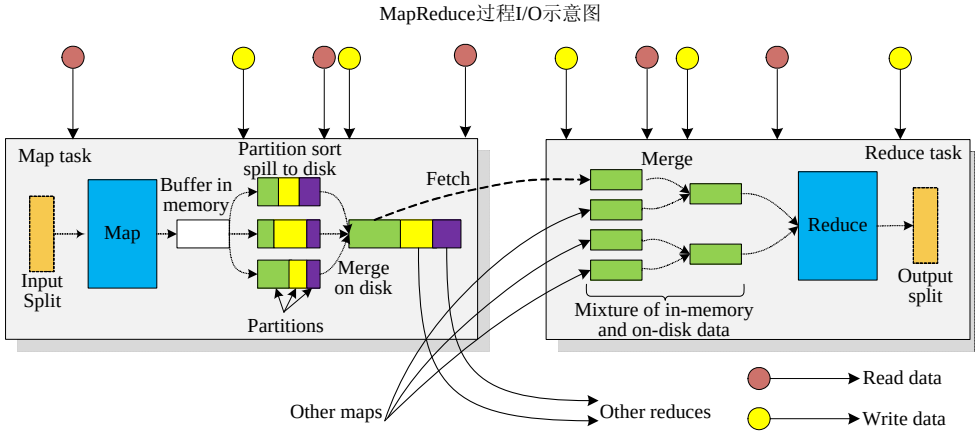


Fig.3 Data reading and wirting in MapReduce process

图 3 MapReduce 过程数据读写

对于分发阶段的数据分发任务 O_D=D:$mem(I_D)$,需要本地读取数据一次,而后经过网络派发(N_m-1)次,由此可计算出该阶段的本地 I/O 量(C_D^L)和网络 I/O 量(C_D^N),见公式(13):

$$C_D^L=I_D,C_D^N=(N_m-1)\cdot I_D \tag{13}$$

对于 Map 任务 O_M=M:$map(I_M)$,需要本地读取数据一次,对其进行相应的操作后,生成 Map 的结果数据集.由图 3 可知,结果数据集共经历了 3 次读写,而在 Map 阶段中,并无网络传输发生,由此可计算出该阶段的本地 I/O 量(C_M^L)和网络 I/O 量(C_M^N),见公式(14):

$$C_M^L=I_M+3O_M,C_M^N=0 \tag{14}$$

若 Map 任务后没有 Reduce 任务,则只需本地读取数据一次,对其进行相应的操作后,将生成的结果数据集

写入分布式文件系统,由此计算出该阶段的本地 I/O 量(C'^{L}_{M})和网络 I/O 量(C'^{N}_{M}),见公式(15):

$$C'^{L}_{M}=I_M+O_M, C'^{N}_{M}=0 \tag{15}$$

对于 Reduce 任务 O_R=R:$reduce$(I_R),Reducer 在 shuffle 阶段先将 Mapper 产生的中间数据集通过网络复制到本地,并且经历了一次内存和磁盘混合式的排序和合并,最终输入到 Reduce 函数中进行相应的操作,并将生成的输出数据集写入分布式文件系统.很明显,此处的输入数据集 I_R 应为 Mapper 输出数据集 O_M,由此计算出该阶段的本地 I/O 量(C^{L}_{R})和网络 I/O 量(C^{N}_{R}),见公式(16):

$$C^{L}_{R}=(1+\lambda)\cdot I_R+O_R, C^{N}_{R}=I_R \tag{16}$$

连接查询算法的 I/O 分本地 I/O 和网络 I/O 两种,二者代价不同.因此,本文为它们分别设置了权重,将二者的加权和作为算法整体的 I/O 代价(见公式(17)),并以此作为算法选择的依据.因此,假设 MapReduce 模型有 n 的阶段,那么其 I/O 代价为

$$Cost=\omega_1\cdot C^{L}+\omega_2\cdot C^{N}=\sum_{i=1}^{n}(\omega_1\cdot C^{L}_{i}+\omega_2\cdot C^{N}_{i}) \tag{17}$$

公式(17)中,ω_1+ω_2=1,其具体取值由运行环境以及具体需求确定.本地每页数据平均访问代价(与 ω_1 有关)与网络单位数据传输代价(与 ω_2 有关)的比值在广域网环境下为 1:20,局域网环境下为 1:1.6.可见,在广域网环境,以传输代价为主;在局域网环境,需综合考虑传输代价和本地代价.大部分 MapReduce 集群均采局域网络环境,因此,ω_1:ω_2=1:1.6.

4 二元连接 I/O 代价

根据第 2 节描述的二元连接的 Map-Join,Reduce-Join 和 Semi-Join 算法,本节设计了 6 种基于 MapReduce 的连接查询算法:M,M^2,M^3R,RM,RM^2 和 R^2M^2,并逐一分析它们的 I/O 代价.算法命名时,采用 M 表示 Map 任务,R 表示 Reduce 任务,指数表示 Map 和 Reduce 任务的次数,M 在前的为 Map-Join,R 在前的为 Reduce-Join.按第 2 节公式(1)设计的查询用例如下述 SQL 所示,数据集 x 包含属性 a,c 等,数据集 y 包含属性 a,b 等;两数据集通过属性 a 等值连接,其中,C_x 和 C_y 分别表示与 x 和 y 相关的查询条件.

Select $y.b$, $x.c$ **From** x,y **Where** $x.a$=$y.a$ **and** C_x **and** C_y.

本节余下部分将具体介绍上述 6 种算法,参照定义 1,按阶段描述算法内容,并根据第 3 节描述,推导它们的 I/O 代价函数.连接查询的实现算法与用例相关,对于有些算法,如 M^3R,若投影或选择条件满足一定条件,算法会随之简化.此外,通过 Map-Reduce-Merge[8],MapChain 或 ReduceChain 技术也可以不同程度地优化这些算法,但本文认为,这种优化没有改变算法的本质,不作为新算法来考虑.基于 MapReduce 的二元连接算法不限于这 6 种,本文研究重点是连接查询的 I/O 代价而不是二元连接查询算法本身,对于一些算法的变形,仍可以按本节描述的方法得出其 I/O 代价函数.

4.1 M算法

M 算法即为 Map-Join 算法,需要一次分发任务、一次 Map 任务.M 算法要求连接操作中较小的数据集 y 可以装入内存.M 算法的形式化描述为公式(18),其中,mem(y)表示把数据集 y 分发到每个 Mapper 的内存中,x'为数据集 x 的一个分片:

$$D_1:mem(y)\rightarrow M_2:\prod\nolimits_{y.b,x.c}(\sigma_{x.a=x.a\wedge C_x\wedge C_y}(x'\times y)) \tag{18}$$

在 D_1 阶段,I_1=O_1=ϕ_y;M_2 阶段,I_2=ϕ_x,O_2=Φ.带入公式(12)~公式(15),各阶段 I/O 代价函数情况见公式(19):

$$\begin{cases}C^{L}_{1}=\phi_y\\ C^{N}_{1}=(N_m-1)\cdot\phi_y\end{cases},\begin{cases}C^{L}_{2}=\phi_x+\Phi\\ C^{N}_{2}=0\end{cases} \tag{19}$$

汇总以上各个阶段的 I/O 代价,可以得到整个过程的 I/O 代价函数,见公式(20):

$$\begin{cases} C^L=\sum_{i=1}^{2}C_i^L=\phi_x+\phi_y+\Phi \\ C^N=\sum_{i=1}^{2}C_i^N=(N_m-1)\cdot\phi_y \end{cases} \tag{20}$$

4.2 M^{3}R算法

M^{3}R 算法是 Semi-Join 和 Map-Join 的结合,需要 2 次分发任务、3 次 Map 任务和 1 次 Reduce 任务.M 算法最明显的局限是 y 必须足够小以装入 Map 节点内存.若 y 较大,则可以先对 y 进行过滤以减少数据量.M^{3}R 算法采用半连接的方式过滤 y.首先,通过一组 Map-Reduce 任务抽取 x 和 y 的连接属性 a(公式(21)中的Θ_{1x}和Θ_{1y}),进行连接操作后,输出连接属性数据集Θ_2保存到分布式存储中;接着分发Θ_2,并通过 Map 任务完成 y 的选择过滤操作,输出结果数据集;最后分发Θ_4,完成 Map 端连接操作.M^{3}R 算法的形式化描述为

$$\begin{aligned} &M_1:\begin{cases}\Theta_{1x}=\prod_{x.a}(x)\\ \Theta_{1y}=\prod_{y.a}(y)\end{cases}\rightarrow R_2:\prod_{y.a}\sigma_{x.a=y.a}(\Theta_{1x}\times\Theta_{1y})\rightarrow D_3:mem(\Theta_2)\rightarrow \\ &M_4:\prod_{y.a,y.b}(\sigma_{y.a\ \text{in}\ \Theta_2\wedge C_y}(y))\rightarrow D_5:mem(\Theta_4)\rightarrow M_6:\prod_{y.b,x.c}(\sigma_{x.a=y.a\wedge C_x}(\Theta'_x\times\Theta_4)) \end{aligned} \tag{21}$$

M_1 阶段,$I_1=\phi_x+\phi_y$,$O_1=(\theta_x+\theta_y)\cdot\tau$;$R_2$ 阶段,$I_2=O_1=(\theta_x+\theta_y)\cdot\tau$,$O_2=k\cdot\tau$;$D_3$ 阶段,$I_3=O_2=O_3=k\cdot\tau$;M_4 阶段,$I_4=\phi_y$,$O_4=\phi_y\cdot\varepsilon_y\cdot\gamma_y$;$D_5$ 阶段,$I_5=O_4=O_5$;M_6 阶段,$I_6=\phi_x$,$O_6=\Phi$.带入公式(12)~公式(15),各阶段 I/O 代价函数见公式(22):

$$\begin{cases} C_1^L=\phi_x+\phi_y+3(\theta_x+\theta_y)\cdot\tau \\ C_1^N=0 \end{cases},\begin{cases} C_2^L=(1+\lambda)\cdot(\theta_x+\theta_x)\cdot\tau+k\cdot\tau \\ C_2^N=(\theta_x+\theta_y)\cdot\tau \end{cases},\begin{cases} C_3^L=k\cdot\tau \\ C_3^N=(N_m-1)\cdot k\cdot\tau \end{cases},$$
$$\begin{cases} C_4^L=\phi_y+\phi_y\cdot\varepsilon_y\cdot\gamma_y \\ C_4^N=0 \end{cases},\quad \begin{cases} C_5^L=\phi_y\cdot\varepsilon_y\cdot\gamma_y \\ C_5^N=(N'_M-1)\cdot\phi_y\cdot\varepsilon_y\cdot\gamma_y \end{cases},\quad \begin{cases} C_6^L=\phi_x+\Phi \\ C_6^N=0 \end{cases} \tag{22}$$

汇总以上各个阶段的 I/O 代价,可以得到整个过程的 I/O 代价函数,见公式(23):

$$\begin{cases} C^L=\sum_{i=1}^{6}C_i^L=2(\phi_x+\phi_y)+(4+\lambda)\cdot(\theta_x+\theta_y)\cdot\tau+2k\cdot\tau+2\phi_y\cdot\varepsilon_y\cdot\gamma_y+\Phi \\ C^N=\sum_{i=1}^{6}C_i^N=(\theta_x+\theta_y)\cdot\ell+(N_m-1)\cdot k\cdot\tau+(N'_M-1)\cdot\phi_y\cdot\varepsilon_y\cdot\gamma_y \end{cases} \tag{23}$$

4.3 M^{2}算法

M^2 算法是本文对 Map-Join 算法做的简单改进,需要 2 次 Map 任务和 1 次分发任务.M^2 算法采用查询的方式过滤 y,在第 1 阶段仅提取从表所有的连接属性,而后用从表查询条件过滤连接属性,组成连接属性数据集;最后采用 Map-Join 实现连接操作.该算法没有采用半连接的方式,牺牲了连接属性数据集的精确性,但却省略了一次对主表的扫描.当从表上查询条件能够过滤从表大量数据时,M^2 最为有效;当从表上无查询条件时,则退化成 M 算法.M^2 算法的形式化描述为

$$M_1:\prod_{y.a,y.b}(\sigma_{C_y}(y))\rightarrow D_2:mem(\Theta_1)\rightarrow M_3:\prod_{y.b,x.c}(\sigma_{x.a=y.a\wedge C_x}(x'\times\Theta_2)) \tag{24}$$

M_1 阶段,$I_1=\phi_y$,$O_1=\phi_y\cdot\beta_y\cdot\gamma_y$;$D_2$ 阶段,$I_2=O_2=O_1$;M_3 阶段,$I_3=\phi_x$,$O_3=\Phi$.带入公式(12)~公式(15),各阶段 I/O 代价函数见公式(25):

$$\begin{cases} C_1^L=\phi_y+\phi_y\cdot\beta_y\cdot\gamma_y \\ C_1^N=0 \end{cases},\begin{cases} C_2^L=\phi_y\cdot\beta_y\cdot\gamma_y \\ C_2^N=(N_m-1)\cdot\phi_y\cdot\beta_y\cdot\gamma_y \end{cases},\begin{cases} C_3^L=\phi_x+\Phi \\ C_3^N=0 \end{cases} \tag{25}$$

汇总以上各个阶段的 I/O 代价,可以得到整个过程的 I/O 代价函数,见公式(26):

$$\begin{cases} C^L=\sum_{i=1}^{3}C_i^L=\phi_x+\phi_y+2\phi_y\cdot\beta_y\cdot\gamma_y+\Phi \\ C^N=\sum_{i=1}^{3}C_i^N=(N_m-1)\cdot\phi_y\cdot\beta_y\cdot\gamma_y \end{cases} \tag{26}$$

4.4 RM算法

RM 算法即为 Reduce-Join 算法[15],通过一组 Map-Reduce 完成选择和连接操作.RM 算法的形式化描述为

$$M_1:\begin{cases}\Theta_{1x}=\prod_{x.a,x.c}(\sigma_{C_x}(x))\\ \Theta_{1y}=\prod_{y.a,y.b}(\sigma_{C_y}(y))\end{cases}\rightarrow R_2:\sigma_{x.a=y.a}(\Theta_{1x}\times\Theta_{1y}) \tag{27}$$

M_1 阶段,$I_1=\phi_x+\phi_y$,$O_1=\phi_x\cdot\beta_x\cdot\gamma_x+\phi_y\cdot\beta_y\cdot\gamma_y$;$R_2$ 阶段,$I_2=O_1$,$O_3=\Phi$.带入公式(12)~公式(15),各阶段 I/O 代价函数见公式(28):

$$\begin{cases}C_1^L=\phi_x+\phi_y+3(\phi_x\cdot\beta_x\cdot\gamma_x+\phi_y\cdot\beta_y\cdot\gamma_y)\\ C_1^N=0\end{cases},\begin{cases}C_2^L=(1+\lambda)\cdot(\phi_x\cdot\beta_x\cdot\gamma_x+\phi_y\cdot\beta_y\cdot\gamma_y)+\Phi\\ C_2^N=\phi_x\cdot\beta_x\cdot\gamma_x+\phi_y\cdot\beta_y\cdot\gamma_y\end{cases} \tag{28}$$

汇总以上各个阶段的 I/O 代价,可以得到整个过程的 I/O 代价函数,见公式(29):

$$\begin{cases}C^L=\sum_{i=1}^{2}C_i^L=\phi_x+\phi_y+(4+\lambda)\cdot(\phi_x\cdot\beta_x\cdot\gamma_x+\phi_y\cdot\beta_y\cdot\gamma_y)+\Phi\\ C^N=\sum_{i=1}^{2}C_i^N=\phi_x\cdot\beta_x\cdot\gamma_x+\phi_y\cdot\beta_y\cdot\gamma_y\end{cases} \tag{29}$$

4.5 R^2M^2算法

R^2M^2 算法是 Semi-Join 和 Reduce-Join 的结合,需要 1 次分发任务、2 次 Map 任务和 2 次 Reduce 任务,类似 M^3R 算法.R^2M^2 算法的形式化描述为

$$\begin{aligned}&M_1:\begin{cases}\Theta_{1x}=\prod_{x.a}(x)\\ \Theta_{1y}=\prod_{y.a}(y)\end{cases}\rightarrow R_2:\prod_{y.a}(\sigma_{x.a=y.a}(\Theta_{1x}\times\Theta_{1y}))\rightarrow D_3:mem(\Theta_2)\rightarrow\\ &M_4:\begin{cases}\Theta_{4x}=\prod_{x.a,x.c}(\sigma_{x.a\text{ in }\Theta_2\wedge C_x}(x))\\ \Theta_{4y}=\prod_{y.a,y.b}(\sigma_{y.a\text{ in }\Theta_2\wedge C_y}(y))\end{cases}\rightarrow R_5:\sigma_{x.a=y.a}(\Theta_{4x}\times\Theta_{4y})\end{aligned} \tag{30}$$

M_1 阶段,$I_1=\phi_x+\phi_y$,$O_1=(\theta_x+\theta_y)\cdot\tau$;$R_2$ 阶段,$I_2=O_1$,$O_2=k\cdot\tau$;D_3 阶段,$I_3=O_2=O_3$;M_4 阶段,$I_4=\phi_x+\phi_y$,$O_4=\phi_x\cdot\varepsilon_x\cdot\gamma_x+\phi_y\cdot\varepsilon_y\cdot\gamma_y$;$R_5$ 阶段,$I_5=O_4$,$O_5=\Theta$.带入公式(12)~公式(15),各阶段 I/O 代价函数见公式(31):

$$\begin{aligned}&\begin{cases}C_1^L=\phi_x+\phi_y+3(\theta_x+\theta_y)\cdot\tau\\ C_1^N=0\end{cases},\begin{cases}C_2^L=(1+\lambda)\cdot(\theta_x+\theta_y)\cdot\tau+k\cdot\tau\\ C_2^N=(\theta_x+\theta_y)\cdot\tau\end{cases},\begin{cases}C_3^L=k\cdot\tau\\ C_3^N=(N_m-1)\cdot k\cdot\tau\end{cases},\\ &\begin{cases}C_4^L=\phi_x+\phi_y+3(\phi_x\cdot\varepsilon_x\cdot\gamma_x+\phi_y\cdot\varepsilon_y\cdot\gamma_y)\\ C_4^N=0\end{cases},\begin{cases}C_5^L=(1+\lambda)\cdot(\phi_x\cdot\varepsilon_x\cdot\gamma_x+\phi_y\cdot\varepsilon_y\cdot\gamma_y)+\Phi\\ C_5^N=\phi_x\cdot\varepsilon_x\cdot\gamma_y+\phi_y\cdot\varepsilon_y\cdot\gamma_y\end{cases}\end{aligned} \tag{31}$$

汇总以上各个阶段的 I/O 代价,可以得到整个过程的 I/O 代价函数,见公式(32):

$$\begin{cases}C^L=\sum_{i=1}^{5}C_i^L=2(\phi_x+\phi_y)+(4+\lambda)\cdot(\theta_x+\theta_y)\cdot\tau+2k\cdot\tau+(4+\lambda)\cdot(\phi_x\cdot\varepsilon_x\cdot\gamma_x+\phi_y\cdot\varepsilon_y\cdot\gamma_y)+\Phi\\ C^N=\sum_{i=1}^{5}C_i^N=(\theta_x+\theta_x)\cdot\tau+(N_m-1)\cdot k\cdot\tau+(\phi_x\cdot\varepsilon_x\cdot\gamma_x+\phi_Y\cdot\varepsilon_y\cdot\gamma_y)\end{cases} \tag{32}$$

4.6 RM^2算法

RM^2 算法是本文对 Reduce-Join 算法做的简单改进,算法需要 1 次分发任务、2 次 Map 任务和 1 次 Reduce 任务.首先,通过 Map 任务抽取较小的数据集 y 的连接属性数据集,并将其分发给各个 Mapper;随后,参照 y 的连接属性数据集,通过一组 Map-Reduce 任务完成查询和 Reduce 端连接操作.RM^2 算法的形式化描述为

$$M_1:\prod_{y.a}(y)\rightarrow D_2:mem(\Theta_1)\rightarrow M_3:\begin{cases}\Theta_{3x}=\prod_{x.a,x.c}(\sigma_{x.a\text{ in }\Theta_1\wedge C_x}(x))\\ \Theta_{3y}=\prod_{y.a,y.b}(\sigma_{C_y}(y))\end{cases}\rightarrow R_4:\sigma_{x.a=y.a}(\Theta_{3x}\times\Theta_{3y}) \tag{33}$$

若 D_2 阶段 Θ_1 过大不能放入内存,则可以通过查询或 *distinct* 操作过滤数据.当然,这些优化方法并未更改算法的流程.

M_1阶段,$I_1=\phi_y,O_1=\theta_y\cdot\tau;D_2$阶段,$I_2=O_1=O_2=\theta_y\cdot\tau;M_3$阶段,$I_3=\phi_x+\phi_y,O_3=\phi_x\cdot\varepsilon_x\cdot\gamma_X+\phi_y\cdot\beta_y\cdot\gamma_y;R_4$阶段,$I_4=O_3,O_4=\Phi$.带入公式(12)~公式(15),各阶段 I/O 代价函数见公式(34):

$$\begin{cases}C_1^L=\phi_y+\theta_y\cdot\tau\\C_1^N=0\end{cases},\quad\begin{cases}C_2^L=\theta_y\cdot\tau\\C_2^N=(N_m-1)\cdot\theta_y\cdot\tau\end{cases},$$
$$\begin{cases}C_3^L=\phi_x+\phi_y+3(\phi_y\cdot\beta_y\cdot\gamma_y+\phi_x\cdot\varepsilon_x\cdot\gamma_x)\\C_3^N=0\end{cases},\begin{cases}C_4^L=(1+\lambda)\cdot(\phi_y\cdot\beta_y\cdot\gamma_y+\phi_x\cdot\varepsilon_x\cdot\gamma_x)+\Phi\\C_4^N=\phi_x\cdot\varepsilon_x\cdot\gamma_x+\phi_y\cdot\beta_y\cdot\gamma_y\end{cases}\tag{34}$$

汇总以上各个阶段的 I/O 代价,可以得到整个过程的 I/O 代价函数,见公式(35):

$$\begin{cases}C^L=\sum_{i=1}^{4}C_i^L=(\phi_x+2\phi_y)+2\theta_y\cdot\tau+(4+\lambda)\cdot(\phi_y\cdot\beta_y\cdot\gamma_y+\phi_x\cdot\varepsilon_x\cdot\gamma_x)+\Phi\\C^N=\sum_{i=1}^{4}C_i^N=(N_m-1)\cdot\theta_y\cdot\tau+(\phi_x\cdot\varepsilon_x\cdot\gamma_x+\phi_y\cdot\beta_y\cdot\gamma_y)\end{cases}\tag{35}$$

5 多元连接执行计划选择

查询优化是数据管理系统的核心问题.对于传统数据库,若数据表按照连接属性进行分区,能够最大程度地优化连接查询的性能.但是在海量数据环境下,维护良好数据分区的代价很高;而且针对各种查询,我们难以找到一种最合适的分区方法.因此,本文认为,数据尚未良好分区.此外,由于数据是海量的,因此我们近似认为:数据量均匀分布在每个节点上,并且属性的值域也均匀分布.在此基础上,我们研究基于代价的执行计划选择,对于多元连接,执行计划的选择即为连接顺序的选择,而查询代价即为前文得出的 I/O 代价模型和代价函数,我们研究多元连接查询的 I/O 代价最小化的执行计划.

等值连接与自然连接是应用最广泛的连接操作,自然连接与等值连接无本质区别,可以通过修改属性名把等值连接操作转化为自然连接操作,所以研究自然连接具有普遍意义.本节以自然连接($\bowtie$)为例,设 R 和 S 是两个数据集,$attr(\cdot)$表示数据集的属性集,自然连接 $R\bowtie S$ 定义 $attr(R)\cup attr(S)$上的一个关系,为

$$\prod\nolimits_{attr(R)\cup attr(S)}(\sigma_{r.a_1=s.a_1\wedge r.a_2=s.a_2\wedge\ldots\wedge r.a_n=s.a_n}(R\times S)),attr(R)\cap attr(S)=\{a_1,a_2,...,a_n\}.$$

特殊地,当 $attr(R)\cap attr(S)=\bowtie$,$R\bowtie S=R\times S$.对于链式连接,可以表示为 S_1~S_n 逐一连接所得结果,如公式(36);对于星型连接,可以看做 R 为事实数据集,S_1 至 S_n 为维数据集,每个维数据都和事实数据进行连接,得到的结果集再按链式连接,如公式(37):

$$((S_1(a_0,a_1)\bowtie S_2(a_1,a_2))\bowtie\ldots\bowtie S_n(a_{n-1},a_n)\tag{36}$$

$$[R(a_1,a_2,\ldots,a_n)\bowtie S_1(a_1)]\bowtie[R(a_1,a_2,\ldots,a_n)\bowtie S_2(a_2)]\bowtie\ldots\bowtie[R(a_1,a_2,\ldots,a_N)\bowtie S_n(a_n)]\tag{37}$$

然而,可证$\bowtie$满足以下性质:幂等律 $S\bowtie S=S$,交换律 $R\bowtie S=S\bowtie R$,结合律$(R\bowtie S)\bowtie T=R\bowtie(S\bowtie T)$,零律 $R\bowtie\bowtie=\bowtie$.因此,当公式(35)和公式(36)中任意一个数据集为空,则整个连接结果为空;否则,星型和链式连接均可化简为 $S_1\bowtie S_2\bowtie\ldots\bowtie S_n$ 的形式.推广到任意多元连接,均可化简为 $S_1\bowtie S_2\bowtie\ldots\bowtie S_n$ 的形式,且连接结果与连接顺序无关.对于 n 个数据集的多元连接 $S_1\bowtie S_2\bowtie\ldots\bowtie S_n$,有 $n!$种排列,每种排列有 $C_{(n-1)}$种组合方式(Catalan 数),因此有 $n!\times\frac{1}{n}\binom{2(n-1)}{n-1}=\frac{(2(n-1))!}{(n-1)!}$ 种执行计划.又因为$\bowtie$满足交换律,所以对于 n 个数据集,每种执行计划会有 2^{n-1} 个等价表达,因此,完全不等价的执行计划共有 $n!\times\frac{1}{n}\binom{2(n-1)}{n-1}\div 2^{n-1}=\frac{(2(n-1))!}{2^{n-1}\cdot(n-1)!}$ 种.例如 n=3 时,执行计划共有 12 种(见公式(38)),但是去重后共有 $12/2^{3-1}$=3 种.

$$\begin{aligned}&S_1\bowtie(S_2\bowtie S_3);S_1\bowtie(S_3\bowtie S_2);(S_2\bowtie S_3)\bowtie S_1;(S_3\bowtie S_2)\bowtie S_1;S_2\bowtie(S_1\bowtie S_3);S_2\bowtie(S_3\bowtie S_1);\\&(S_1\bowtie S_3)\bowtie S_2;(S_3\bowtie S_1)\bowtie S_2;S_3\bowtie(S_1\bowtie S_2);S_3\bowtie(S_2\bowtie S_1);(S_1\bowtie S_2)\bowtie S_3;(S_2\bowtie S_1)\bowtie S_3\end{aligned}\tag{38}$$

由于连接运算结果是确定的,因此无论采用何种顺序执行多元连接,结果集大小是一致的.因此,我们需找到这些执行计划中 I/O 代价最小的作为最优执行计划.较直接的算法为逐一计算每个执行计划的 I/O 代价,搜索 I/O 代价最小的执行计划.首先,对于任意一个执行计划,需要逐一计算 $S_i\bowtie S_j$ 的结果集大小 Φ_{ij} 和 6 种算法的 I/O

代价,选取 I/O 代价最小的算法作为 $S_i \bowtie S_j$ 的执行算法.那么对于一个 n 元连接的执行计划选择,至少要计算 $M\times(n-1)\times\frac{(2(n-1))!}{2^{n-1}\cdot(n-1)!}$ 次连接代价(M 为连接算法的种类,本文为 6),并在 $\frac{(2(n-1))!}{2^{n-1}\cdot(n-1)!}$ 个执行计划中搜索最小代价.

n 元连接执行计划选择算法是一个典型的搜索算法,当 n 较小时,执行计划选择算法是可行的,如 n=5 时,一共有 105 种执行计划;而当 n=7 时就有 10 395;当 n=10 时就会过亿,搜索空间过大.因此,需要找到一种更加简便的方法.我们采用动态规划算法求解,动态规划算法将待求解的问题分解为若干个子问题(阶段),按顺序求解子问题,前一子问题的解,为后一子问题的求解提供了有用的信息.在求解任意子问题时,列出各种可能的局部解,通过决策保留那些有可能达到最优的局部解,丢弃其他局部解.在 n 元连接执行计划选择问题上,可以通过划分子问题来减小求解的复杂性.例如当 n=5,假设我们已知 S_1,S_2,S_3 首先以某种顺序连接,得到的结果再和 S_4,S_4 以某种顺序连接,这样的执行计划会优于其他计划.因此,搜索空间由 $\frac{(2(5-1))!}{2^{5-1}(5-1)!}=105$ 降为 $\frac{(2(3-1))!}{(3-1)!}+\frac{(2(3-1))!}{(3-1)!}=24$.基于这一特性,分解的子问题就是$\{S_1,\ldots,S_n\}$的子集$\{S_1,\ldots,S_i\}$和$\{S_i,\ldots,S_n\}$的最优执行计划.如果子问题求解,当前连接的代价就为 $\Phi_{1-i}\bowtie\Phi_{i-n}$ 的代价.n 元连接执行计划选择算法如算法 1 所述.

算法 1. n 元连接执行计划选择算法.

Input: $\Omega=\{S_1,\ldots,S_n\}$, S_i is a dataset;

Return: $Bp[J]\langle plan,cost\rangle$ //*Bp* is best-query-plan, include three part, *J* is set of involved dataset, *plan* is the best-query-plan, *cost* is the cost of best-query-plan.

Globe

Function *FindBestPlan*(*J*)

1. **If** (*J.size*==1) //当 Join 只包含一个数据集时,为初始化阶段
2. *Bp*[*J*].*plan*=∅;
3. *Bp*[*J*].*cost*=0;
4. **Return**;
5. **EndIf**
6. **ElseIf** (*J*!=*Ω*) //当 Join 包含的数据集是所有数据集时,执行计划已经获得
7. **Foreach** $J_1\subset J$, J_1!=∅ //*J* 所有非空子集
8. *Bp*[J_1]=*FindBestPlan*(J_1);
9. *Bp*[J_2]=*FindBestPlan*($J-J_1$);
10. *JoinAlgorithm MR=FindBestAlgorithm*(·) //根据前文介绍的 I/O 代价函数
11. *Costcost=Bp*[J_1].*cost*+*Bp*[J_2].*cost*+*cost*(*JA*).
12. **If** (*cost*<*Bp*[*J*].*cost*)
13. *Bp*[*J*].*cost=cost*;
14. *Bp*[*J*].*plan=Bp*[J_1].*plan*, *Bp*[J_2].*plan*, and $\Phi_{J_1}\bowtie\Phi_{J_2}$ by *MR*
15. **EndIf**
16. **EndFor**
17. **EndElse**
18. **Return***Bp*[*J*];

在获得最佳执行计划以后,最后讨论计划的执行方式问题.n 元连接查询的执行计划可以看做一棵满二叉树,其中,叶子节点为 n 个数据集,根节点为⋈.可以采用并行的方式执行 n–1 个二元连接,也可以采用中根序遍历二叉树并顺序执行各 n–1 个二元连接.如图 4 所示.

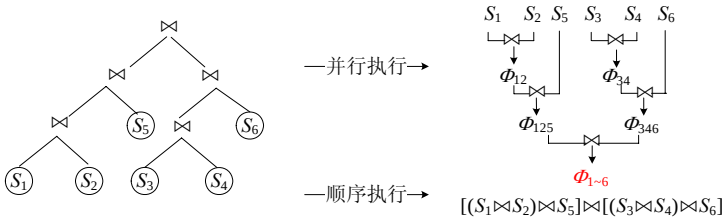


Fig.4 Parallel or sequential execution plan of joined query

图 4 并行或顺序执行的查询执行计划

我们通过前期实验发现,在一个MapReduce集群中同时执行多个相同类型的作业(如数据密集型作业)的时间代价大于顺序执行他们的时间代价.这是因为:① MapReduce 环境每个作业本身就是并行执行的,因此,保证多作业的并行性同时会降低每个作业的并行性,多作业并行并没有带来额外的性能提升;② 若 I/O 密集型和 CPU 密集型作业并行执行,则有助于提高资源利用率,提高性能,若多个 I/O 密集型作业并行执行,每个作业需要的资源不能互补,而且会争抢 I/O 资源;③ 作业调度会损失部分性能;④ 多元连接并行执行仍需要同步,某一个同步步骤的性能取决于最慢的作业分支(如图 4 中左右两个分支中较慢的),这无疑给调度带来更多的复杂性.因此,我们顺序地执行最佳连接查询执行计划.

6 实验验证

为测试本文提出的连接查询算法的有效性,并验证 I/O 代价模型的正确性,我们选用 13 台同构 PC 机,构建了基于 Hadoop HDFS 的云计算环境.节点为清华同方超翔 Z900 计算机,CPU 为 Inter Core i5-2300 2.80GHz,8GB 内存,1TB 硬盘,操作系统采用 CentOS 5.6,Linux 2.6.18 内核,Hadoop 采用 1.0.4 版,千兆网络环境.测试数据为 TPC-H 中的两个数据集 *LINEITEM* 和 *ORDERS*,分别作为主表(x)和从表(y),具体测试用例设计见表 2.

Table 2 SQL of joined query in test cases

表 2 查询用例中连接查询的 SQL 表达

```
SELECT y.o_custkey, x.l_extendedprice
INTO   NEWTABLE
FROM   LINEITEM AS x, ORDERS AS y
WHERE x.l_orderkey=y.o_orderkey
AND    x.l_shipmode="MODE"
AND    y.o_orderdate BETWEEN MIN AND MAX
```

测试用例对应的数据集设计见表 3.我们设定从表数据量不变,为 150 万条(较小的从表可保证所有连接算法均可用),主表数据量分别为 6 000 万(对应编号 1~6)和 60 000 万条(对应编号 7~12);连接率取决于连接属性的取值特征,我们通过调整 TPC-H 的数据生成规则得到 0.1(较小值),0.67(中间值)和 0.9(较大值)这 3 种连接率;选择率取决于测试用例中的查询条件,我们通过调整该条件获得两组选择率,β_x=0.14,β_y=0.45 和β_x=0.86,β_y= 0.91;投影率取决于 SELECT 语句,本测试用例中为常量.综上,共有 2×3×2=12 组实验.

Table 3 Data set oftest cases

表 3 测试用例数据集

连接率(α)	选择率(β)		
	β_x=0.14 β_y=0.45	β_x=0.86 β_y=0.91	
α_x=α_y=0.1	No.1, No.7	No.2, No.8	θ_x=60000000 或 600000000
α_x=α_y=0.67	No.3, No.9	No.4, No.10	θ_y=1500000
α_x=α_y=0.9	No.5, No.11	No.6, No.12	γ_x=0.152, γ_y=0.145

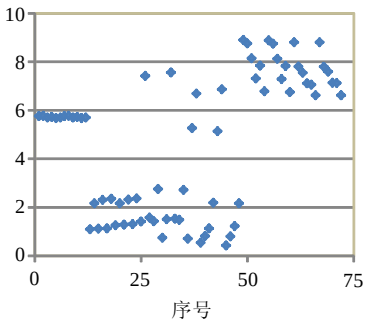
I/O 代价模型是一个计算模型,其中涉及较多参数.在实际应用中,可将这些参数分为两类:一类可以根据数

据特征和查询作业特征直接获取,如主从表数据量(ϕ_x,ϕ_y)、主从表记录条数(θ_x,θ_y)、投影率(γ)、连接属性大小(τ)等;另一类参数的确定则较为复杂,如连接率(α)、选择率(β)、连接条件和公式(7)中查询条件的相关度(ω_ε)、查询连接后的数据条数(θ)等,通常可采用以下 3 种方法获得:① 数据采样,Hadoop 自带了很多数据采样工具,如 *IntercalSmapler,RandomSampler,SplitSampler* 能够方便地完成数据采样工作,本实验采用这种方法,由于实验数据是随机生成的,随机采样得到的参数值近似于真实值;② 数据特征分析和估算,如数据分布已知可以估算查询的选择率,而连接选择率可以通过分析连接属性和选择属性的信息增益获得;③ 依赖特定应用场景和背景知识确定.

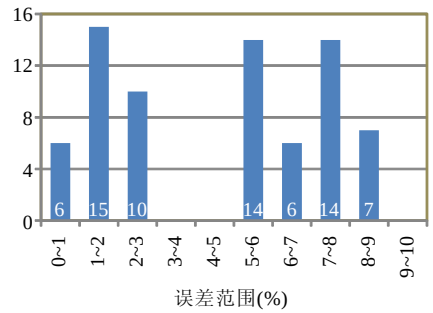
本文通过实验主要验证以下 3 个方面:首先是验证 I/O 代价模型和代价函数的准确性;其次,简单分析了 6 种二元连接算法的 I/O 代价特点;最后,验证了 I/O 代价和连接查询性能的关系.实验没有涉及多元连接的最优执行计划的 I/O 代价验证,因为在 I/O 代价函数正确的前提下,多元连接最优执行计划的 I/O 代价是可以计算得到的并从推导上保证其正确性.

6.1 I/O代价模型验证

图 5 给出了 6 种算法通过 I/O 代价函数计算出的理论值与实际 I/O 值(通过 MapReduce 框架获取)的误差分析情况,误差计算方法为 $\frac{|\text{理论值}-\text{测量值}|}{\text{测量值}}\times 100\%$.由图 5(a)可以看出,72 组数据的误差均不超过 9%,误差较均匀地分布在 0~9%的范围内.图 5(b)、图 5(c)为误差在各个区间的频数直方图和统计数据.可以看出,多组实验的 I/O 代价理论值与实际值均近似相等,平均误差 4.71%,数据略微逆向倾斜,较多的值集中在较大的值域,误差在可接受的范围内.此外,我们对 6 种算法按照 I/O 代价排序,理论值和实际值得出的排序结果一致.综上所述,我们提出的 I/O 代价模型是准确的,由它计算出的理论值能够充分代表各个算法的实际 I/O 代价.



(a) 误差散点图



(b) 误差直方图

个数	72
均值	4.71%
标准差	2.87%
最小值	0.43%
最大值	8.9%
一分位数	1.52%
中位数	5.71%
三分位数	7.18%

(c) 误差统计

Fig.5 Analysis of theoretic and actual I/O costs of six join algorithms

图 5 6 种算法 I/O 代价理论值与实际值的误差(%)分析

6.2 算法I/O分析

图 6 给出了 12 个实验组(不同数据量和参数,图序号均为实验组号,横轴均为算法,纵轴均为 I/O 代价(GB)测量值)中 6 种算法的 I/O 代价.由图 6 可以看出:

(1) 总体上,M 算法的 I/O 代价最高.这是因为 M 算法中,从表被反复分发到各个节点,这将增加算法的 I/O 代价;

(2) M^2 算法的 I/O 代价较低.这是因为与 M 算法相比,M^2 分发的是从表经选择投影后的数据,数据量远小于从表本身;而与其他算法相比,M^2 算法对主表和从表的读取次数较少,且选择与连接操作同时进行,网络 I/O 代价也少;

(3) 同样是 Map 端 Join,M^3R 的 I/O 代价高于 M^2.这是因为 M^3R 算法中主表读了两遍,而 M^2 只读取一遍,

由此造成的 I/O 代价远大于经连接表过滤后减少的网络 I/O 代价;

(4) 同样是 Reduce 端 Join,R^2M^2 的 I/O 代价高于 RM^2.其原因同样是因为读取主表的 I/O 代价远大于经属性数据集过滤后减少的网络 I/O 代价;

(5) 6 种算法在同一数据集上的 I/O 代价主要取决于算法本身.

此外,若横向比较每种算法在不同数据集上的 I/O 代价可以发现:两表数据量大小是决定 I/O 代价的主要因素,7~12 组实验的 I/O 代价是 1~6 组实验的近 10 倍;此外,I/O 也受连接率、选择率等参数的影响,他们之间的关系可以从 I/O 代价计算公式推出.

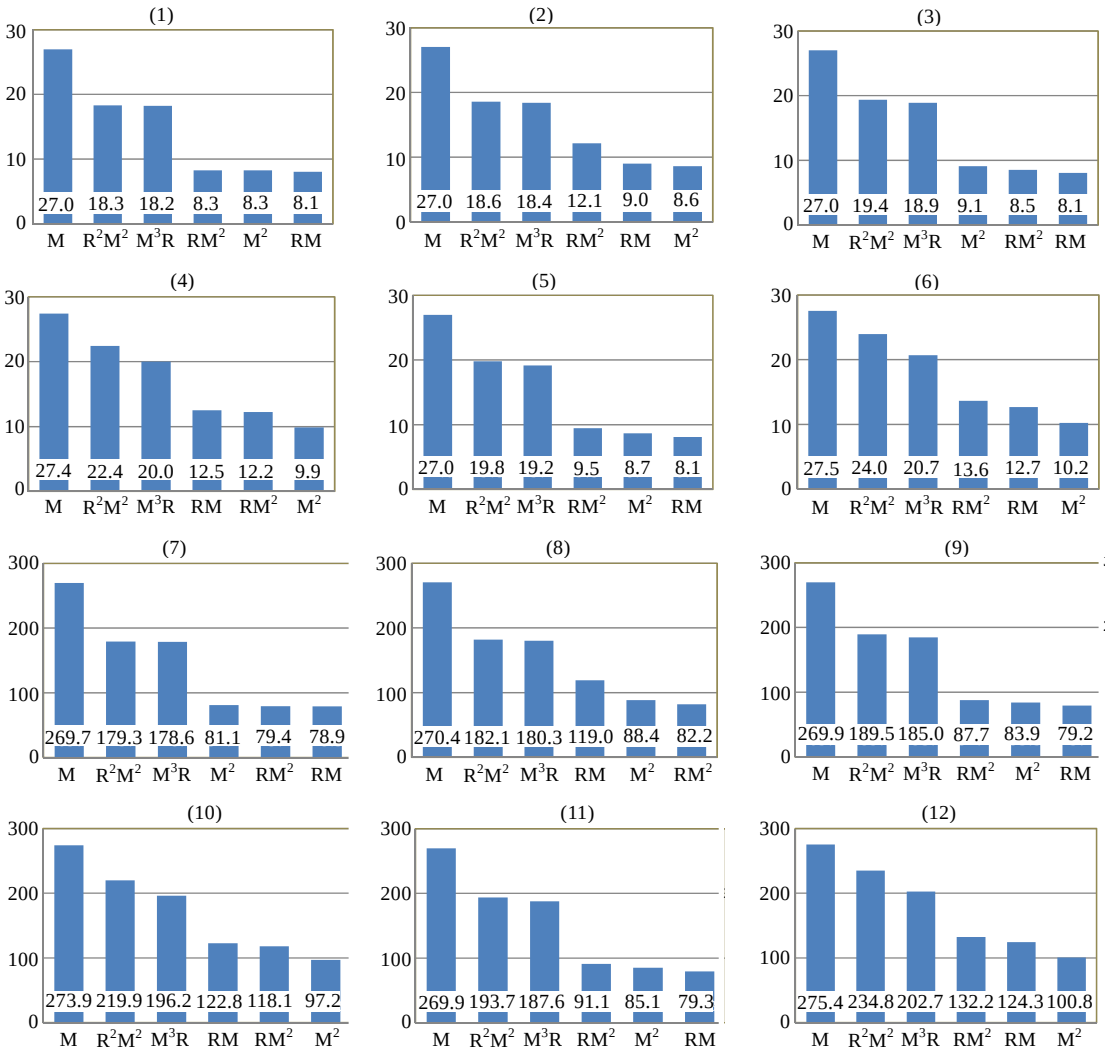


Fig.6 Actual I/O costs of six join algorithms on 12 data sets

图 6 6 种算法在 12 个数据集上的实际 I/O 代价

本文设计的 12 组实验数据集未能穷举所有情况,因此并非每种算法在 I/O 代价上的优势都能得以体现.例如:两表的连接率极低时,通过半连接获取连接属性数据集从而对两表先进行过滤的 M^3R^2 和 R^3M^3 算法将会凸显优势;而在数据仓库中,当从表为维表、主表为事实表时(此时,连接属性一般为从表主键),从表的连接属性集包含所有连接属性值,此时半连接失效,只提取从表连接属性的 RM^2 算法将会凸显优势.

6.3 算法I/O与执行时间的关系

本研究认为,连接查询的性能取决于 I/O 代价.连接查询的运算简单,I/O 代价明显高于运算代价,这一点在分布式环境下尤为明显,因此,选择 I/O 代价较小的算法执行将会提高查询性能.为验证这一结论,我们以查询的 I/O 代价为横轴,对应的执行时间为纵轴,对 72 组连接查询做散点图分析,如图 7 所示.由图 7 可得出如下结论:① I/O 代价与执行时间成线性正相关,说明 I/O 代价是影响执行时间的主要因素;② *A* 线附近的 63 个点可以回归到一条斜线上,说明 I/O 和算法执行时间之间的关系与算法无关,由于每个节点的 I/O 处理速度一致,且这些算法的并行性相同,因此这些点都在 *A* 斜线附近,而 *A* 斜线的斜率则表示整个集群的 I/O 吞吐量;③ *B* 线附近的 9 个点均为 MR 算法的实验结果,*B* 线的斜率小于 *A* 线,说明 MR 算法的 I/O 处理速度大于其他算法,这是因为 MR 算法的并行性高于其他算法,在 MR 算法中,从表虽需派发多次,但由此产生的 I/O 均是并行执行的.由此可知:当研究算法的 I/O 代价与执行时间的关系时,应该充分考虑并行度的影响.优化连接查询可以通过减少 I/O 操作或提高 I/O 操作的并行度来实现.

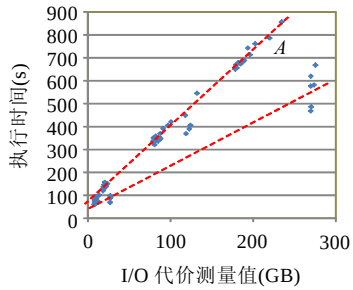


Fig.7 Relationship between I/O costs and time consumptions of six join algorithms on 12 data sets

图 7 6 种算法在 12 个数据集上的 I/O 代价与执行时间的关系图

7 结论和进一步工作

本文研究了基于 MapReduce 的连接查询算法的 I/O 代价模型.首先基于 MapReduce 连接现有的两种主流算法 Map-Join 和 Reduce-Join,定义 MapReduce 连接查询和其关键算法步骤;接着提出基于 MapReduce 的连接查询的 I/O 代价模型;随后对现有 Map-Join 和 Reduce-Join 加以整理和扩展,给出 6 种二元连接算法的形式化表达,并推导它们的 I/O 代价函数;随后将 I/O 代价模型和函数扩展到星型连接、链式连接等多元连接上,提出一种最优执行计划的选择算法.实验结果表明:6 种算法能够有效完成查询任务,提出的 I/O 代价模型也能够在可接受的误差范围内准确地估算各个算法的 I/O 代价,且 I/O 代价能够准确地反映算法的性能优劣.除上述研究成果外,本研究还得出以下结论:基于 MapReduce 的连接查询的性能取决于实现算法的 I/O 代价,两者成正比关系,其 I/O 代价主要受数据量、连接率、查询率、投影率等的影响,它们之间的关系可以通过 I/O 代价函数量化地表达,选择 I/O 代价较小的执行计划将会提高连接查询性能;此外,研究算法的 I/O 代价与执行时间的关系时,应该充分考虑并行度的影响,相同 I/O 代价的前提下,并行性高的算法性能更好.

基于 MapReduce 的连接查询是海量数据分析中常用的数据操作,本研究提出的 I/O 代价模型能够清晰定义 I/O 代价与其影响因素之间的关系,能够提前准确预测算法 I/O 代价,从而为算法选择、执行计划的选择和 I/O 代价优化提供理论依据,对分析 MapReduce 数据密集型任务的 I/O 代价起指导作用,有助于基于 MapReduce 的数据管理系统和分析系统的研发.本研究还处于初步阶段,首先,定义 I/O 代价模型的目的是为了更好地优化 I/O,因此我们将依据本文得出的结论进一步研究 I/O 代价优化方法.此外,除 I/O 代价以外,算法的并行度对数据密集型计算的性能影响也很大,而并行度又受数据倾斜等因素的影响,在性能优化时应该充分考虑并行度因素.

References:

[1] Big data: Science in the petabyte era. 2014. http://www.nature.com/nature/journal/v455/n7209/edsumm/e080904-01.html

[2] Directorate for Computer & Information Science & Engineering. 2014. http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503324&org=IIS2014,2,18

[3] Ghemawat S, Gobioff H, Leung ST. The Google file system. In: Scott ML, Peterson LL, eds. Proc. of the 19th ACM Symp. on Operating Systems Principles. BoltonLanding: ACM Press, 2003. 29−43. [doi: 10.1145/945445.945450]

[4] Hadoop™ distributed file system. 2014. http://hadoop.apache.org/docs/stable1/hdfs_design.html

[5] Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. Communication of the ACM, 2008,51(1):107−113. [doi: 10.1145/1327452.1327492]

[6] Blanas S, Patel JM, Ercegovac V, Rao J, Shekita EJ, Tian YY. A comparison of join algorithms for log processing in MapReduce. In: Elmagarmid AK, Agrawal D, eds. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. Indianapolis: ACM Press, 2010. 975−986. [doi: 10.1145/1807167.1807273]

[7] Luo G. Efficient join in Hadoop. Technical Report, NC 27705, Durham: Duke University.

[8] Hadoop MapReduce. 2014. http://hadoop.apache.org/docs/stable1/mapred_tutorial.html

[9] Yang H, Dasdan A, Hsiao RL, Parker DS. Map-Reduce-Merge: Simplified relational data processing on large clusters. In: Chan CY, Ooi BC, Zhou AY, eds. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. Beijing: ACM Press, 2007. 1029−1040. [doi: 10.1145/1247480.1247602]

[10] Ranger C, Raghuraman R, Penmetsa A, Bradski G, Kozyrakis C. Evaluating Mapreduce for multi-core and multiprocessor systems. In: Proc. of the 13st Int'l Conf. on High-Performance Computer Architecture (HPCA-13 2007). Phoenix: IEEE Computer Society, 2007. 13−24. [doi: 10.1109/HPCA.2007.346181]

[11] Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig latin: A not-so-foreign language for dataprocessing. In: Tsong J, Wang L, eds. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. Vancouver: ACM Press, 2008. 1099−1110. [doi: 10.1145/1376616.1376726]

[12] Apache hive. 2014. http://hive.apache.org/

[13] Beyer KS, Ercegovac V, Gemulla R, Balmin A. Jaql: A scripting language for large scale semistructured data analysis. PVLDB, 2011,4(12):1272−1283.

[14] Wang LQ, Gao J, Wang TJ, Li HY. A cost aware adaptive multiple table joinevaluation in MapReduce. In: Proc. of the 9th Int'l Conf. on Fuzzy Systems and Knowledge Discovery. Chongqing: IEEE, 2012. 2437−2441. [doi: 10.1109/FSKD.2012.6233855]

[15] Luo G, Dong L. Adaptive join plan generation in Hadoop. Technical Report, NC 27705, Durham: Duke University.

[16] Afrati FN, Ullman JD. Optimizing joins in a Map-Reduce environment. In: Manolescu I, Spaccapietra S, Teubner J, Kitsuregawa M, eds. Proc. of the 13th Int'l Conf. on Extending DatabaseTechnology. Lausanne: ACM Press, 2010. 99−110. [doi: 10.1145/1739041.1739056]

[17] Lynden SJ, Tanimura Y, Kojima I, Matono A. Dynamic data redistribution for MapReduce joins. In: Lambrinoudakis C, Rizomiliotis P, Wlodarczyk TW, eds. Proc. of the 3rd IEEE Int'l Conf. on Cloud Computing Technology and Science (CloudCom 2011). Athens: IEEE, 2011. 717−723. [doi: 10.1109/CloudCom.2011.111]

[18] Zhu HT, Zhou MQ, Xia F, Zhou AY. Efficient star join for column-oriented data storein the MapReduce environment. In: Proc. of the 8th Web Information Systems and Applications Conf. Chongqing: IEEE, 2011. 13−18. [doi: 10.1109/WISA.2011.10]

[19] Zhou MQ, Zhang R, Zeng DD, Qian WN, Zhou AY. Join optimization in the MapReduce environment for column-wise data store. In: Proc. of the 6th Int'l Conf. on Semantics Knowledge and Grid. Beijing: IEEE, 2010. 97−104. [doi: 10.1109/SKG.2010.18]

[20] Lin YT, Agrawal D, Chen C, Ooi BC, Wu S. Llama: Leveraging columnar storage for scalable joinprocessing in the MapReduce framework. In: Sellis TK, Miller RJ, Kementsietsidis A, Velegrakis Y, eds. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. Athens: ACM Press, 2011. 961−972. [doi: 10.1145/1989323.1989424]

[21] Okcan A, Riedewald M. Processing theta-joins using MapReduce. In: Sellis TK, Miller RJ, Kementsietsidis A, Velegrakis Y, eds. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. Athens: ACM Press, 2011. 949−960. [doi: 10.1145/1989323.1989423]

[22] Mi CC, Chen Q, Liu TY. An efficient cross-match implementation based ondirected join algorithm in MapReduce. In: Proc. of the IEEE 4th Int'l Conf. on Utility and Cloud Computing. Melbourne: IEEE Computer Society, 2011. 41−48. [doi: 10.1109/UCC. 2011.16]

[23] Zhang XF, ChenL, WangM. Towards efficient join processing over largeRDF graph using MapReduce. In: Ailamaki A, BowersS, eds. Proc. of the 24th Int'l Conf. on Scientific and Statistical Database Management. Chania: Springer-Verlag, 2012. 250−259. [doi: 10.1007/978-3-642-31235-9_16]

[24] Han XX, Yang DH, Li JZ. Approximate join aggregate on massive data. Chinese Journal of Computers, 2010,33(10):1919−1933 (in Chinese with English abstract). [doi: 10.3724 / SP.J.1016.2010.01919]

[25] She R, Wang K, Fu AW, Xu YB. Computing join aggregates over private tables. In: Taniar D, Rusu LI, eds. Proc. of the Strategic Advancements in Utilizing Data Mining and Warehousing Technologies: New Concepts and Developments (IGI Global). 2010. 256−276.

[26] Taniar D, Jiang Y, Liu KH, Leung CHC. Parallel aggregate-join query processing. Informatica (Slovenia), 2002,26(3).

[27] 2014. http://www.enet.com.cn/article/2012/0401/A20120401990472.shtml

[28] Swami AN, Schiefer KB. On the estimation of join result sizes. In: Jarke M, Jr. Bubenko JA, Jeffery KG, eds. Proc. of the Advances in Database Technology (EDBT'94), 4th Int'l Conf. on Extending Database Technology. Cambridge: Springer-Verlag, 1994. 287−300. [doi: 10.1007/3-540-57818-8_58]

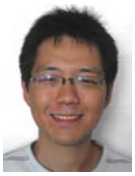
[29] Getoor L, Taskar B, Koller D. Selectivity estimation using probabilistic models. In: Mehrotra S, Sellis TK, eds. Proc. of the 2001 ACM SIGMOD Int'l Conf. on Management of Data. Santa Barbara, 2001. 461−472. [doi: 10.1145/375663.375727]

[30] Tzoumas K, Deshpande A, Jensen CS. Efficiently adapting graphical models for selectivity estimation. Journal of VLDB, 2013, 22(1):3−27. [doi: 10.1007/s00778-012-0293-7]

[31] Knuth DE. The Art of Computer Programming. 3rd ed., Addison-Wesley, 1973.

附中文参考文献:

[24] 韩希先,杨东华,李建中.海量数据上的近似连接聚集操作.计算机学报,2010,33(10):1919−1933. [doi: 10.3724/SP.J.1016.2010.01919]



宋杰(1980－),男,安徽淮北人,博士,副教授,CCF 高级会员,主要研究领域为高能效计算,海量数据计算,云计算.

李甜甜(1989－),女,博士生,CCF 学生会员,主要研究领域为海量数据计算.

朱志良(1962－),男,博士,教授,博士生导师,主要研究领域为 Web 服务,复杂网络.

鲍玉斌(1968－),男,博士,教授,CCF 高级会员,主要研究领域为数据仓库,图数据处理.

于戈(1962－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为数据库理论和技术,分布与并行系统.