密级：_____

浙江大学

硕 士 学 位 论 文

（工程硕士）

论文题目 基于超图和浓密树的大数据实时查询优化研究与实现

作者姓名 马骄阳

指导教师 陈 岭 副教授

陈根才 教授

学科(专业) 计算机技术

所在学院 计算机科学与技术学院

提交日期 2015.01.26

A Dissertation Submitted to Zhejiang University for the Degree of Master of Engineering

TITLE: Hypergraph and Bushy-Tree based Research and Implementation of Big Data Real-Time Query Optimization

Author: Ma Jiaoyang

Supervisor: Associate Professor Chen Ling

Professor Chen Gencai

Subject: Computer Technology

College: Computer Science and Technology

Submitted Date: 2015.01.26

基于超图和浓密树的大数据实时查询优化研究与实现

马骄阳　浙江大学

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 浙江大学 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：　　签字日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解 浙江大学 有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 浙江大学 可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名：　　导师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

摘要

Impala 是 Cloudera 公司推出的大数据实时查询系统，其利用分布式技术，使用 Hadoop 作为存储，通过 Hive 的元数据表获取统计信息，实现大数据的高效查询。Impala 的最新版本提供了一些查询优化技术，但目前只支持左深树形式的查询计划，同时由于目前基于 McCHyp（MinCutConservative Hypergraph）算法的查询优化存在搜索空间大、优化时间较长的问题，本文提出了基于浓密树和改进的 McCHyp 算法的 Impala 查询优化方法。首先，修改 Impala 使其支持浓密树形式的查询计划；接着，使用剪枝策略对 McCHyp 算法进行改进，减少查询优化的时间，同时对剪枝策略的完整性和正确性进行了说明；然后提出一种综合考虑了磁盘 I/O、网络传输和右表的大小的代价模型，并将改进的 McCHyp 算法集成到 Impala 中，根据用户的 SQL 语句生成较优的查询计划。最后在 Impala 2.0 系统上实现了本文提出的查询优化方法并在 TPC-DS 数据集上进行了对比实验，结果表明，改进的 McCHyp 算法与 McCHyp 算法对连接超图的优化结果一致，且前者的运行时间减少了 43.82%~62.55%。使用改进的 McCHyp 算法及新的代价模型对查询语句优化后，查询响应时间较原始的 Impala 系统减少了 34.66%。同时，支持浓密树形式查询计划的 Impala 系统具有更好的可扩展性，通过增加集群节点可以减少查询响应时间。

关键词： 查询优化，Impala，代价模型，浓密树，查询计划

Abstract

Impala is a big data real-time query system, which is launched by the Cloudera Inc. It makes use of the technology of distributed system to make big-data query efficiently. The system uses Hadoop as storage and Hive's metadata table as a tool for statistics. Although the up-to-date version of Impala has some techniques for query optimization, it only supports query plans in left-deep tree form. On the other hand, McCHyp (MinCutConservative Hypergraph) based query optimization algorithm has problems with large search space and long optimization time. As the big-data real-time query system Impala has problem with query optimization, we proposed a bushy-tree and Improved-McCHyp algorithm based Impala query optimization method. The method firstly modified Impala to support query plans in bushy-tree form. Then we improved McCHyp algorithm with pruning strategy to reduce query optimization time, and explained the integrity and correctness. After that we proposed a new cost model which considers disk I/O, network transfer and the size of right table, and integrated Improved-McCHyp algorithm into Impala to generate better query plans with user's SQL statement. Finally, the query optimization method is implemented in Impala 2.0 and evaluated by TPC-DS, experimental results show that Improved-McCHyp algorithm had the same result with McCHyp algorithm, and the running time of the former decreased by 43.82%~62.55%. The processing time of the query, which was optimized by Improved-McCHyp algorithm and the new cost model, decreased by34.66%. Also, Impala system which support query plans in bushy tree form has good scalability, which can decrease the query response time by adding more nodes.

Keywords： query optimization, Impala, cost model, bushy tree, query plan

目录

图目录

表目录

第1章 绪论

1.1 研究背景及意义

数据库查询优化的研究已有很久的历史，从几十年前的单机数据库到现在的分布式大数据平台，衍生出了许多查询优化技术。查询优化中比较重要的研究之一是对多表连接查询的优化。根据连接语句的特性，一条查询语句可以有多种执行顺序：其可得到同样的查询结果，但是耗费的时间可能相差很大。因此多表连接查询优化的目标就是要找到一个较好的连接执行顺序。从查询语句中选择最优的执行策略是一个 NP 问题[1]，通常会选取近似最优的解，或避免代价较高的解。

查询优化主要由搜索空间（search space）、搜索策略（search strategy）和代价模型（cost model）三部分组成。

搜索空间中存在着许多等价的执行计划，其中“等价”指的是其执行得到的结果都相同，但是执行的顺序不一样。这些等价执行计划通常是使用转换规则生成的。如果搜索空间较大，会导致优化的时间较长，甚至长于实际查询的时间，因此需要对搜索空间进行限制。通常从 2 个方面着手：一方面，可以采用启发式方法，如优先执行选择和投影操作、避免笛卡儿积连接操作等；另一方面，可以通过查询计划树的形状来限制搜索空间。最初的优化对象通常是单机的数据库，因此查询优化主要集中在左深树空间：其更适用于单机的系统，且搜索空间小。随着分布式系统的出现，优化的重点转移到浓密树，因为某个节点的 2 个非叶子节点通常可以并行执行，使系统的并行度提高。但浓密树的搜索空间太大，优化时间较长，因此又有人提出了剪枝策略，减少决策时需考虑的候选方案的数量。

搜索策略主要有三大类：一类是基于备忘录的自顶向下算法，一类是基于动态规划的自底向上算法，还有一类是启发式算法。前 2 类算法可以找到最优解，而第三种算法只能找到次优解（近似最优解），但通常可以大幅减少查询优化的时间。

代价模型通常是带权重的磁盘 I/O、CPU 及网络开销的组合。对于单机数据

库，查询执行中间不存在网路传输的过程，因此主要关注磁盘 I/O 和 CPU 的开销。对于分布式数据库，网络拓扑的情况极大的影响着这些参数的权重。早期的局域网，由于网络带宽较小，主要开销在网络传输。随着网络硬件设备的发展，传输速率从几十 Kbps 提高到了现在的几百 Mbps 甚至几十 Gbps，而 CPU 浮点运算能力和磁盘 I/O 的提升没有这么大，因此在数据量一定的情况下，网络开销对查询的影响越来越小。对于广域网，网络带宽的提升不是很明显，因此需要同时考虑磁盘 I/O、CPU 和网络开销。

按优化的时机进行划分，查询优化又可分为三类：动态查询优化、静态查询优化和混合查询优化。其中动态查询优化只在查询执行的过程中进行优化，通常使用分离（detachment）和置换（substitution）2 种技术。静态查询优化是在查询执行之前进行优化，生成最优（或次优）的查询执行计划。而混合查询优化则是将动态和静态 2 种优化方法结合在一起，以解决某些应用场景中遇到的问题。

目前研究关注最多的是静态查询优化，其方法主要是改变连接的顺序。传统的查询优化通常关注于内连接（自然连接）的优化，随后又发展出对半连接、外连接等其他连接的优化。基于简单图的优化方法可以很好的解决内连接的优化问题，但无法对其他连接进行优化，因此又发展出了基于超图的查询优化方法。这些优化方法有些已应用到实际的产品中，取得了较好的效果。

在业界，随着大数据时代的到来，企业每天需要处理海量的数据，而传统的查询处理已无法满足这种需求。Google 的 GFS[2]、MapReduce[3]及受其启发的开源版本 Apache Hadoop[4]、Apache Hive[5]，有效提高了系统的数据存储和计算能力，降低了运维的成本。

虽然上述系统可以容纳、处理更多的数据，但其查询处理时间较长，无法实时（准实时）的返回结果，不适用于一些应用场景。对此，Google 在 2010 年公开了一篇论文[6]，介绍了可扩展、交互式的实时查询系统 Dremel。Google 的 Dremel 作为 MapReduce 的有力补充，可以处理 PB 级的数据，并将查询的时间缩短到秒级。Cloudera Impala[7]便是受其影响而开发的开源大数据实时查询平台。Cloudera Impala 可以直接为存储在 HDFS 或 HBase 中的数据提供快速、交互式的类 SQL

查询，查询时间比基于 MapReduce 的 Apache Hive 提高了 3~90 倍[8]。Impala 1.0 版本中的查询优化主要提供了 2 种不同的 Join 方法：Broadcast Join 和 Partitioned Join。这 2 种方法主要用于决定 Join 执行的节点和数据传输的方式，并未对 Join 顺序进行优化。同时，当采用 Broadcast Join 方法时，Impala 会将参与连接操作的右表的全部数据发送到左表所在的节点。由于未对连接顺序进行优化，执行用户的查询请求时容易出现查询缓慢，甚至内存溢出等问题。现有的查询优化方法通常关注如何使查询时间最短，而很少关注中间结果的大小，无法满足 Impala 系统的要求。最新 Impala 2.0 系统对查询语句进行了简单的优化，并且增加了 Spill to Disk 功能[9]以支持大于内存大小的数据的连接查询，但其生成的查询计划是左深树形式的，仍有进一步优化的空间。

我们将基于已有的工作，从搜索空间、搜索策略和代价模型等方面对查询优化算法进行研究，并在大数据实时查询平台 Impala 上进行实现，以提高 Impala 的查询效率及对多表连接查询的处理能力。

1.2 相关研究工作

查询优化主要分为搜索空间、搜索策略和代价模型三部分。本文工作对这三方面均有所涉及，下面分别介绍这几方面相关的研究工作。

1.2.1 搜索空间

搜索空间可以通过查询树（查询图）来进行表示，可在 2 个维度进行划分。一个维度是树的形状：左深树（右深树）和浓密树；另一个维度是节点间的关系：笛卡儿积和非笛卡儿积。

1. 2. 1. 1 左深树和浓密树

Steinbrunn 等人[10]对左深树和浓密树的搜索空间进行分析，得出 n 个基本关系的左深树有n!种可能，而浓密树有$\binom{2(n-1)}{n-1}(n-1)!$种可能。该结论为后续研究提供了理论基础。在此基础上，Ono 等人[11]对搜索空间进行优化，得出在线性连接图下，左深树的搜索空间为$(\mathrm{n}-1)^2$，而浓密树的搜索空间为$\frac{n^3-n}{6}$。

20 世纪 90 年代之前，研究人员更多关注于基于左深树的查询优化[12][13][14]，

这主要是受当时计算机软硬件的制约，多数数据库都是单机的，左深树的查询计划比较适合。而随着科学技术的发展，在分布式系统出现后研究的重点转向了基于浓密树的查询优化[15][16][17]，因为浓密树的 2 棵子树可以在不同节点并行执行，提高了查询的效率。

将左深树空间改为浓密树空间已经有过许多尝试。Katsoulis[18]在 Hadoop 平台上实现了并行哈希连接算法。该算法可以有效的提高 MapReduce 查询的执行效率，但其未考虑连接的顺序问题，还有进一步优化的空间。Wu 等人[19]提出 AQUA 2 阶段查询优化方法，可在基于 MapReduce 的系统上执行，搜索空间为浓密树。其通过将搜索空间从左深树扩展到浓密树，大幅提高了系统的并行执行能力，减少了查询响应时间。Apache Derby 在其中一个 Issue[20]中提到了对浓密树的查询优化支持：有用户反应当连接表大于 5 张时，Derby 的效率会非常差，因此希望能将查询优化的搜索空间从左深树改为浓密树，但至今仍未实现。Microsoft SQL Server 基于瀑布框架[21]对查询进行了优化，已经可以支持浓密树的查询优化方案。

1.2.1.2 笛卡尔积

通常，由于笛卡儿积的运算会导致中间结果庞大，多数系统在优化时不考虑笛卡儿积。Ono 等人[11]设计了称为 Starburst 的连接生成器，可让用户选择是否允许连接关系是笛卡儿积，通过禁止笛卡儿积，部分查询执行效率可以得到显著的提高。System R[12]和 INGRES[22]通过将笛卡儿积推延到最后进行计算来降低执行时的代价。

Vance 等人[23]认为考虑整个搜索空间更加合理，提出了笛卡儿积查询优化算法，并通过实验证明当关系个数小于 15 时，考虑笛卡儿积的查询优化算法可以得到更好的结果。该算法使用一个简化的代价模型，较容易实现，但当连接表较多时，优化效果会比较差。

1.2.2 搜索策略

搜索策略一般可分为 2 类，一类是传统的精确算法，可获得最优解，主要包括自顶向下算法和自底向上算法 2 种；另一类是可在较短时间获得近似最优解的启发式算法。启发式算法可分为简单启发式算法（如爬山法、贪心法等）和元启

发式算法（如蚁群算法、遗传算法、微粒群算法等）[24]。

1.2.2.1 自顶向下算法

DeHaan 等人[16]提出关于连接图的最优的自顶向下多表连接生成算法 MinCutLazy，并通过与分支定界算法（branch-and-bound）的结合，提高了查询优化的效率。其关注于不含笛卡尔积的浓密树搜索空间，首次使用备忘录算法进行自顶向下的查询优化，可在 CPU 利用率与内存占用之间灵活均衡。

Fender 等人[17]对 MinCutLazy 算法进行了复杂度分析，得出其对团状查询的复杂度为 $O(n^2)$，并提出了自己的算法 MinCutBranch，将团状查询的复杂度降低为 O(1)。Fender 等人[25]在此基础上对算法进一步优化，提出了 MinCutLazyImp 算法，同时基于最优划分的概念，提出了 MinCutAGat 算法。该算法省去了一些不必要的步骤，使优化的效率得到了提高。

Fender 等人[26]在 MinCutLazy 算法[16]的基础上，改进了剪枝算法，并提出了一个新的自顶向下算法 MinCutConservative，比 MinCutLazy 和 MinCutBranch 更容易实现，且性能有少量提升。但该算法只支持二元谓词、只支持内连接，在实际应用中有很大局限。

Fender 等人[27]基于 DPhyp（Dynamic Programming Hypergraph）算法[28]的想法，设计了基于超图的自顶向下算法 McCHyp 以支持多元谓词，同时还可以对半连接、外连接等查询进行优化。该算法是目前自顶向下查询优化算法中效率最高的算法，适用于支持浓密树搜索空间的分布式数据库系统。

1.2.2.2 自底向上算法

Moerkotte 等人[29]在 DPsize[11]和 DPsub[30]的基础上，提出可达到 Ono 等人[11]提出的复杂度下界的自底向上动态规划算法 DPccp。该算法效率比 DPsize 提高很多，但是仍存在不足。其不支持超图，无法处理外连接、反连接（anti-join）等不同顺序可能会得到不同结果的连接操作，在应用上受到了一定的限制。

Moerkette 等人[28]基于 DPccp 算法，设计了新算法 DPhyp。其采用超图的思想实现，可以处理复杂的查询谓词，将优化时间减少了十倍。Neumann[31]提出一种查询简化框架，对于多连接查询，首先使用启发式贪婪算法将连接图进行简化，然后再使用 DPccp 算法进行查询优化，同时通过实验证明该方法比之前的方法有

更好的表现。

周强等人[43]提出了改进的 DPhyp 算法，该算法将搜索空间减小到左深树空间，并集成到 Impala 中，使 Impala 的查询响应时间平均减少 67%~80%。但由于使用左深树的搜索空间，无法利用分布式系统并行性的特点，查询优化的效果受到了一定的限制。

近年来一些新的应用，如对象关系映射（ORM）工具，给查询优化带来了新的挑战：动态生成的查询语句通常比较复杂，但是执行代价较低，且需要在运行时进行优化。Nica[32]提出 orderd-Par 技术，并与 DPhyp 算法结合，提出 orderd-DPhyp 算法。该算法运行效率优于前者，且已集成在 SQL Anywhere 16.0[33]产品中。

1.2.2.3 启发式算法

赵鹏等人[34]将伪随机概率转移规则引入最大最小蚁群系统（Max-Min Ant System，MMAS）中，并结合局部搜索策略，提高了算法的收敛速度。Dökeroğlu 等人[35]将动态规划算法和蚁群算法相结合，设计出多项式时间的近似最优解查询优化算法。Golshanara 等人[36]首次将多种群蚁群算法运用在查询优化上，并通过实验证明其优化相比基于遗传算法的查询优化节省大约 80%的时间。

Sevinç等人[37]提出了一种基于新的遗传算法（NGA）的查询优化，并通过实验证明其比之前的基于遗传算法的查询优化性能提高了 50%。该算法很好的将遗传变异的思想融入到查询优化中，取得了不错的效果。

Dokeroglu 等人[38]首次提出基于微粒群算法[39]的查询优化方法，并与基于动态规划算法及遗传算法的查询优化比较。该算法可以有效的找到近似最优解，且在性能上比现有的动态规划算法和遗传算法有小幅提高。

1.2.3 代价模型

查询优化的代价模型通常会根据统计信息、操作符、原始数据等特征来计算运算的代价，而代价通常以执行时间的方式表示。一个分布式执行策略的代价可以表示为总时间或响应时间。总时间等于所有组件花费的时间之和，而响应时间是从查询开始到结束的时间。

Loham 等人[40]提出计算总时间的通用公式（1.1）：

$$T_{total} = T_{CPU} \times \#insts + T_{I/O} \times \#I/Os + T_{MSG} \times \#msgs + T_{TR} \times \#bytes \quad \text{公式（1.1）}$$

该公式考虑了 CPU、磁盘 I/O、数据传输时间和网络通信时间，基本涵盖了查询涉及的组件。但该公式有 8 个变量，计算较复杂，且在某些系统中一些参数无法获取，或计算代价较大，因此在实际使用时通常会忽略影响较小的参数，或将其设为一个常量。

若将响应时间作为优化器的目标函数，则还需考虑系统的并行性（本地并行处理及并行通信）。Khoshafian 等人[41]提出计算响应时间的通用公式（1.2）：

$$T_{response} = T_{CPU} \times seq\_\#insts + T_{I/O} \times seq\_\#I/Os + T_{MSG} \times seq\_\#I/Os + T_{TR} \times \#seq\_bytes \quad \text{公式（1.2）}$$

其中 *seq_#x* 表示执行查询时 *x* 必须串行执行的最大值。该公式同样过于复杂，在实际应用中应根据情况对参数进行取舍。

Ganguly 等人[42]提出了一种适用于并行查询优化的代价模型，如公式（1.3）所示。

$$C_{ca}(T) = \begin{cases} 0 \\ |T| + C_{ca}(T_1) + C_{ca}(T_2) \end{cases} \quad \text{公式（1.3）}$$

其中 T 是一棵连接树，T_1 和 T_2 是 T 的子树。当 T 是一个叶节点（单一表）时，代价为 0，否则 T 应满足$T = T_1 \bowtie T_2$。该代价函数主要关注于连接的势（cardinality），同时也考虑了子树连接操作的代价。但该函数并不适用于哈希连接操作，由公式可以推出$T_1 \bowtie T_2$的代价与$T_2 \bowtie T_1$的代价是相同的，但事实上并非如此，哈希连接操作的左右 2 表并不是对等的。

周强等人[43]在对 Impala 进行查询优化时提出了一个新的代价模型，见公式（1.4）。

$$C_{opt}(i+1) = \max_{j \in [1,n]} |T_i \bowtie R_j| \quad \text{公式（1.4）}$$

其中$i \in [1, n-1]$，T_i表示已经计算出代价的查询树，R_j表示一个单一表，其不在 T_i 中，且与 T_i 之间有连接谓词。该代价模型主要考虑数据传输的代价和参与

连接的左右 2 表的大小。使用此代价模型的优化结果通常是按表中数据条数从大到小排列。对于左深树形式的查询计划树，使用该代价模型有较好的优化效果，但由于未考虑中间结果的大小及分布式系统并行执行的特点，并不适用于浓密树形式的查询计划，代价估计会有偏差。

1.2.4 目前研究存在的问题

本文将对查询优化进行研究，并在 Impala 2.0 系统上实现。目前 Impala 的查询实现存在以下问题：

1. Impala 的查询计划生成、执行方式是基于左深树的，内部使用链表来保存表关系，不支持浓密树的查询计划，可优化空间小，大部分优化算法无法直接应用到该系统上。同时系统并行性差，多表连接只能按顺序进行；
2. Impala 1.0 系统没有对查询进行很好的优化，完全根据用户查询语句的书写顺序构建查询计划，且连接操作使用基于内存的哈希连接，当连接表中右表较大时，很容易发生内存溢出，导致查询失败。Impala 2.0 系统未考虑中间结果的大小，当连接表右表较大时，查询速度缓慢；
3. Impala 只提供通用的查询代价模型，未考虑 Impala 系统的查询执行特点。

基于 Impala 系统的特点，当前的自顶向下查询优化算法存在以下问题：

1. 基于简单图的自顶向下查询优化算法只能支持自然连接，不支持外连接、半连接等其他的连接操作。基于超图的自顶向下查询优化算法虽然可以对多种连接进行优化，但该算法论文中未提及代价估计等具体细节；
2. 自顶向下算法在计算时可通过剪枝策略来减少计算量，加快获得最优解的速度，但目前还没有基于超图的此类算法；
3. 现有自顶向下查询优化算法通常关注于如何获得最短查询时间，而不考虑中间结果的大小。在大数据环境下，Impala 会将超过内存大小的数据放在硬盘上进行读写，导致查询缓慢。

1.3 研究目标及内容

1.3.1 研究目标

基于目前研究存在的问题，本课题的研究目标主要有以下几个：

1. Impala 系统的研究和改进；
2. 提出改进的基于超图和浓密树的大数据实时查询优化方法；
3. 在 Impala 上实现该优化方法；
4. 设计实验对 Impala 系统改进前后的性能进行对比。

1.3.2 研究内容

本课题研究的主要内容如下：

1. Impala 系统的研究和改进
   a) 对 Impala 系统现有查询计划生成、执行方式进行研究，分析相关数据结构和执行流程；
   b) 针对 Impala 系统只支持左深树查询执行的特点，拟研究 Impala 环境下浓密树形式查询计划的构建、执行方法，建立合理的数据结构来保存查询计划。
2. 提出改进的基于超图和浓密树的大数据实时查询优化方法
   a) 针对简单图只支持内连接的问题，拟研究基于超图和浓密树的查询优化算法；
   b) 针对基于浓密树的查询优化时间较长的问题，探索在算法上集成剪枝策略的方法。
3. 在 Impala 上实现该优化方法
   a) 探索在 Impala 系统上实现该算法的方法；
   b) Impala 系统构建在 Hadoop 之上，共享 Hive 的元数据，针对 Impala 系统的特点，拟研究查询代价估算方法，建立查询代价模型。
4. 设计实验对 Impala 系统改进前后的性能进行对比
   a) 实验并分析剪枝策略对查询优化时间的影响；
   b) 实验并分析不同代价模型对查询性能的影响；

c） 实验并分析基于超图和浓密树的大数据实时查询优化方法对查询效率的影响；

d） 实验并分析基于超图和浓密树的大数据实时查询优化方法对集群可扩展性的影响。

1.4 本文结构组织

本文共分 7 章。第 1 章为绪论，主要介绍了大数据实时查询系统的发展过程及研究意义，从搜索空间、搜索策略和代价模型三方面介绍了查询优化的方法及相关研究工作，并针对大数据实时查询系统对当前方法存在的不足进行了分析，最后介绍了本课题的研究目标和内容，以及本文的组织结构。

第 2 章为 Impala 大数据实时查询系统，首先对 Impala 大数据实时查询系统进行介绍，接着从语法解析、查询计划生成、执行计划生成、查询和汇总几部分介绍 Impala 1.0 系统的查询过程，最后介绍 Impala 2.0 系统对查询过程的改进。

第 3 章为代价估计，首先对代价估计进行了概述，接着提出了适用于 Impala 大数据实时查询系统的代价模型，最后从基于表的代价估计和基于列的代价估计两方面对代价估计方法进行了介绍和分析。

第 4 章为基于超图和浓密树的大数据实时查询优化，首先给出相关概念和基本定义，接着对超图的建模方式进行了介绍，然后提出了改进的 McCHyp 算法，最后对剪枝策略的正确性和完整性进行了说明。

第 5 章为系统实现，首先给出了系统的总体框架，接着介绍了查询计划形式的修改方法，最后将改进的 McCHyp 算法在 Impala 系统中进行了集成。

第 6 章为实验评估，首先介绍了实验环境和实验设置，接着给出实验使用的数据集，并从优化算法、代价模型、查询性能和可扩展性几方面进行了实验，最后展示了实验结果并对其进行了详细的分析。

第 7 章为总结与展望，对本文所做的工作进行了系统的总结，并对大数据实时查询优化的发展和进一步工作进行了展望。

1.5 本章小结

本章首先介绍了大数据实时查询优化的研究背景及意义；接着详细介绍了相关的研究工作，主要包括搜索空间、搜索策略和代价模型三方面；然后分析了目前研究存在的问题并提出了本文的研究目标和内容；最后给出了全文的结构组织。

第2章 Impala 大数据实时查询系统

2.1 Impala 介绍

Impala 是 Cloudera 公司的开源大数据实时查询系统，其基于具有灵活性和可扩展性的 Hadoop 平台，提供支持 SQL 和多用户的大数据分析与查询服务。Impala 是开源商业智能和数据发现领域的业界标准，已在国外多家领头商业智能和分析应用领域的企业中使用，并在 Hadoop 生态系统中得到了广泛的支持[7]。

Impala 主要处理关系型数据，在 Impala 出现之前，当传统关系型数据库遇到容量瓶颈时，通常只能通过扩展系统来保证查询的性能，而传统关系型数据库的扩展在技术和成本上都是较高的。若之前使用的是 Hadoop 系统，为了能够获得实时查询的性能，需要将数据复制到一个速度较快的关系型数据库中，因此需要花费额外的代价来处理数据的复制和同步，同时数据的分析和查询将受限于目标关系型数据库。Impala 的出现，使需要进行数据实时分析和查询的用户多了一个选择，由于 Impala 是 Hadoop 生态系统的一部分，其具有 Hadoop 框架的所有特性，包括灵活性、可扩展性、高性能、易用性以及高性价比等，在学术界和工业界有较大的影响力。

2.1.1 Impala 1.0 系统

Impala 1.0 发布于 2013 年 4 月[44]，是第一个正式版本，自 2012 年 10 月 0.1 beta 版本发布后，经历了半年的时间，修复了数十个 bug，并增加多个功能使 Impala 系统更加易用，此时已支持一般的 SQL 操作，但还不能处理一些复杂查询（如嵌套查询）。

Impala 1.0 支持多种连接操作，包括左连接、右连接、半连接、全连接和外连接等，但并不支持笛卡尔积（既要求所有参与连接的表之间都有直接或间接的连接谓词约束）[45]。进行连接操作时，参与连接的右表数据会完全放在内存中，当内存不足时，会导致内存溢出，查询失败。

2.1.2 Impala 2.0 系统

Impala 2.0 发布于 2014 年 10 月[44]，经过 2 年的发展，Impala 更加成熟和稳定，其支持了嵌套查询以及笛卡尔积查询[45]。同时，通过提供 Spill to Disk 功能，可以支持大于内存容量的数据查询等操作[46]。

2.2 Impala 1.0 系统查询过程

Impala 1.0 系统的查询处理流程如图 2.1 所示。用户提交 SQL 查询语句，查询语句经过解析后发送给 Planner；Planner 根据表的元数据信息，生成查询计划；Coordinator 根据查询计划生成执行计划，并分配执行的节点，将执行计划分发到对应的执行节点上；各个执行节点上的 Executor 进行具体的查询操作，并将结果返回给负责分配任务的那个节点；该节点上的 Coordinator 将结果进行汇总并进一步处理后，由 Planner 将最终的查询结果返回给用户。

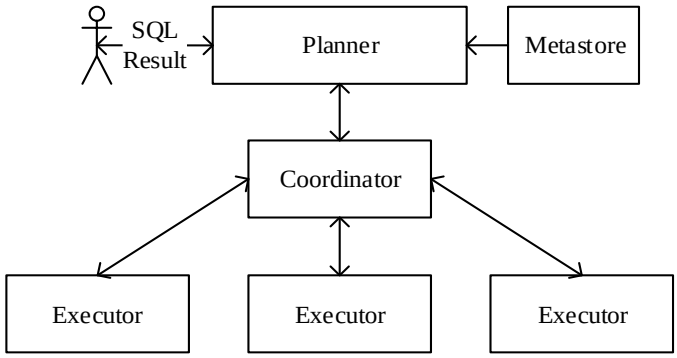

图 2.1 Impala 1.0 查询处理流程图

2.2.1 语法解析

Impala 使用 JFlex[47]和 CUP（Construction of Useful Parsers）[48]对 SQL 进行解析，生成解析树。JFlex 是一个开源的词法分析器，基于 Elliot Berk 的 JLex[49]进行了重写以用于 Java 应用。JFlex 可以与 LALR 解析生成器 CUP 一起工作，也可以与 Java 版的 BYacc/J[50]以及其它解析生成器如 ANTLR[51]一起工作，甚至可以作为一个独立的工具。CUP 是一个 Java 版的 LALR 解析生成器，其实现了标

准的 LALR(1)分析方法，可以与词法分析器 JFlex 一起对语句进行解析。

解析生成的 parse trees 由 SelectStmt 对象表示，其主要包含 selectList（选择的列）、tableRefList（参与查询的表）、wherePredicate（where 子句）、groupingExprs（group by 表达式）、havingPredicate（having 子句）、orderByClause（order by 语句）和 limitOffsetClause（limit 语句）。

2.2.2 查询计划生成

查询语句经过解析和分析后，会生成一组查询计划片段，在逻辑上可以用查询计划树来进行表示，对于一个多表连接查询$R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$，Impala 1.0 生成的查询计划树如图 2.2 所示。其中，Scan Node 为扫描节点，负责从底层存储（如 HDFS、HBase 等）获取数据；Exchange Node 为数据交换节点，负责将本节点的数据发送到目标节点；HashJoin Node 为哈希连接节点，负责执行 2 个表的哈希连接操作。可以看出，该查询计划树中的所有右节点均为 Scan Node 和 Exchange Node 的组合，是典型的左深树形式的查询计划。

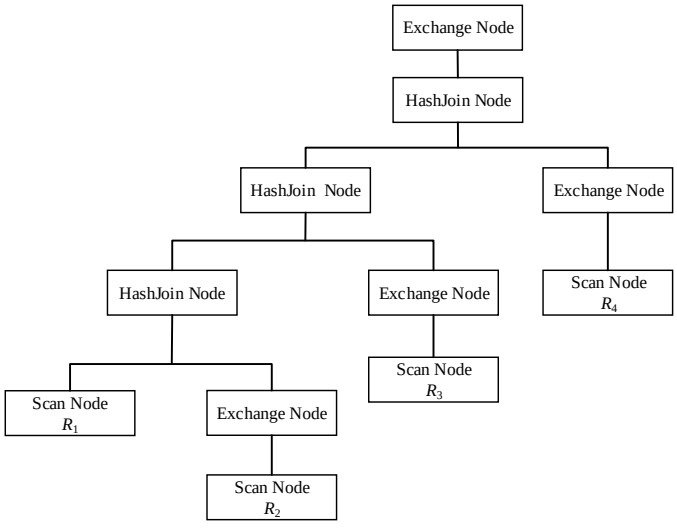

图 2.2 查询计划树

对于图 2.2 所示的查询计划，其执行步骤如下：

1）通过 Scan Node 获取 R_1 和 R_2 的数据，并通过 Exchange Node 将 R_2 的数据

发送到 R_1 所在的节点；

2）在 R_1 所在的节点上执行哈希连接操作；

3）通过 Scan Node 获取 R_3 的数据，并通过 Exchange Node 将数据发送到 R_1 所在的节点；

4）在 R_1 所在的节点上执行哈希连接操作；

5）通过 Scan Node 获取 R_4 的数据，并通过 Exchange Node 将数据发送到 R_1 所在的节点；

6）在 R_1 所在的节点上执行哈希连接操作；

7）将查询结果返回给用户。

2.2.3 执行计划生成

查询计划生成后，系统根据查询计划片段生成对应的执行计划：首先为各查询片段分配唯一的编号；接着为各查询片段中的 Scan Nodes 分配数据扫描的位置和范围；最后创建结果集的元数据信息，包括结果集的类型，数据列的数量、名称和类型等。

当执行计划生成后，由 Coordinator 对执行计划进行分发，并由 Executor 进行具体的查询操作。

2.2.4 查询和汇总

Executor 获得执行计划后，会启动一个线程进行查询操作，并且用户可以在查询过程中通过快捷键（Ctrl+c）终止查询。Executor 为每一个具体的子操作提供了实现，包括聚合、哈希连接、HBase 表的扫描、HDFS 中多种存储格式文件的扫描、选择操作和 top n 操作等。同时，通过实现 MergeNode 以支持对子查询结果进行汇总和过滤。

2.3 Impala 2.0 系统对查询过程的改进

Impala 2.0 在 Impala 1.0 的基础上增加了 Catalog Server[52]，Catalog Server 负责收集和更新元数据信息。在 Catalog Server 之前，Impala 通过 JDBC 插件直接从 Hive 的元数据表中获取表的元数据信息，当表的数据更新时（如插入、删除、更

新表数据），用户需要手动输入命令来让 Hive 更新元数据表。当用户在 Impala 上执行数据更新操作时，Catalog Server 可以自动的对元数据进行更新，不需要用户再手动执行元数据更新命令。

同时，Impala 2.0 系统还提供了查询优化功能，当统计信息可用时，会使用简单的代价模型对多表连接查询操作按表从大到小的顺序进行优化，并选择使用 broadcast join 或 partitioned join 的方式执行查询操作[53]。Impala 2.0 系统的查询处理流程图如图 2.3 所示。

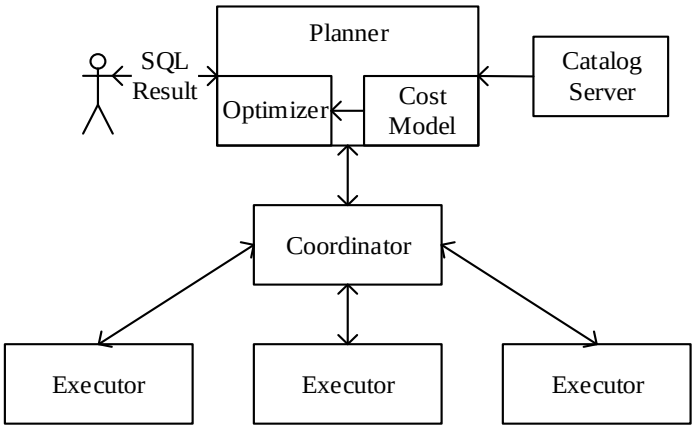

图 2.3 Impala 2.0 查询处理流程图

2.4 本章小结

本章首先对 Impala 大数据实时查询系统进行了介绍；接着介绍了 Impala 中 2 个具有里程碑意义的版本 Impala 1.0 和 Impala 2.0；然后对 Impala 1.0 的查询过程进行了详细的说明，并重点分析了语法解析、查询计划生成、执行计划生成和查询汇总几个过程；最后介绍了 Impala 2.0 对查询过程的改进。

第3章 代价估计

3.1 概述

代价估计主要分为统计信息收集、代价模型和代价估计方法三个部分，代价估计流程图如图 3.1 所示。

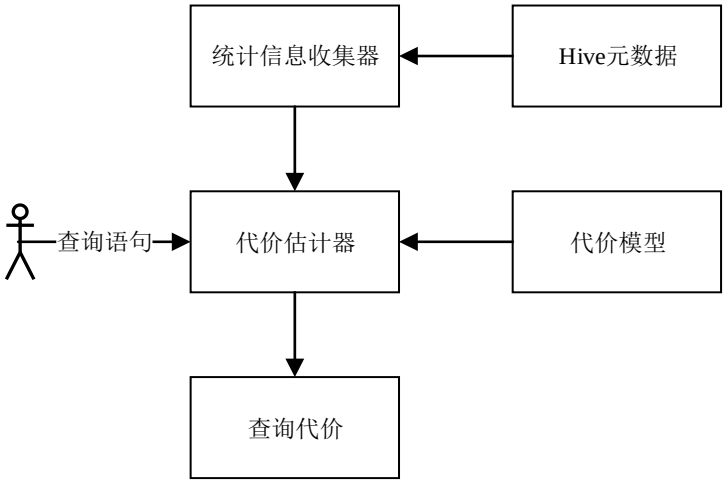

图 3.1 查询代价估计流程图

用户的查询语句经过解析，在构建查询计划树的过程中由代价估计器计算每一棵子树的查询代价。代价估计依赖于统计信息和代价模型。统计信息由 Hive 元数据提供，并由系统或用户保证元数据与数据的一致性，避免因元数据过时而导致代价估计不准确。获得的查询代价主要用于查询优化算法，本文提出的查询优化算法是基于代价的算法，在递归计算时会保存已计算的代价，避免重复计算。

3.2 统计信息收集

Hive 0.13 版本的元数据总共有 51 个表，统计信息收集器主要使用其中 3 个表：TBLS、TABLE_PARAMS 和 TAB_COL_STATS，3 个表的定义如表 3.1~表 3.3 所示。

表 3.1 TBLS 表定义

Field	Type	Null	Key	Default	Extra
TBL_ID	bigint(20)	NO	PRI	NULL	
CREATE_TIME	int(11)	NO		NULL	
DB_ID	bigint(20)	YES	MUL	NULL	
LAST_ACCESS_TIME	int(11)	NO		NULL	
OWNER	varchar(767)	YES		NULL	
RETENTION	int(11)	NO		NULL	
SD_ID	bigint(20)	YES	MUL	NULL	
TBL_NAME	varchar(128)	YES	MUL	NULL	
TBL_TYPE	varchar(128)	YES		NULL	
VIEW_EXPANDED_TEXT	mediumtext	YES		NULL	
VIEW_ORIGINAL_TEXT	mediumtext	YES		NULL	
LINK_TARGET_ID	bigint(20)	YES	MUL	NULL	

表 3.2 TABLE_PARAMS 表定义

Field	Type	Null	Key	Default	Extra
TBL_ID	bigint(20)	NO	PRI	NULL	
PARAM_KEY	varchar(256)	NO	PRI	NULL	
PARAM_VALUE	varchar(4000)	YES		NULL	

表 3.3 TAB_COL_STATS 表定义

Field	Type	Null	Key	Default	Extra
CS_ID	bigint(20)	NO	PRI	NULL	
DB_NAME	varchar(128)	NO		NULL	
TABLE_NAME	varchar(128)	NO		NULL	
COLUMN_NAME	varchar(128)	NO		NULL	
COLUMN_TYPE	varchar(128)	NO		NULL	
TBL_ID	bigint(20)	NO	MUL	NULL	
LONG_LOW_VALUE	bigint(20)	YES		NULL	
LONG_HIGH_VALUE	bigint(20)	YES		NULL	
DOUBLE_HIGH_VALUE	double(53,4)	YES		NULL	
DOUBLE_LOW_VALUE	double(53,4)	YES		NULL	
BIG_DECIMAL_LOW_VALUE	varchar(4000)	YES		NULL	
BIG_DECIMAL_HIGH_VALUE	varchar(4000)	YES		NULL	
NUM_NULLS	bigint(20)	NO		NULL	
NUM_DISTINCTS	bigint(20)	YES		NULL	
AVG_COL_LEN	double(53,4)	YES		NULL	
MAX_COL_LEN	bigint(20)	YES		NULL	
NUM_TRUES	bigint(20)	YES		NULL	

续表 3.3

Field	Type	Null	Key	Default	Extra
NUM_FALSES	bigint(20)	YES		NULL	
LAST_ANALYZED	bigint(20)	NO		NULL	

下面通过具体例子来展示 Hive 存储的元数据信息。有一个简单的测试表 test，其只含有一列 int 类型的名为 number 的域，里面有从 1 到 6 共 6 行的整数。对该表分析后的元数据信息如表 3.4~表 3.6 所示。

表 3.4 TBLS 表中数据（纵向表示）

Field	Value
TBL_ID	170
CREATE_TIME	1415947603
DB_ID	1
LAST_ACCESS_TIME	0
OWNER	impala
RETENTION	0
SD_ID	483
TBL_NAME	test
TBL_TYPE	MANAGED_TABLE
VIEW_EXPANDED_TEXT	NULL
VIEW_ORIGINAL_TEXT	NULL
LINK_TARGET_ID	NULL

表 3.5 TABLE_PARAMS 表中数据

TBL_ID	PARAM_KEY	PARAM_VALUE
170	COLUMN_STATS_ACCURATE	true
170	numFiles	1
170	numRows	6
170	rawDataSize	6
170	totalSize	12
170	transient_lastDdlTime	1418976155

表 3.6 TAB_COL_STATS 表中数据（纵向表示）

Field	Value
CS_ID	1
DB_NAME	default

续表 3.6

Field	Value
TABLE_NAME	test
COLUMN_NAME	number
COLUMN_TYPE	int
TBL_ID	170
LONG_LOW_VALUE	1
LONG_HIGH_VALUE	6
DOUBLE_HIGH_VALUE	NULL
DOUBLE_LOW_VALUE	NULL
BIG_DECIMAL_LOW_VALUE	NULL
BIG_DECIMAL_HIGH_VALUE	NULL
NUM_NULLS	0
NUM_DISTINCTS	5
AVG_COL_LEN	0.0000
MAX_COL_LEN	0
NUM_TRUES	0
NUM_FALSES	0
LAST_ANALYZED	1418978598

可以看出，Hive 元数据会保存表的行数、原始数据大小、总大小，每一列类型、空值的数量、不同值的数量。对于数值型的列，会统计最大最小值；对于字符型的列，会统计最大长度和平均长度；对于布尔类型的列，会统计 TRUE 和 FALSE 的数量。当通过 Impala 进行数据操作时，其会自动对元数据进行统计和更新；当通过 Hive 进行数据操作时，需要在 Impala 下执行“COMPUTE STATS 表名”来更新元数据信息。

3.3 代价模型

本文在第 1 章介绍了几种代价模型，分别有各自的特点和适用范围。通用的代价模型通常无法获得最优的代价估计效果，设计和选择代价模型需要综合考虑系统的特点和各因素对查询执行的影响。Impala 1.0 在执行哈希连接时，默认会将右表的全部数据发送到左表所在的各个节点，当右表的大小超过系统可用内存时，查询就会失败。对于支持浓密树形式查询计划的分布式查询系统，其右节点也可以是哈希连接节点，因此哈希连接的结果大小对查询的执行也有一定的影响。同时，若左右 2 节点均为哈希连接节点，则这 2 节点可以并行执行。

本文提出的代价模型的优化目标是减少单个查询语句的执行时间，需综合考虑磁盘读取的代价、网络传输的代价、右表的大小及自然连接后的表大小。由于目前从元数据信息中只能获取表中数据的行数和总字节数，因此在计算磁盘和网络代价时使用总字节数，而计算表的大小时使用数据的行数。

本文提出的代价模型更加适用于支持浓密树形式查询计划的分布式查询系统（如后面提出的改进的 Impala 系统），对于一棵查询树 T，其左右子树分别为 L 和 R，查询代价模型见公式（3.1）。

$$C_{\text{improved}} = \begin{cases} \alpha_L IO_L + \alpha_R IO_R + \beta TR_R + \gamma S_R + \delta S_{LR} & \text{（a）} \\ \alpha_L IO_L + \alpha_R C_R + \beta TR_R + \gamma S_R + \delta S_{LR} & \text{（b）} \\ \alpha_L C_L + \alpha_R IO_R + \beta TR_R + \gamma S_R + \delta S_{LR} & \text{（c）} \\ (\alpha_L + \alpha_R) Max(C_L, C_R) + \beta TR_R + \gamma S_R + \delta S_{LR} & \text{（d）} \end{cases} \quad \text{公式（3.1）}$$

当 L 和 R 均为单一表时，使用（a）计算；当 L 为单一表，而 R 不为单一表时，使用（b）计算；当 L 不为单一表，而 R 为单一表时，使用（c）计算；当 L 和 R 均不为单一表时，使用（d）计算。

其中$\alpha_L + \alpha_R + \beta + \gamma + \delta = 1$，$C_x$ 为子树 x 的代价，IO_x 为磁盘读取子树 x 对应的数据的代价，TR_x 为网络传输子树 x 对应的数据的代价，S_x 为子树 x 的数据大小，S_{xy} 为子树 x 与子树 y 进行哈希连接后结果的大小。

关于参数的取值，需要考虑系统的硬件情况，对于磁盘性能相对较弱或与其他 IO 密集型程序共享硬件的系统，需要增加α_L和α_R的权重。对于局域网环境，可以减少β的权重；而对于广域网的环境，需要适当增加其权重。由于 Impala 系统的特性，建议将γ和δ设为固定值。

3.4 代价估计方法

本研究使用基于表的代价估计方法和基于列的代价估计方法来计算相关代价。当表在存储时未进行划分时，使用基于表的代价估计方法；当表在存储时进行了划分，则使用基于列的代价估计方法。

3.4.1 基于表的代价估计方法

基于表的估计方法计算代价时，主要依据表的行数计数。表的行数计数根据

表的势和表的选择度综合得出，计算公式如公式（3.2）所示。

$$T_{\text{numRows}} = T_{\text{card}} \times T_{\text{sel}} \quad \text{公式（3.2）}$$

其中T_{card}为表的势，T_{sel}为表的选择度，表的选择度可以由有过滤条件的列的势除以有过滤条件的列的总行数得到，计算公式如公式（3.3）所示。

$$T_{\text{sel}} = \frac{T_{\text{Col}_{\text{card}}}}{T_{\text{Col}_{\text{total}}}} \times 100\% \quad \text{公式（3.3）}$$

其中$T_{\text{Col}_{\text{card}}}$为有过滤条件的列的势，$T_{\text{Col}_{\text{total}}}$为有过滤条件的列的总行数。例如有一张表，其有 100000 行数据，完全不相同的行的个数为 80000，有过滤条件的列的不同行个数为 50000，则该表的行数计数为$T_{\text{numRows}} = 80000 \times \frac{50000}{100000} \times 100\% = 40000$。

3.4.2 基于列的代价估计方法

Impala 底层存储常使用 HDFS，HDFS 支持列式存储，会将关系数据库中的表进行水平和垂直划分。在执行哈希连接操作时，不用将整表的数据进行传输，而只需传输需要参与计算的列（列簇）。

基于列的代价估计方法计算代价时，主要依据表的行数计数和列的行数计数。表的行数计数根据表的势和表中所有有过滤条件的列的选择度乘积综合得到，计算公式如公式（3.4）所示。

$$T_{\text{numRows}} = T_{\text{card}} \times \prod_{k=1}^{n} \text{Col}_{\text{sel}_k} \quad \text{公式（3.4）}$$

其中T_{card}为表的势，$\text{Col}_{\text{sel}_k}$表示有过滤条件的列的选择度，计算方法同公式（3.3）。列的行数计数根据选定的列的势和该列的选择度综合计算得到，计算公式如公式（3.5）所示。

$$Col_{\text{numRows}} = Col_{\text{card}} \times Col_{\text{sel}} \quad \text{公式（3.5）}$$

其中Col_{card}为选定的列的势，Col_{sel}为选定的列的选择度，计算方法同公式（3.3）。

3.5 本章小结

本章介绍了代价估计，首先给出了在 Impala 系统中进行代价估计的流程；接

着介绍了统计信息收集方法，主要依赖 Hive 元数据的 3 张表的信息；然后提出了一种适用于浓密树形式的查询计划的代价模型；最后给出两种代价估计方法：基于表的代价估计方法和基于列的代价估计方法。

第4章 基于超图和浓密树的大数据实时查询优化

4.1 概述

本章将提出基于超图和浓密树的大数据实时查询优化方法。如第 1 章所述，浓密树形式的查询计划更适用于分布式系统，通过子树的并发查询可以提高查询的效率。本章将对目前效率最好的自顶向下查询优化算法 McCHyp 进行优化，该算法基于超图划分，可以支持多元谓词、多种连接操作，可以较好的满足 Impala 大数据实时查询系统的查询优化需求。

本章结构安排如下：首先给出查询优化算法相关的基本定义；接着对超图建模的方法进行说明；然后介绍基于超图的自顶向下查询优化算法 McCHyp，并使用剪枝策略对算法进行优化；最后分析并说明剪枝算法的完整性和正确性，保证使用改进后的算法可以获得最优的查询计划。

4.2 基本定义

在介绍优化方法前首先给出一些定义。

定义 4.1 单一关系。给定一棵连接树 T，T 的左、右子树分别是 T_{L} 和 T_{R}；若 T_{L} 没有左子树，也没有右子树，则称 T_{L} 为一个单一关系（T_{R} 同理）。

定义 4.2 左深树。给定一棵连接树 T，T 的左、右子树分别是 T_{L} 和 R，其中 R 表示一个单一关系；若满足：（1）T_{L} 和 R 均存在，（2）T_{L} 为单一关系，或其所有子树中右子树均为单一关系，则称 T 为左深树。

定义 4.3 浓密树。给定一棵连接树 T，T 的左、右子树分别是 T_{L} 和 T_{R}；若满足：（1）T_{L} 和 T_{R} 均存在，（2）T_{L} 为一棵子树，或为单一关系（T_{R} 同理），则称 T 为浓密树。

定义 4.4 查询超图。查询超图$G = (V, E)$，其中 V 为非空点集，表示所有参与连接操作的表（单一关系），E 为一组超边的集合，表示表之间的所有连接谓词。其中超边是一个无序对(u, v)，u 和 v 是属于点集 V 的非空子集，且满足$u \cap v =$

$\emptyset$。

对于一条超边$e=(u,v)$，若满足$|u|=|v|=1$，则超边就成为一条简单边；当超图中的所有边都是简单边时，超图就成为一个简单图。

定义 4.5 连接超图。给定一个查询超图$G=(V,E)$，若满足$\forall v_1,v_2\in V,\exists(v_1\in u,v_2\in v\vee v_1\in v,v_2\in u),(u,v)\in E$，则称查询超图 G 为一个连接超图。

定义 4.6 连接子图及其补集对。对于一个连接超图$G=(V,E)$，S_1，S_2 为 V 的子集，当满足以下条件时，(S_1,S_2)称为连接子图及其补集对：

1) $S_1\subset V$，且G_{S_1}为连接超图，
2) $S_2\subset V$，且G_{S_2}为连接超图，
3) $S_1\cap S_2=\emptyset$，且
4) $\exists(v_1,v_2)\in E|v_1\in S_1\wedge v_2\in S_2$。

4.3 查询超图建模

查询语句可通过查询超图来进行表示，查询语句 From 子句的表对应查询超图中的顶点集，查询语句 Where 子句、On 子句以及 Using 子句的连接谓词对应查询超图的边集。本小节将通过具体示例来介绍如何对一条查询语句进行查询超图建模。

给定一条查询语句 Select * From T_1, T_2, T_3, T_4, T_5, T_6 Where $T_1.c_1 = T_2.c_2$ and $T_3.c_3 + T_4.c_4 = T_5.c_5 + T_6.c_6$，由连接谓词可以得到两条超边$E_1=(\{T_1\},\{T_2\})$，$E_2=(\{T_3,T_4\},\{T_5,T_6\})$，点集$V=\{T_1,T_2,T_3,T_4,T_5,T_6\}$，超边集$E=\{E_1,E_2\}$，查询超图如图 4.1 所示。

本文提出的自顶向下查询优化方法会对超图进行递归划分，在划分过程中会得到多个子图，子图的表示方法也如上所述，子图的点集是点集 V 的子集，而子图的超边集是超边集 E 的子集，或超边集 E 中某条超边的子集。

4.4 改进的 McCHyp 算法

在介绍改进算法前，首先介绍基于超图和浓密树的自顶向下优化算法 McCHyp。其输入参数为查询超图，输出为一棵查询树，主要执行步骤如下：

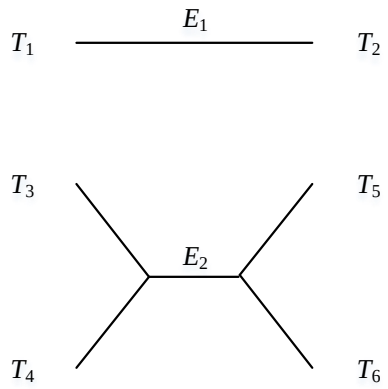

图 4.1 查询超图

1) 对超图中的每一个点（即单一关系）进行初始化，获取代价信息后保存到全局的最优树集合中；
2) 生成超图的所有划分；
3) 每一个划分均将超图分为 2 部分，对这 2 部分分别递归调用步骤 2）直到不可再分为止，返回最优树集合中对应的子树；
4) 将每次递归返回的 2 棵子树分别正序和反序合并，并比较合并后的树的代价，若小于最优树集合中对应的代价，则替换最优树集合中原来的子树；
5) 递归完成后，返回最优集合中包含超图中所有点的那棵树。

由于该算法需要对超图的所有划分进行比较，而正如前面所述，浓密树的划分有$\binom{2(n-1)}{n-1}(n-1)!$种可能，但最终只选用其中一种，当连接表的数量较多时计算代价很大。因此本文使用剪枝策略对算法进行改进，减少计算量。

4.4.1 集成剪枝策略的改进的 McCHyp 算法

剪枝策略主要适用于自顶向下的算法，在由整体到局部的遍历过程中，“剪”去不必要的分枝，减少计算量。集成剪枝策略的改进的 McCHyp 算法如算法 4.1 所示。

算法 4.1 Improved-McCHyp.
输入：*graph*，*budget*；
输出：*tree*。
①*tree* = *bestTree*[*graph*]；
②if *tree* != null and cost(*tree*) <= *budget*
③ then return *tree*；
④end if
⑤*attempt* = *attempts*[*graph*]；
⑥if *attempt* == 0
⑦ then *attempts*[*graph*] = 1；
⑧else
⑨ if *budget* < upperBound(*graph*)
⑩ then *budget* = upperBound(*graph*)；
⑪ else
⑫ *budget* = max(*budget*, lowerBound(*graph*) $\times 2^{attempt}$)；
⑬ end if
⑭end if
⑮*partitions* = partition(*graph*)；
⑯for each *partition* in *partitions*
⑰ *budget2* = min(*budget*, cost(*bestTree*[*graph*]))；
⑱ if *bestTree*[*partition*.right( )] != null
⑲ then *cr* = cost(*bestTree*[*partition*.right( )])；
⑳ else
㉑ *cr* = lowerBound(*partition*.right( ))；
㉒ end if
㉓ *leftTree* = Improved-McCHyp(*partition*.left( ), *budget2* - *cr*)；
㉔ if *leftTree* != null
㉕ then *rightTree* = Improved-McCHyp(*partition*.right( ), *budget2* - cost(*leftTree*))；
㉖ buildTree(*graph, leftTree, rightTree, budget*)；
㉗ end if
㉘end for
㉙return *bestTree*[*graph*]。

本算法的具体执行过程如下：首先从最优集合中获取包含超图的查询树，若不为空且代价小于等于 *budget*，则返回该查询树，如伪代码行①~④所示；接下来获取对该超图的尝试次数，若次数为 0，则置为 1，否则更新 *budget*，如行⑤~⑭所示；然后将超图进行划分，如行⑮所示；遍历每一个被划分的超图，估计右子

图的代价，并递归调用 Improved-McCHyp 算法获得左子树，如行⑰~㉓所示；若左子树不为空，则递归调用 Improved-McCHyp 算法获得对应的右子树，如行㉔㉕所示；然后将 2 棵子树分别正序和反序合并，并比较合并后的树的代价，若小于最优树集合中对应的代价，则替换最优树集合中原来的子树，如行㉖所示；最后返回最优集合中包含超图中所有点的那棵树，如行㉙所示。由于算法在遇到现有代价超过上界或已知最优代价时便不再计算，因此节省了许多计算量，提高了 McCHyp 算法的效率。

4.4.2 改进的 McCHyp 算法优化过程举例

下面通过对一个 4 表连接查询语句的查询优化来解释改进的 McCHyp 算法的优化过程。对于一个 4 表连接查询语句，首先构造查询超图 $G\{r_1, r_2, r_3, r_4\}$（其中 r_1, r_2, r_3, r_4 表示表，为便于描述，此处省略对边的表示），查询超图 $G\{r_1, r_2, r_3, r_4\}$ 经划分后形成多个子图，如图 4.2 所示，后续描述将基于此进行。以查询超图 $G\{r_1, r_2, r_3, r_4\}$作为分解目标，进行第一级分解，得到两个子图对，分别为子图对 $G11$ 和子图对 $G12$，子图对 $G11$ 包括两个查询超图分别为 $G_1\{r_1\}$和 $G_5\{r_2, r_3, r_4\}$，子图对 $G12$ 包括两个查询超图分别为 $G_7\{r_1, r_2\}$和 $G_6\{r_3, r_4\}$。

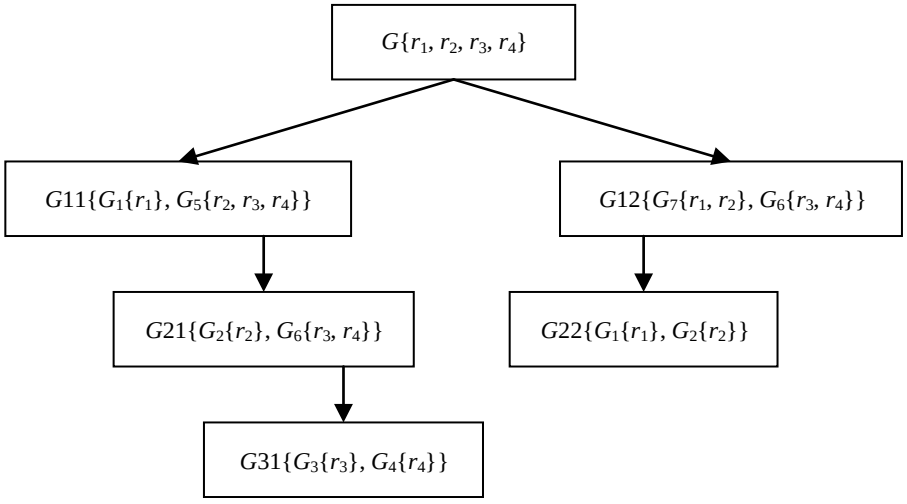

图 4.2 超图划分结果图

$G_1\{r_1\}$为单表，不用继续分解。进一步以一级分解得到的查询超图 $G_5\{r_2, r_3,$

r_4}、G_7\{r_1, r_2\}和 G_6\{r_3, r_4\}作为分解目标进行二级分解。G_5\{r_2, r_3, r_4\}分解得到一个子图对 $G21$，包括两个查询超图，分别为 G_2\{r_2\}和 G_6\{r_3, r_4\}。进一步以 G_6\{r_3, r_4\}作为三级分解目标继续进行分解，得到一个子图对 $G31$，包括两个查询超图 G_3\{r_3\}和 G_4\{r_4\}。

对查询超图 G_7\{r_1, r_2\}分解得到一个子图对 $G22$，包括两个查询超图 G_1\{r_1\}和 G_2\{r_2\}。

根据算法 4.1，本查询超图的逐级分解过程如下：

1) 初始化全局最优树映射和全局尝试次数映射，如表 4.1 所示，将查询超图 G\{r_1, r_2, r_3, r_4\}放入全局最优树映射，对应最优解为 Ø，同时把单一表对应的查询计划树放入全局最优树映射。

表 4.1 全局最优树映射（1）

超图	查询计划树
G\{r_1, r_2, r_3, r_4\}	Ø
G_1\{r_1\}	T_1
G_2\{r_2\}	T_2
G_3\{r_3\}	T_3
G_4\{r_4\}	T_4

将超图 G\{r_1, r_2, r_3, r_4\}放入全局尝试次数映射中，对应尝试次数为 0，得到的全局尝试次数映射如表 4.2 所示。

表 4.2 全局尝试次数映射（1）

超图	尝试次数
G\{r_1, r_2, r_3, r_4\}	0

2) 从全局最优树映射中获取 G\{r_1, r_2, r_3, r_4\}对应的最优查询计划树，发现是 Ø（不存在），因此从全局尝试次数映射中获取 G\{r_1, r_2, r_3, r_4\}的尝试次数，并根据获取的尝试次数利用公式（4.1）更新 budget。

$$budget = \max(budget, \text{lowerBound}(graph) \times 2^{attempt})$$　公式（4.1）

其中 *graph* 为超图（即查询超图），*attempt* 为尝试次数，lowerBound 用于获取查询超图的代价下界，通常使用简单的代价模型来计算下界，以保证 $\text{lowerBound}(graph) \leq \text{cost}(graph)$，cost(*graph*)表示 *graph* 的实际代价，此时 *graph* = *G*。

设初始时的 *budget* 为 b_0，值为正无穷，更新后的值为 b_0'。

更新 *budget* 后将其全局尝试次数映射中对应的尝试次数加 1。更新后的全局尝试次数映射如表 4.3 所示。

表 4.3 全局尝试次数映射（2）

超图	尝试次数
$G\{r_1, r_2, r_3, r_4\}$	1

3) 以 $G\{r_1, r_2, r_3, r_4\}$作为分解目标，对其进行划分，生成两个子图对，$G11\{G_1\{r_1\}, G_5\{r_2, r_3, r_4\}\}$和 $G12\{G_7\{r_1, r_2\}, G_6\{r_3, r_4\}\}$。

4) 下面对子图对 $G11\{G_1\{r_1\}, G_5\{r_2, r_3, r_4\}\}$进行分析，分析时以 $G_5\{r_2, r_3, r_4\}$作为右划分，以 $G_1\{r_1\}$作为左划分，以 b_0'作为子图对 $G11\{G_1\{r_1\}, G_5\{r_2, r_3, r_4\}\}$对应的 *budget* 基准值。

a) 首先判断右划分是否已存在于全局最优树映射中，并根据判断结果基于相应的更新准则更新 *budget*，然后利用更新后的 *budget* 构建 $G_1\{r_1\}$对应的左子树。若右划分已存在于最优树映射中，直接计算对应查询计划树的代价，并以 *budget* 与计算得到的查询计划树的代价的差值作为新的 *budget*；否则，计算其代价下界，并更新 *budget* 为 *budget* 与代价下界的差值。

在本例中右划分 $G_5\{r_2, r_3, r_4\}$不在全局最优树映射中，根据以上方法更新该子图对对应的 *budget* 基准值（此时 *budget* 基准值为 b_0'），更新后的 *budget* 为 c_{11}。

从全局最优树映射中获取 $G_1\{r_1\}$对应的最优树，为 T_1，直接返回

最优查询计划树 T_1（本文算法规定单表的查询计划树的代价总是小于给定的 *budget*）。

b) 根据左子树的代价利用公式（4.2）更新 $G11\{G_1\{r_1\}, G_5\{r_2, r_3, r_4\}\}$ 对应的 *budget* 基准值，并根据更新后的 *budget* 构建 $G_5\{r_2, r_3, r_4\}\}$对应的查询计划树作为右子树。

$$budget' = budget - \text{cost}(left) \qquad \text{公式（4.2）}$$

其中$\text{cost}(left)$表示构建的左子树的代价。

本例中更新后的 *budget* 为 $b_1 = b_0' - c_{12}$，其中 c_{12} 为 $G_1\{r_1\}$的代价。

本例中构建 $G_5\{r_2, r_3, r_4\}$对应的查询计划树作为右子树的具体步骤如下：

b.1) 从全局最优树映射中获取 $G_5\{r_2, r_3, r_4\}$对应的最优查询计划树，不存在，则从全局尝试次数映射中获取 $G_5\{r_2, r_3, r_4\}$对应的尝试次数，不存在，初始化为 0，并将得到的尝试次数代入公式（4.1）以更新 *budget*，更新后的 *budget* 为 b_1'。

同时将尝试次数增 1 后保存到全局尝试次数映射中。此时，更新后的全局尝试次数映射如表 4.4 所示。

表 4.4 全局尝试次数映射（3）

超图	尝试次数
$G\{r_1, r_2, r_3, r_4\}$	1
$G_5\{r_2, r_3, r_4\}$	1

b.2) 对查询超图 $G_5\{r_2, r_3, r_4\}$进行划分，生成一个子图对 $G21\{G_2\{r_2\}, G_6\{r_3, r_4\}\}$，以 $G_6\{r_3, r_4\}$为右划分，以 $G_2\{r_2\}$为左划分，以 b_1'作为子图对 $G21\{G_2\{r_2\}, G_6\{r_3, r_4\}\}$的 *budget* 基准值，继续进行如下操作：

b.2.1) 判断右划分 $G_6\{r_3, r_4\}$是否已存在于全局最优树映射中，并使用更新准则更新 *budget*，然后使用更新后的 *budget*

构建 $G_2\{r_2\}$对应的左子树。由于 $G_6\{r_3, r_4\}$不在全局最优树映射中，根据更新准则将 *budget* 由基准值 b_1'更新为 c_{21}。

从全局最优树映射中获取 $G_2\{r_2\}$对应的最优树，为 T_2，直接返回最优查询计划树 T_2 作为左子树。

b.2.2) 根据 $G_2\{r_2\}$的代价更新 *budget*，利用公式（4.2）更新该子图对对应的 *budget* 基准值，更新后 *budget* 为 $b_2 = b_1' - c_{22}$，其中 c_{22} 为 $G_2\{r_2\}$的代价（即查询计划树 T_2 的代价），然后构建 $G_6\{r_3, r_4\}$对应的右子树，具体如下：

b.2.2.1) 从全局最优树映射中获取 $G_6\{r_3, r_4\}$对应的最优树，不存在，则从全局尝试次数映射中获取 $G_6\{r_3, r_4\}$对应的尝试次数，不存在，初始化尝试次数为 0，并根据该尝试次数利用公式（4.1）将 *budget* 的取值由 b_2 更新为 b_2'，同时将尝试次数增 1 后保存到全局尝试次数映射中。更新后的全局尝试次数映射如表 4.5 所示。

表 4.5 全局尝试次数映射（4）

超图	尝试次数
$G\{r_1, r_2, r_3, r_4\}$	1
$G_5\{r_2, r_3, r_4\}$	1
$G_6\{r_3, r_4\}$	1

b.2.2.2) 对查询超图 $G_6\{r_3, r_4\}$进行划分，生成一个子图对 $G31\{G_3\{r_3\}, G_4\{r_4\}\}$，以 $G_4\{r_4\}$为右划分，以 $G_3\{r_3\}$左划分，以 b_2'作为子图对 $G31\{G_3\{r_3\}, G_4\{r_4\}\}$对应的 *budget* 基准值。

b.2.2.3) 判断右划分 $G_4\{r_4\}$是否已存在于全局最优树映射中，并使用更新准则更新 *budget*，然后使用更新后的 *budget* 构建 $G_3\{r_3\}$对应的左子树。此时右划分

$G_4\{r_4\}$在全局最优树映射中，并基于相应的更新准则，将 *budget* 的值由 b_2'更新为 c_{31}，并构建 $G_3\{r_3\}$对应的左子树。

b.2.2.4) 从全局最优树映射中获取 $G_3\{r_3\}$对应的最优树，为 T_3，直接返回最优查询计划树 T_3。

b.2.2.5) 利用构建得到的左子树，根据公式（4.2）更新 *budget*，更新后的 *budget* 为 $b_3 = b_2' - c_{32}$，其中 c_{32} 为 $G_3\{r_3\}$的代价。

b.2.2.6) 利用更新后的 *budget*（即 b_3）构建 $G_4\{r_4\}$对应的右子树：从全局最优树映射中获取 $G_4\{r_4\}$对应的最优树，为 T_4，直接返回最优查询计划树 T_4。

b.2.2.7) 将 T_3 和 T_4 进行正序和反序合并，正序和反序合并示意图如图 4.3 所示。正序合并结果为 T_{34}，反序合并结果为 T'_{34}，且 T_{34} 代价小于 T'_{34}。选择代价最小的查询计划树并更新全局最优树映射，更新后的全局最优树映射如表 4.6 所示。

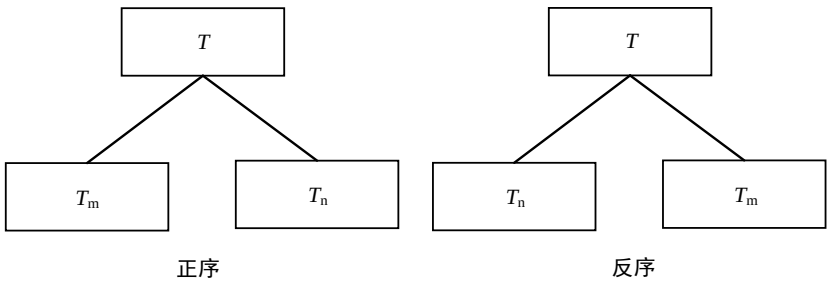

图 4.3 正序和反序合并示意图

b.2.2.8) 所有子图对已计算完毕，返回全局最优树映射中 $G_6\{r_3, r_4\}$对应的最优查询计划树 T_{34}。

表 4.6 全局最优树映射（2）

超图	查询计划树
$G\{r_1, r_2, r_3, r_4\}$	Ø
$G_1\{r_1\}$	T_1
$G_2\{r_2\}$	T_2
$G_3\{r_3\}$	T_3
$G_4\{r_4\}$	T_4
$G_6\{r_3, r_4\}$	T_{34}

b.2.3) 将 T_2 和 T_{34} 进行正序和反序合并，正序合并结果为 T_{234}，反序合并结果为 T'_{234}，且设 T_{234} 代价小于 T'_{234}。选择代价最小的查询计划树并更新全局最优树映射。更新后的全局最优树映射如表 4.7 所示。

表 4.7 全局最优树映射（3）

超图	查询计划树
$G\{r_1, r_2, r_3, r_4\}$	Ø
$G_1\{r_1\}$	T_1
$G_2\{r_2\}$	T_2
$G_3\{r_3\}$	T_3
$G_4\{r_4\}$	T_4
$G_6\{r_3, r_4\}$	T_{34}
$G_5\{r_2, r_3, r_4\}$	T_{234}

b.3) 所有子图对已计算完毕，返回全局最优树映射中 $G_5\{r_2, r_3, r_4\}$对应的最优查询计划树 T_{234}。

c) 将 T_1 和 T_{234} 进行正序和反序合并，设正序合并结果为 T_{1234}，反序合并结果为 T'_{1234}，且设 T_{1234} 代价小于 T'_{1234}。选择代价最小的查询计划树并更新全局最优树映射。更新后的全局最优树映射如表 4.8 所示。

5) 下面对子图对 $G12\{G_7\{r_1, r_2\}, G_6\{r_3, r_4\}\}$进行分析，分析时以 $G_7\{r_1, r_2\}$作为左划分，$G_6\{r_3, r_4\}$为右划分，以 b_0'作为子图对 $G12\{G_7\{r_1, r_2\}, G_6\{r_3, r_4\}\}$对应的 *budget* 基准值。

表 4.8 全局最优树映射（4）

超图	查询计划树
$G\{r_1, r_2, r_3, r_4\}$	T_{1234}
$G_1\{r_1\}$	T_1
$G_2\{r_2\}$	T_2
$G_3\{r_3\}$	T_3
$G_4\{r_4\}$	T_4
$G_6\{r_3, r_4\}$	T_{34}
$G_5\{r_2, r_3, r_4\}$	T_{234}

a) 首先计算右划分 $G_6\{r_3, r_4\}$的代价，以此更新 *budget*。由于 $G_6\{r_3, r_4\}$已存在于全局最优树映射中，因此直接计算其代价并将 *budget* 由基准值 b_0'更新为 c_{41},并构建 $G_7\{r_1, r_2\}$对应的左子树。构建 $G_7\{r_1, r_2\}$对应的左子树的方法如下：

a.1) 从全局最优树映射中获取 $G_7\{r_1, r_2\}$对应的查询计划树，不存在。接着从全局尝试次数映射中获取 $G_7\{r_1, r_2\}$对应的尝试次数，不存在，初始化尝试次数为 0，并以此更新 *budget*，将 *budget* 的值由 c_{41} 更新为 c_{41}'，同时将尝试次数增 1 后保存到全局尝试次数映射中。更新后的全局尝试次数映射如表 4.9 所示。

表 4.9 全局尝试次数映射（5）

超图	尝试次数
$G\{r_1, r_2, r_3, r_4\}$	1
$G_5\{r_2, r_3, r_4\}$	1
$G_6\{r_3, r_4\}$	1
$G_7\{r_1, r_2\}$	1

a.2) 对超图 $G_7\{r_1, r_2\}$进行划分，生成一个子图对 $G22\{G_1\{r_1\}, G_2\{r_2\}\}$。

a.3) 计算右划分 $G_2\{r_2\}$的代价，以此更新 *budget*，将基准值由 c_{41}'更新为 c_{51}，并构建 $G_1\{r_1\}$对应的左子树：从全局最优树映射

中获取 $G_1\{r_1\}$对应的最优查询计划树，为 T_1，直接返回最优查询计划树 T_1。

a.4)　更新 *budget*，$b_5 = c_{41}' - c_{52}$，其中 c_{52} 为 $G_1\{r_1\}$的代价，然后构建 $G_2\{r_2\}$对应的右子树：从全局最优树映射中获取 $G_2\{r_2\}$对应的最优查询计划树，为 T_2，直接返回最优查询计划树 T_2。

a.5)　将 T_1 和 T_2 进行正序和反序合并，不妨设正序合并结果为 T_{12}，反序合并结果为 T'_{12}，且设 T_{12} 代价小于 T'_{12}。选择代价最小的查询计划树并更新全局最优树映射。更新后的全局最优树映射如表 4.10 所示。

表 4.10 全局最优树映射（5）

超图	查询计划树
$G\{r_1, r_2, r_3, r_4\}$	T_{1234}
$G_1\{r_1\}$	T_1
$G_2\{r_2\}$	T_2
$G_3\{r_3\}$	T_3
$G_4\{r_4\}$	T_4
$G_6\{r_3, r_4\}$	T_{34}
$G_5\{r_2, r_3, r_4\}$	T_{234}
$G_7\{r_1, r_2\}$	T_{12}

a.6)　所有子图对已计算完毕，返回全局最优树映射中 $G_7\{r_1, r_2\}$对应的最优查询计划树 T_{12}。

b)　更新 *budget* 并构建 $G_6\{r_3, r_4\}$对应的右子树，具体如下：

b.1)　将 *budget* 由基准值 b_0'更新为 b_4，$b_4 = b_0' - c_{42}$（其中 c_{42} 为 $G_7\{r_1, r_2\}$的代价），并以此构建 $G_6\{r_3, r_4\}$对应的右子树。

b.2)　从全局最优树映射中获取 $G_6\{r_3, r_4\}$对应的最优查询计划树，存在，则将其代价与给定的 *budget* 进行比较：若代价小于给定的 *budget*，则直接返回最优树映射中对应的查询计划树；否则需要继续执行划分过程。

在本例中假设 $G_6\{r_3, r_4\}$对应的最优树代价小于给定 *budget* 基准值 b_4，因此直接返回全局最优树映射中 $G_6\{r_3, r_4\}$对应的最优查询计划树 T_{34}。

c) 将 T_{12} 和 T_{34} 进行正序和反序合并，正序合并结果为 T''_{1234}，反序合并结果为 T'''_{1234}，且设 T''_{1234} 代价小于 T'''_{1234} 和 T_{1234}。选择代价最小的查询计划树并更新全局最优树映射。更新后的全局最优树映射如表 4.11 所示。

表 4.11 全局最优树映射（6）

超图	查询计划树
$G\{r_1, r_2, r_3, r_4\}$	T''_{1234}
$G_1\{r_1\}$	T_1
$G_2\{r_2\}$	T_2
$G_3\{r_3\}$	T_3
$G_4\{r_4\}$	T_4
$G_6\{r_3, r_4\}$	T_{34}
$G_5\{r_2, r_3, r_4\}$	T_{234}
$G_7\{r_1, r_2\}$	T_{12}

6) 所有子图对已计算完毕，返回全局最优树映射中 $G\{r_1, r_2, r_3, r_4\}$对应的最优查询计划树 T''_{1234}。

4.5 剪枝策略的完整性和正确性

首先说明剪枝策略的完整性，伪代码行⑮的 partition 函数将超图进行划分，生成所有的连接子图及其补集对，且在行⑳的 buildTree 函数中，会分别考虑左右子树的两种组合方式，因此不会发生错过某些连接组合的情况，保证了剪枝策略的完整性。

接下来说明剪枝策略的正确性，即保证可以获得最优解，且最优解不会被剪枝掉。在本算法中，主要根据 *budget* 和实际代价来判断是否剪枝，在最上层调用该算法时，*budget* 为∞，防止因初始 *budget* 小于实际代价而得不到最优解，在后

续循环及递归调用中会逐步减少此 *budget*。行⑩保证 *budget* 不小于任何一种可能情况的代价，而行⑫通过不断扩大超图代价的下界来寻找可能的解。

构建查询计划时首先会构建左子树，行⑰~㉒通过减去右子树代价（若已得出右子树，则为实际代价；否则是代价的下界）进一步缩小 *budget*；若返回不为空，则表明存在小于 *budget* 的左子树，并在此基础上递归构建右子树，此时传入的 *budget* 减去的是左子树的实际代价。当返回的右子树也不为空时，则调用 buildTree 函数来构建计划树，在该函数中，会对左右子树两种组合方式的代价分别进行判断，若代价小于已知最优解的代价，并且还小于 *budget*，才组合成一棵最优计划树。

因此，剪枝策略可以保证不错过最优解，同时最优解也不会被剪枝掉。

4.6 本章小结

本章介绍基于超图和浓密树的大数据实时查询优化方法，首先给出查询优化算法相关的定义；接着通过具体示例对超图的建模方法进行说明；然后介绍基于超图的自顶向下查询优化算法 McCHyp，并使用剪枝策略对算法进行优化，同时通过一个示例说明了算法的执行过程；最后分析并说明剪枝算法的完整性和正确性，保证使用改进后的算法可以获得最优的查询计划。接下来会将改进的优化算法集成到大数据实时查询系统 Impala 上，以改进其查询性能。

第5章 系统实现

5.1 系统总体框架

集成改进的 McCHyp 算法的 Impala 系统总体框架如图 5.1 所示，图中有斜线背景的是 Impala 系统的主要组成部分。

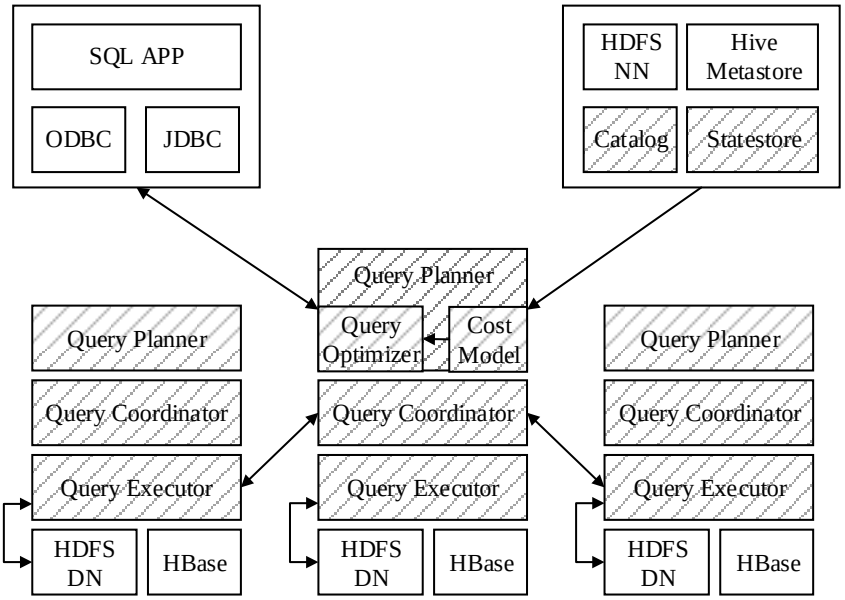

图 5.1 集成改进的 McCHyp 算法的 Impala 总体框架图

系统由以下几部分组成：

1） 数据访问接口

系统支持多种数据访问接口，包括 Cloudera Impala Query UI、ODBC Driver、JDBC Driver、Hue Beeswax 等，还可通过 Impala Shell 进行访问。多种数据访问接口为用户开发多种类型的应用提供了有力保证。

2） 元数据和集群状态管理

Impala 通过 Statestore 监控集群中 Impala 节点的状态，每个 Impala 节点在功能上都是等价的，任何节点都可以接收外部的查询请求，当某个节点发

生故障时，Statestore 会及时感知，并将该节点从有效节点集中删除，保证在分配查询任务时不会将任务分发到失效节点上。

Catalog 负责收集、更新数据的元数据信息，当通过 Impala 进行数据的更新操作（包括插入、更新和删除）时，Catalog 会主动更新 Hive 的元数据信息。

通过 Hadoop 的 Namenode 可以获取当前有效的 Hadoop Datanode，在创建执行计划分配数据读取任务时，会依据此信息进行任务的分配。

3） 查询处理模块

查询处理模块由 Query Planner、Query Coordinator 和 Query Executor 三部分组成，在上文中已对三者的功能进行了描述，本文的工作主要涉及对 Query Planner 的修改。在 Query Planner 中添加 Query Optimizer 和 Cost Model 模块，分别实现查询优化和代价模型（代价计算）的功能，具体实现和集成方法将在本章的后面小节进行说明。

4） 存储引擎

Impala 可以操作 HDFS、HBase 中的数据，支持多种 HDFS 的文件格式，包括文本格式、Sequence File、RCFile、Avro、Parquet 等等，同时还支持多种数据压缩格式，包括 Snappy、GZIP 等。在读取 HDFS 中的文件时，还可以使用 short circuit read 技术直接从本地磁盘中读取数据，提高了数据读取的效率。

5.2 查询计划形式的修改方法

在 2.2.2 小节已对左深树形式的查询计划的执行过程进行了描述，在执行过程中每一步都是串行执行的，并未充分利用分布式系统并行性的特点。通过将查询计划修改为浓密树的形式，可以使查询计划树中每个节点的 2 棵子树并行执行，提高了查询的效率。

图 5.2 展示了一种浓密树形式的查询计划。与图 2.2 的主要区别是右节点也可能是哈希节点，也可以进行哈希连接操作。在这种情况下，R_1 与 R_2，以及 R_3 与

R_4 的哈希连接就可以并行执行了。

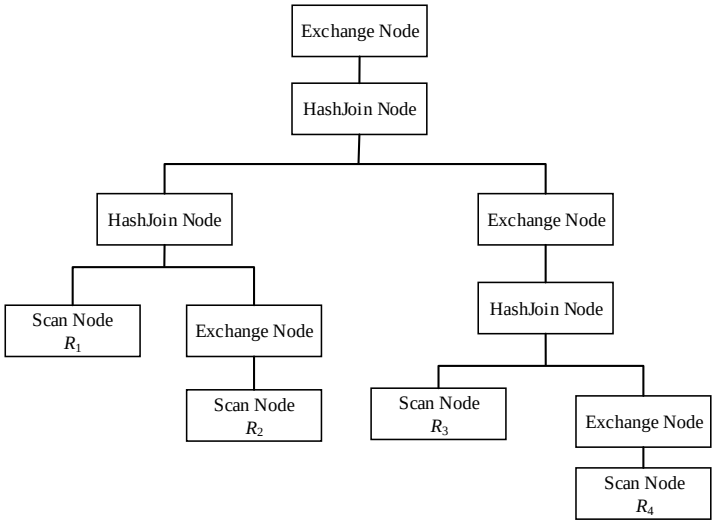

图 5.2 浓密树形式的查询计划

为使 Impala 支持浓密树形式的查询计划，需要提供新的数据结构并修改构造方法。首先创建一个数据结构 JoinTree，用来保存根节点的引用，所有节点用数据结构 JoinNode 来表示；接下来创建一个继承 TableRef 数据结构的 HashJoinRef，并保存对左、右子节点的引用；最后修改构造查询计划的方法。

5.2.1 JoinTree 结构

JoinTree 的结构如图 5.3 所示。JoinTree 保存了该哈希连接树根节点的引用，并提供 get/set 方法供上层调用，同时为方便调试重写了 toString 方法以输出更丰富的信息。

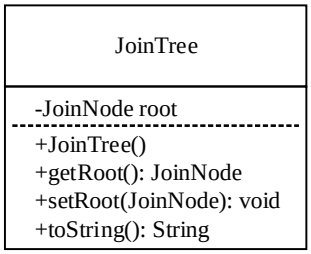

图 5.3 JoinTree 结构

5.2.2 JoinNode 结构

JoinNode 的结构如图 5.4 所示。JoinNode 保存了节点的编号、哈希表的引用、左节点和右节点的引用以及该节点（包含子节点）的代价，并提供了这些变量的 get/set 方法供相应模块调用。下面对该结构的主要方法进行说明。

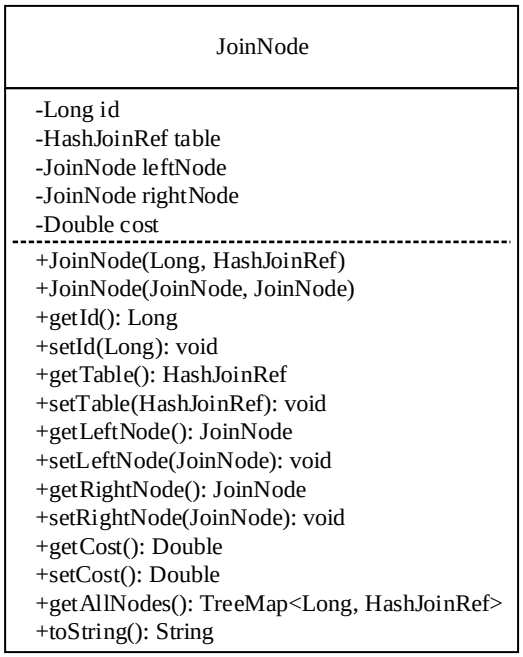

图 5.4 JoinNode 结构

1) public JoinNode(Long id, HashJoinRef table)

该方法为 JoinNode 的构造方法，主要用于创建叶节点，需要传入叶节点的编号和哈希表的引用。

2) public JoinNode(JoinNode leftNode, JoinNode rightNode)

该方法为 JoinNode 的构造方法，主要用于创建中间节点，需要传入左节点和右节点的引用，同时需要修改两子节点对应哈希表的关系，步骤如下：

a) 当右节点的哈希表（以下简称右哈希表，左哈希表同理）的连接操作符为空，且左哈希表的连接操作符不为空，则将左哈希表的连接

操作符复制到右哈希表，并将左哈希表的连接操作符置为空；

b) 当右哈希表的 On 语句为空，且左哈希表的 On 语句不为空，则将左哈希表的 On 语句复制到右哈希表的 On 语句，并将左哈希表的 On 语句置为空；

c) 将右哈希表的 leftTblRef 变量设定为左哈希表，将左哈希表的 leftTblRef 变量置为空。

3) public TreeMap<Long, HashJoinRef> getAllNodes()

该方法返回所有的叶节点编号与对应哈希表的映射，并按编号从小到大排列。构造映射步骤如下：

a) 创建一个空的节点编号与哈希表的映射；

b) 判断左节点与右节点是否均为空，若是，说明该节点为叶节点，将该节点编号和对应的哈希表放入映射中，到步骤 e；若否，到步骤 c；

c) 当左节点不为空时，将左节点调用 getAllNodes 方法返回的映射集保存到当前映射中；

d) 当右节点不为空时，将右节点调用 getAllNodes 方法返回的映射集保存到当前映射中；

e) 返回节点编号与哈希表的映射。

4) public String toString()

该方法由根节点递归输出所有节点信息，包括节点编号、节点之间的关系和节点的代价。

5.2.3 **HashJoinRef 结构**

HashJoinRef 继承 BaseTableRef，用于浓密树形式的查询计划，结构如图 5.5 所示。HashJoinRef 保存了对左节点和右节点的引用，下面对 HashJoinRef 的主要方法进行说明。

1) public HashJoinRef(TableName name, String alias)

该方法为 HashJoinRef 的构造方法，给定表名和表的别名。

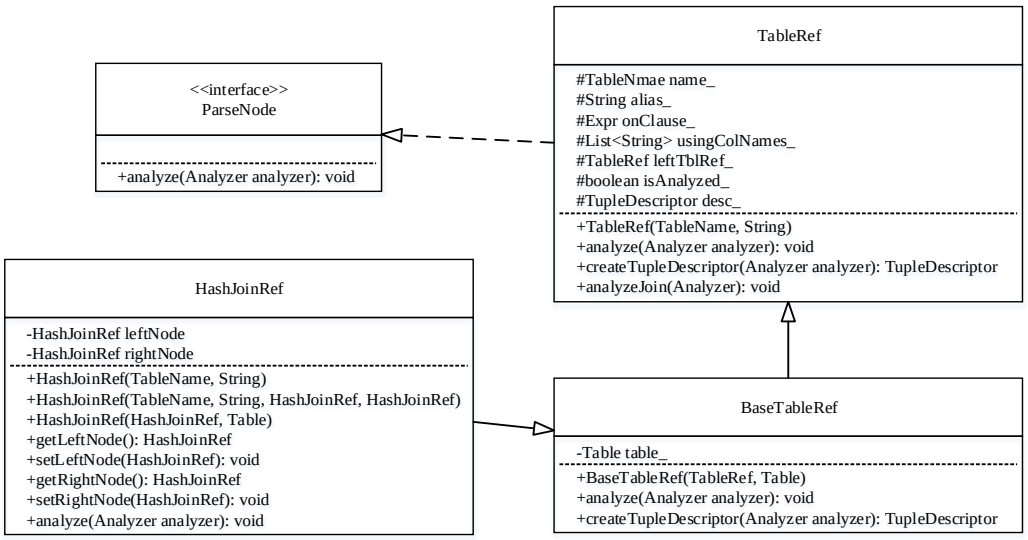

图 5.5 HashJoinRef 结构

2) public HashJoinRef(TableName name, String alias, HashJoinRef leftNode, HashJoinRef rightNode)

该方法为 HashJoinRef 的构造方法，给定表名、表的别名以及表的左节点和右节点的引用，该方法主要用于中间节点。

3) public HashJoinRef(HashJoinRef tableRef, Table tbl)

该方法为 HashJoinRef 的构造方法，给定 HashJoinRef 的引用（可用介绍的第 1 个构造方法获得）和具体存储的引用。具体存储可以是 HBase 表或 HDFS 表。

4) public void analyze(Analyzer analyzer)

该方法会对 HashJoinRef 进行分析，具体步骤如下：

a) 调用 analyzer 的 registerHashJoinRef 方法对具体存储进行注册，并获取描述符（分析的结果）。描述符在 Impala 中用 TupleDescriptor 的对象表示，其主要包括 id、Slot 的集合、具体存储的引用、别名以及数据的大小等信息。Slot 为列的描述信息，在 Impala 中用 SlotDescriptor 的对象表示，主要包括 id、对应 TupleDescriptor 的引用、列类型、列的引用、列的名称以及列的统计信息等；

b) 标记该对象已经分析过（isAnalyzed_设为 true）；

c) 调用父结构（TableRef）的 analyzeJoin 方法对连接语句进行分析。analyzeJoin 方法主要对连接语句进行分析，主要流程如下：首先分析连接操作提示词，Impala 允许用户通过提示词指定使用 broadcast join 还是 partitioned join 进行连接操作，当不指定时默认使用 broadcast join；接着判断 using 子句是否为空，当不为空时，会将 using 子句转化为等价的 on 子句；然后根据连接操作的类型调用 analyzer 的 register*Tid 方法对一些种类的连接操作进行注册，其中*对应的连接类型有 left outer join、right outer join、full outer join、left semi join、left anti join、right semi join 和 right anti join；最后判断 on 子句是否为空，当不为空时，对 on 子句的每一个谓词进行分析，并构建对应的 tuple id。由于外连接和半连接操作均要求 on 子句不为空，因此当 on 子句不存在时，会抛出异常。

5.2.4 构造查询计划的方法

修改后的查询计划构造流程如图 5.6 所示。

该方法执行步骤如下：首先将查询计划置空，并获取当前节点的左右子节点；接下来对子节点进行判断，若不为空，则递归构建左、右子节点的查询计划，然后使用左、右子查询计划构建哈希连接节点，并设置连接谓词。若该节点没有子节点（说明是单一关系），则直接创建表节点。最后，将构造好的查询计划返回。

5.3 改进的 McCHyp 算法在 Impala 中的集成

将改进的 McCHyp 算法集成到 Impala 2.0 系统中，需要对系统前端（fe，front end）部分的代码进行修改并添加一些新的方法，主要有以下几部分。

1) Planner 的 createSelectPlan 方法

修改 Planner 的 private PlanNode createSelectPlan(SelectStmt selectStmt, Analyzer analyzer)方法，在创建所有表的 PlanNode 之后，对连接顺序进行优化，并生成浓密树形式的查询计划，流程如图 5.7 所示。

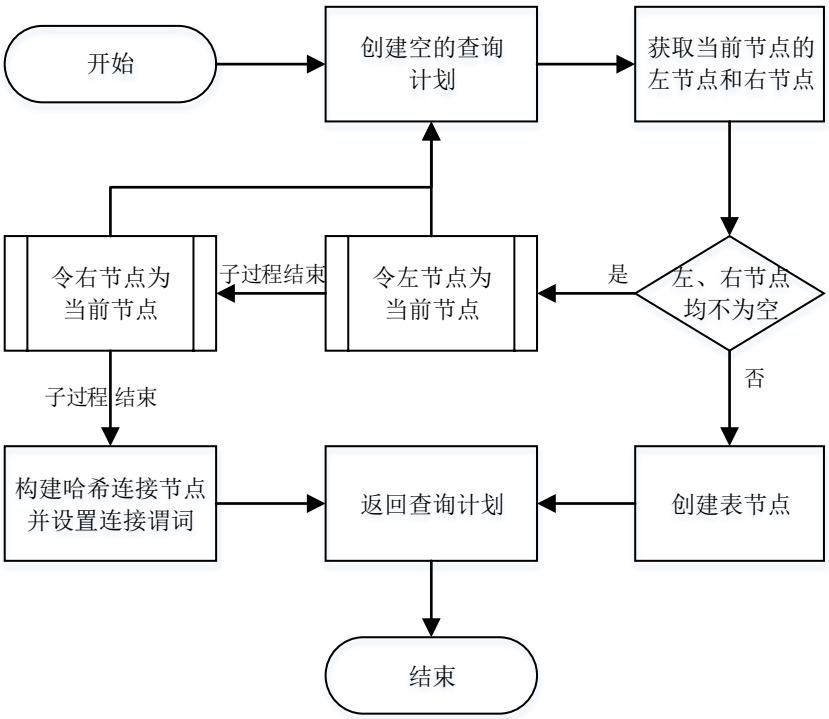

图 5.6 查询计划构造流程图

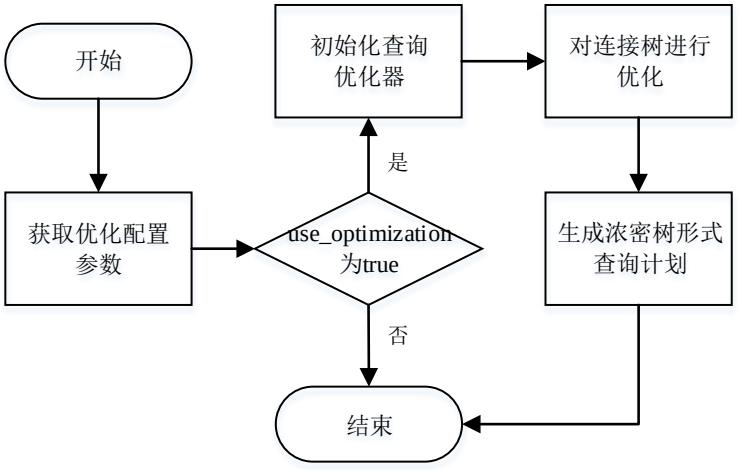

图 5.7 查询计划创建流程图

首先通过配置文件获取优化配置参数，本方法提供可配置的优化方案，可以选择优化的方法、设置代价模型相关参数等等，其中 use_optimization 为是否进行查询优化的开关，只有设置为 true 时才会进行查询优化。当

use_optimization 为 true 时，初始化查询优化器 TDEnumerator，接着调用其 optimize 方法对连接树进行优化。最后使用优化后的连接树生成浓密树形式的查询计划，该过程已在 5.2.4 小节进行了介绍。

2) 查询优化器 TDEnumerator 的 optimize 方法

增加 TDEnumerator 的 public JoinTree optimize(JoinTree joinTree, boolean usePruning) throws NotImplementedException 方法对查询进行优化，流程如图 5.8 所示。

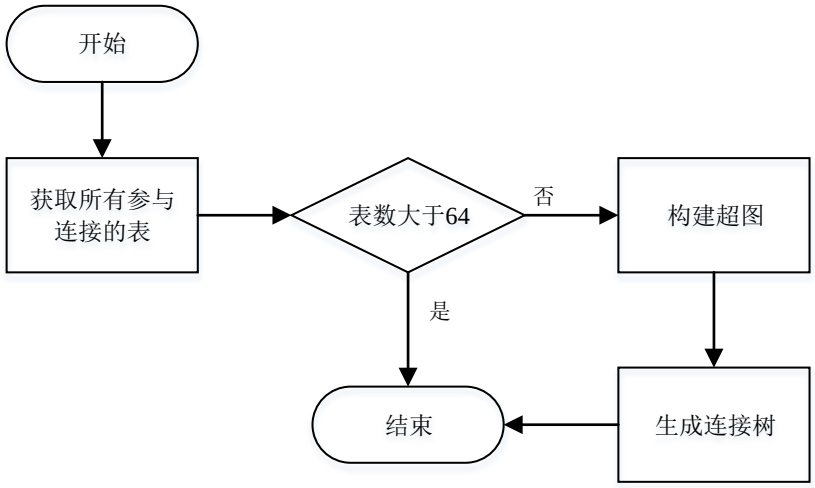

图 5.8 连接树优化流程图

首先获取所有参与连接的表，接着判断表的数量是否大于 64，若是，则抛出 NotImplementedException 并结束。由于在进行超图优化时是用数值的每一位来表示一个表，系统中使用 long 类型进行表示，long 类型在 Java 中是 64 位，因此最多只支持对 64 个不同表的连接操作进行优化，可以满足大部分的查询请求。当表数不大于 64 时，会按 4.3 节给出的超图建模方法构建超图。最后使用构建的超图生成连接树，优化算法主要在这个过程中进行，优化算法的执行流程如图 5.9 所示。

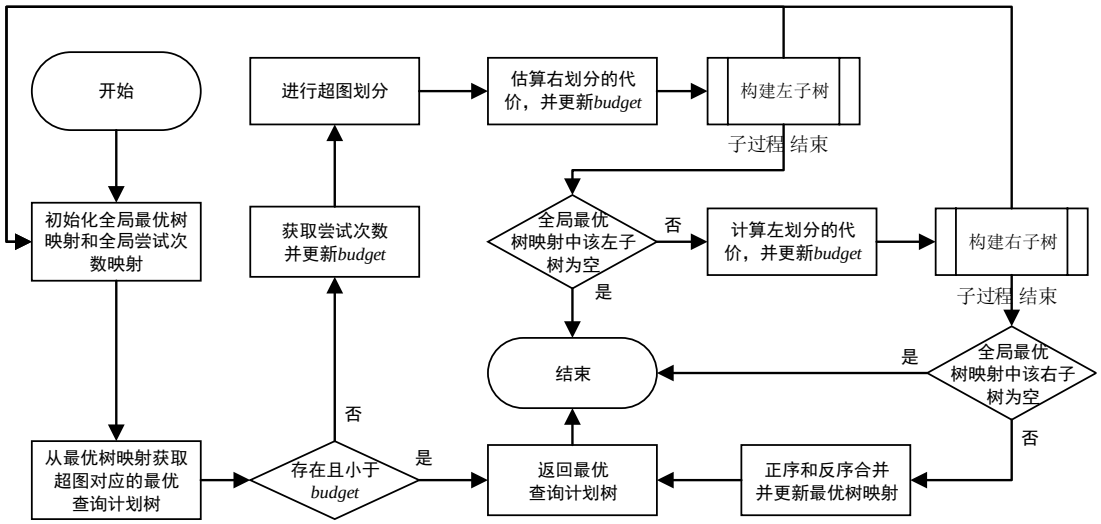

图 5.9 优化算法执行流程图

首先初始化全局最优树映射和全局尝试次数映射，将单一表对应的单节点查询计划树放入全局最优树映射中；接着从最优树映射获取该超图对应的最优查询计划树，并判断其是否存在且小于给定的代价 *budget*，若存在且代价小于 *budget*，则直接返回该查询计划树并结束。由于程序是第一遍执行，此时映射中并不存在超图对应的最优查询计划树，因此将继续执行。从全局尝试次数集合中获取该超图的尝试次数并以此更新 *budget*，接着对超图进行划分，生成多组不同的左划分和右划分对。

对每一组划分，首先估算右划分的代价，并将 *budget* 设为 *budget* 与右划分代价的差值，使用更新后的 *budget* 递归构建左子树。当递归过程完成后，判断全局最优树映射中该左子树对应的查询计划树是否为空，若为空，说明左子树构建失败，直接结束，否则计算左划分的代价并将 *budget* 设为 *budget* 与左划分代价的差值。接着使用更新后的 *budget* 递归构建右子树。当递归过程完成后，判断全局最优树映射中该右子树对应的查询计划树是否为空，若为空，说明右子树构建失败，直接结束，否则正序和反序合并左右子树的查询计划树，将代价最低的作为最优查询计划树，并将全局最优树映射更新后返回最优的查询计划树。

5.4 本章小结

本章主要介绍了系统实现，首先给出了集成改进的 McCHyp 算法的 Impala 总体架构，接着介绍了查询计划树形式的修改方法，对其中几个重要的数据结构进行了说明。然后介绍了构造查询计划的方法，使用递归的方式自顶向下构建查询计划。最后介绍了改进的 McCHyp 算法在 Impala 中集成的方法，可通过配置文件选择优化的方法以及配置代价模型相关的参数。

第6章 实验评估

6.1 实验环境

本实验在 Cloudera 集群和大数据实时查询系统 Impala 上进行，Cloudera 集群有 13 个节点，系统配置如表 6.1 所示。

表 6.1 集群系统配置

Hardware / Software	Info
CPU	4 core, 1.6GHz
Memory	32GB
Disk	2TB
Network	1000Mbit/s
Operating System	Ubuntu 12.04 Desktop 64bit
Cloudera CDH	5.2.0-1
Impala	2.0.0-1

本实验使用 Cloudera Manager 部署 Hadoop 集群，并基于 Impala 2.0 进行修改，表 6.2 展示了各节点运行的服务。

表 6.2 系统各节点运行的服务

Node	Services
pc1	NameNode JobTracker
pc2	Secondeary NameNode Hive Metastore Statestored Catalogd
pc3~pc5	Zookeeper
pc6 ~pc13	DataNode TaskTracker Impalad

6.2 实验设置

本实验分为 4 部分：

1) 比较优化算法的优化效率，即比较 McCHyp 算法和集成剪枝策略的改进的 McCHyp 算法；
2) 比较不同代价模型对优化效果的影响，即比较本文提出的代价模型 C_{improved} 和另外两种代价模型 C_{ca}、C_{opt};
3) 比较集成不同优化算法的 Impala 的查询性能，即比较原始 Impala 系统、集成改进的 DPhyp 算法的 Impala 系统和集成改进的 McCHyp 算法的 Impala 系统；
4) 比较 2 种 Impala 系统的可扩展性，即比较原始 Impala 系统和集成改进的 McCHyp 的 Impala 系统当集群节点数变化时对查询性能的影响。

6.3 实验数据

本实验采用 TPC-DS 数据集，其由 25 个基本表和 99 条查询语句组成，该测试集可以生成不同规格的数据。根据集群环境和 Impala 的特性，本实验使用 5 种规格的数据：20GB、40GB、60GB、80GB 和 100GB，数据表名称和对应行数如表 6.3 所示。

表 6.3 TPC-DS 数据表

Table Name	Number of Tuples				
	20GB	40GB	60GB	80GB	100GB
call_center	6	8	8	10	30
catalog_page	1×10^4	1×10^4	1×10^4	1×10^4	2×10^4
catalog_returns	288×10^4	576×10^4	864×10^4	1152×10^4	1440×10^4
catalog_sales	2880×10^4	5760×10^4	8640×10^4	11520×10^4	14400×10^4
customer	27×10^4	60×10^4	93×10^4	127×10^4	200×10^4
customer_address	13×10^4	30×10^4	47×10^4	63×10^4	100×10^4
customer _demographics	192×10^4	192×10^4	192×10^4	192×10^4	192×10^4
date_dim	7×10^4	7×10^4	7×10^4	7×10^4	7×10^4
dbgen_version	1	1	1	1	1
household _demographics	7200	7200	7200	7200	7200

续表 6.3

Table Name	Number of Tuples				
	20GB	40GB	60GB	80GB	100GB
income_band	20	20	20	20	20
inventory	1827×10^4	4072×10^4	6760×10^4	10022×10^4	39933×10^4
item	3×10^4	5×10^4	7×10^4	10×10^4	20×10^4
promotion	355	466	577	688	1000
reason	36	38	40	42	55
ship_mode	20	20	20	20	20
store	44	112	178	244	402
store_returns	576×10^4	1151×10^4	1727×10^4	2303×10^4	2880×10^4
store_sales	5760×10^4	11520×10^4	17280×10^4	23039×10^4	28800×10^4
time_dim	9×10^4	9×10^4	9×10^4	9×10^4	9×10^4
warehouse	5	6	7	8	15
web_page	264	672	1082	1490	2040
web_returns	144×10^4	288×10^4	432×10^4	576×10^4	720×10^4
web_sales	1440×10^4	2880×10^4	4320×10^4	5760×10^4	7200×10^4
web_site	28	24	20	16	24

6.4 实验结果与分析

6.4.1 优化算法对比

为验证集成剪枝策略的 McCHyp 算法的有效性，我们比较了 McCHyp 算法和改进的 McCHyp 算法在对 2~28 张表进行优化时耗费的时间。同时，我们在链式、星型和随机无环三种不同的连接类型下做了对比实验，算法使用本文提出的代价模型，实验结果如图 6.1~图 6.3 所示。

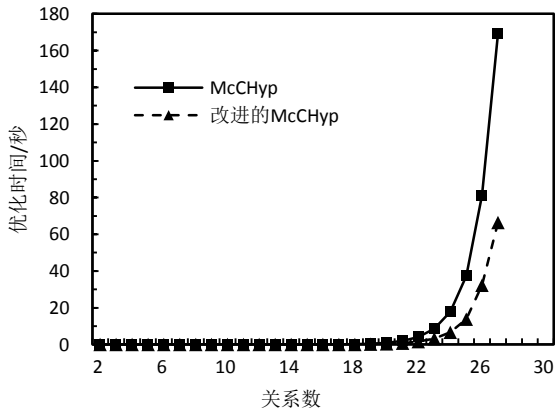

图 6.1 链式连接优化时间

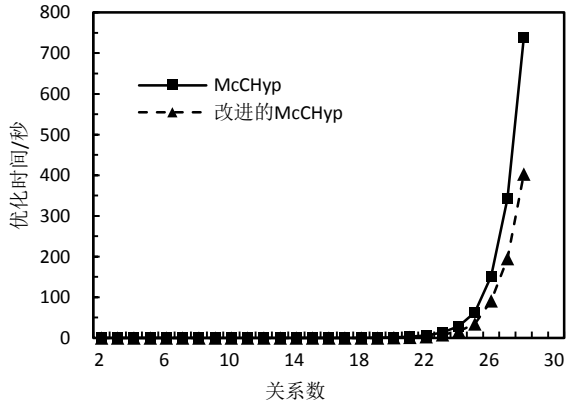

图 6.2 星型连接优化时间

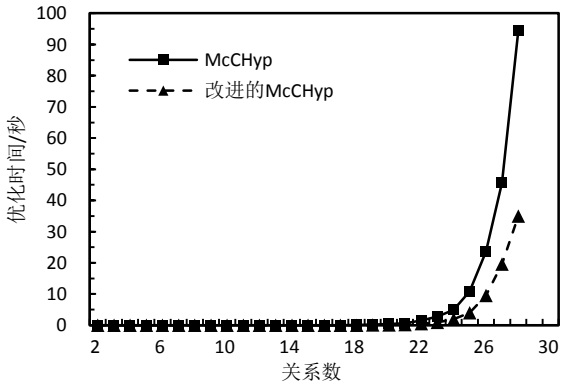

图 6.3 随机无环连接优化时间

从图 6.1 可以看出，当关系数小于等于 20 时，2 个算法的优化时间均在 1 秒以内；当关系数大于 20 时，优化时间随关系数增加大幅增长。改进的 McCHyp 算法的优化时间比原算法平均减少 61.65%。图 6.2 是星型连接下算法的优化时间，可以看到当关系数小于等于 19 时，2 个算法的优化时间均在 1 秒以内；当关系数大于 19 时，优化时间随关系数增加大幅增长。改进的 McCHyp 算法的优化时间比原算法平均减少 43.82%。图 6.3 是随机无环连接下算法的优化时间，可以看到当关系数小于等于 21 时，2 个算法的优化时间均在 1 秒以内；当关系数大于 21 时，优化时间随关系数增加大幅增长。改进的 McCHyp 算法的优化时间比原算法平均减少 62.55%。

由于链式连接在进行超图划分时会有更多的划分，而星型连接只有很少的划分，因此剪枝策略对于链式连接有更好的优化效果。结果显示使用相同的代价模型，改进的 McCHyp 算法比原算法的优化时间减少了 43.82%~62.55%。

6.4.2 代价模型对比

参数调优是一个复杂的问题，需要理论的支持和大量的实验，本文在对代价模型进行对比时仅使用默认的参数（$\alpha_L = \alpha_R = 0.3$，$\beta = \delta = 0.1$，$\gamma$=0.2）。

为评估本文提出的代价模型 C_{improved}，我们在 2~28 张表的随机无环连接查询上比较了使用 C_{improved}、C_{ca}、C_{opt} 三个代价模型的查询响应时间，结果如图 6.4 所示。

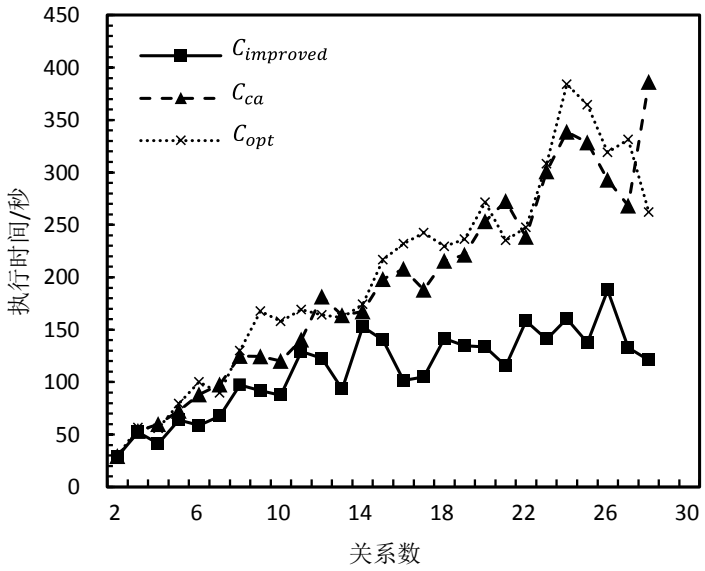

图 6.4 不同代价模型下的查询响应时间

由于 C_{ca} 和 C_{opt} 均为对左深树形式的查询计划进行优化，且优化结果通常是按表行数从大到小排列，两者在查询响应时间上有相似的优化结果。而 C_{improved} 在大多数情况下有更好的表现，本文提出的代价模型不仅保证按表行数从大到小的排列，同时可以生成浓密树形式的查询计划，使查询可以并行执行。当关系数较多时，可并行执行的情况也相应增加，优化效果更明显。

另一方面，当更改代价模型的参数时，优化结果也会发生变化，查询响应时间可能会增加或者减少。参数的调整需要考虑集群的环境，比如当集群是百兆网时，可以适当增加β的权重，而当集群是千兆网时，可以适当减少β的权重。

6.4.3 查询性能对比

为评价集成改进的 McCHyp 算法的 Impala 系统的查询性能，我们使用 TPC-DS 数据集在不同数据量（20GB、40GB、60GB、80GB 和 100GB）下进行了测试，比较了 Impala 系统、集成改进的 DPhyp 算法的 Impala 系统和集成改进的 McCHyp 算法的 Impala 系统的查询响应时间，结果如表 6.4 所示。

表 6.4 查询响应时间

		SQL 7	SQL 19	SQL 42	SQL 43	SQL 52	SQL 55	SQL 85	SQL 96	Total
20 GB	Impala	8.48s	6.21s	5.05s	5.10s	4.96s	5.16s	17.06s	3.98s	56.00s
	Impala-DPhyp	1.25x	1.30x	1.00x	1.10x	1.04x	0.96x	1.00x	1.11x	1.09x
	Impala-McCHyp	1.01x	1.30x	0.92x	1.12x	0.94x	0.98x	0.86x	1.07x	0.99x
40 GB	Impala	14.50s	15.65s	9.05s	9.40s	9.48s	8.98s	33.66s	6.89s	107.61s
	Impala-DPhyp	0.97x	0.84x	1.04x	1.00x	0.96x	1.03x	1.00x	1.07x	0.98x
	Impala-McCHyp	0.98x	0.84x	0.98x	0.96x	0.93x	1.00x	0.60x	1.10x	0.85x
60 GB	Impala	19.51s	40.19s	12.68s	13.24s	12.69s	12.70s	41.25s	10.00s	162.26s
	Impala-DPhyp	1.31x	0.40x	1.04x	1.04x	1.00x	1.00x	0.96x	0.93x	0.88x
	Impala-McCHyp	1.10x	0.39x	1.00x	0.96x	0.96x	0.96x	0.52x	0.90x	0.71x
80 GB	Impala	24.52s	52.22s	16.19s	16.72s	16.70s	15.70s	54.11s	12.30s	208.46s
	Impala-DPhyp	1.37x	0.40x	1.03x	0.94x	0.97x	1.00x	0.98x	0.96x	0.88x
	Impala-McCHyp	1.03x	0.41x	1.00x	1.01x	0.97x	1.03x	0.40x	0.92x	0.69x
100 GB	Impala	30.04s	65.24s	20.19s	21.24s	20.22s	20.20s	72.66s	14.30s	264.09s
	Impala-DPhyp	1.37x	0.42x	0.98x	0.96x	0.98x	1.00x	0.97x	1.00x	0.88x
	Impala-McCHyp	1.05x	0.40x	1.00x	0.93x	0.97x	1.00x	0.31x	0.97x	0.65x

表格中 5 种规格数据的 Impala 行是 8 条查询语句在 Impala 2.0 系统中的查询响应时间，而 Impala-DPhyp 行和 Impala-McCHyp 行是集成改进的 DPhyp 算法的

Impala 2.0 系统与集成改进的 McCHyp 算法的 Impala 2.0 系统的查询响应时间与对应语句在 Impala 2.0 系统中的查询响应时间的倍数关系。为保证查询响应时间的准确性，避免因缓存等原因导致的性能差异，本实验关闭了 HDFS 与 Impala 系统的缓存设置。

查询语句和参与连接操作的表信息见表 6.5。

表 6.5 查询语句与表信息

Query	Table Nums.
SQL 7	5
SQL 19	6
SQL 42	3
SQL 43	3
SQL 52	3
SQL 55	3
SQL 85	8
SQL 96	4

首先比较 Impala 和集成改进的 McCHyp 算法的 Impala 在 7 条查询语句（除 SQL 85）下的查询响应时间。对于只有 3 个表的查询（如 SQL 42、SQL 43、SQL 52 和 SQL 55），2 种系统均对语句进行了优化，并生成左深树形式的查询计划，查询时间基本相同。虽然 SQL 7 和 SQL 96 是多表（≥4）连接的查询操作，但其为星型连接，候选的超图划分数量少，通常只能生成左深树形式的最优查询计划，与在 Impala 系统下的查询响应时间基本相同。SQL 19 是 6 表连接的查询语句，当数据量较大时，集成改进的 McCHyp 算法的 Impala 系统的查询响应时间比原始的 Impala 系统减少了 60%。

接着比较集成改进的 DPhyp 算法的 Impala 和集成改进的 McCHyp 算法的 Impala 在 7 条查询语句（除 SQL 85）下的查询响应时间。从表中可以看出， 2 者的查询响应时间基本相同，其中 SQL 19 是 6 表连接的查询语句，其最优查询计划是左深树形式的，2 者的查询响应时间基本相同且均优于原始的 Impala 系统，说明两者均可获得最优的左深树形式的查询计划。

SQL 85 是 8 表连接查询语句，其在 3 个系统的查询响应时间如图 6.5 所示。

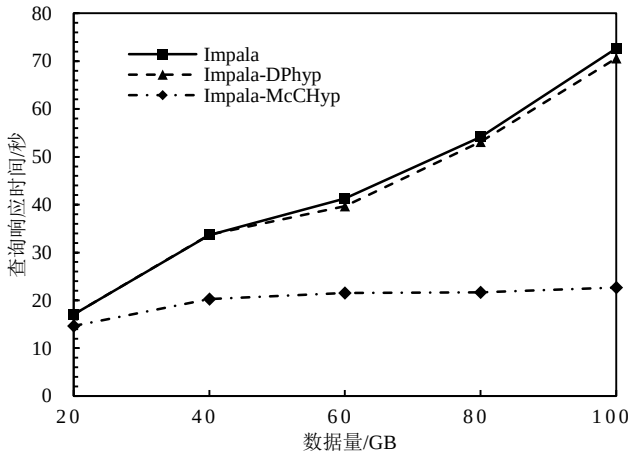

图 6.5 不同数据量下 SQL 85 的查询响应时间

从图中可以看出，随着数据量的增大，3 者的查询响应时间均增大，其中 Impala 系统和集成改进的 DPhyp 算法的 Impala 系统的查询响应时间相似，增长幅度较大，且后者的查询性能略优于前者。集成改进的 McCHyp 算法的 Impala 系统由于生成的最优查询计划是浓密树形式的，可以并行执行，查询响应时间优于另外 2 个系统，且随着数据量的增大，查询响应时间的增长幅度较小。集成改进的 McCHyp 的 Impala 系统查询响应时间比 Impala 系统和集成改进的 DPhyp 算法的 Impala 系统分别减少了 14.24%~68.81%和 14.24%~67.89%。

图 6.6 显示了不同数据量下 8 条语句的总查询响应时间，可以看出，集成改进的 McCHyp 算法的 Impala 系统具有最优的查询性能，且随着数据量的增大，三者的性能差距越明显。集成改进的 McCHyp 的 Impala 系统总查询响应时间比 Impala 系统和集成改进的 DPhyp 算法的 Impala 系统分别减少了 0.95%~34.66%和 8.96%~26.02%。

6.4.4 可扩展性对比

为了验证基于浓密树形式的查询计划随着集群节点数的增加查询性能会更好，本小节在 100GB 的数据集上对 Impala 和集成改进的 McCHyp 算法的 Impala

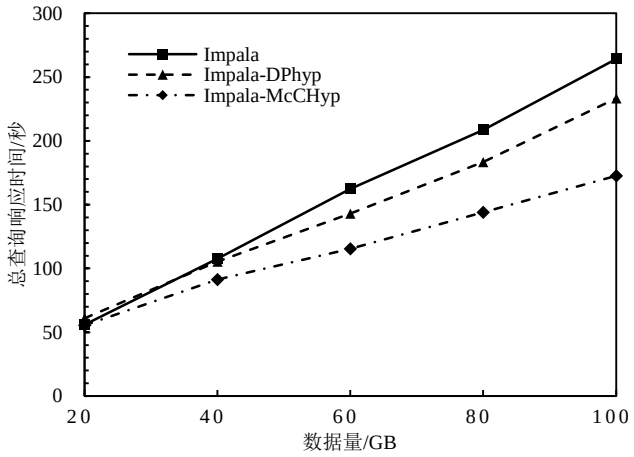

图 6.6 不同数据量下总查询响应时间

在 1 节点、3 节点、5 节点和 7 节点的集群上的查询响应时间进行了比较。由于 SQL 85 是有 8 张表参与连接的查询语句，本实验使用此查询语句进行评估，实验结果如图 6.7 所示。

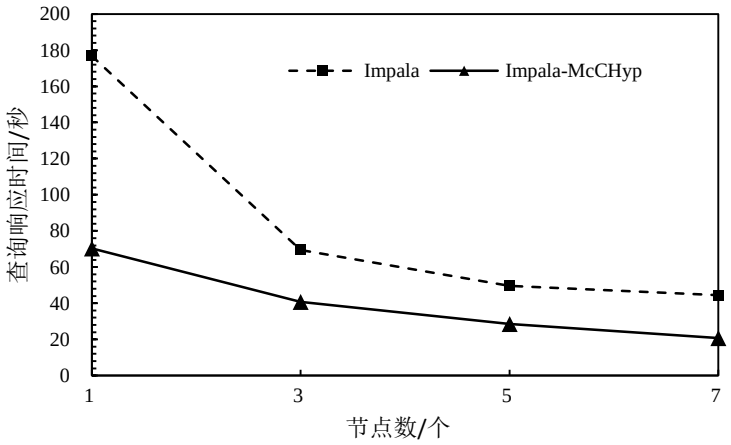

图 6.7 不同节点数下 SQL 85 的查询响应时间

对于左深树形式的查询计划，当集群节点数从 1 个增加到 3 个时，Impala 的查询响应时间减少了 60.70%；当集群节点数从 3 个再继续增加时，查询响应时间减少的幅度逐渐减小（3→5：28.64%，5→7：10.43%）。由于 Impala 使用 HDFS

作为存储引擎，默认有 3 个副本，当节点数较多时，3 个副本会分配在不同的机器上。当只有 1 个节点时，查询依赖的数据块都存在于该节点上，而当节点数大于等于 3 时，数据块会分布在不同的节点上，在 Scan node 进行扫描操作时可以并行执行，因此查询响应时间会相应减少。对于较大数据量的数据集，其数据块较多，磁盘 I/O 是查询的主要瓶颈，因此增加节点数可以有效提高查询的效率，但由于连接操作是串行执行的，节点继续增加后磁盘 I/O 不再是主要性能瓶颈，因此无法获得较大幅度的性能提升。

对于浓密树形式的查询计划，集成改进的 McCHyp 算法的 Impala 的查询响应时间随集群节点数的增加而减少。当节点数从 1 个增加到 3 个时，SQL 85 的查询响应时间减少了 41.94%；当集群节点数从 3 个再继续增加时，查询响应时间减少的幅度逐渐减小，但小于 Impala 系统（3→5：30.01%，5→7：27.30%）。对基于浓密树形式的查询计划的 Impala 系统，不仅可以并行读取数据块，连接操作也可以并行执行。同时，当查询中参与连接的表较多时，增加集群的节点数可以明显减少查询响应时间。因此，基于浓密树形式的查询计划在增加集群节点数时，查询性能的提升效果会更好。

6.5 本章小结

本章主要对本文提出的改进的优化算法和代价模型进行实验评估，首先介绍了实验的环境，使用 13 台机器和 Cloudera Manager 部署集群；接着介绍了实验的设置，从优化算法、代价模型、查询性能和可扩展性 4 个方面进行实验和比较；然后介绍了实验使用的数据，使用 TPC-DS 数据集的 5 种不同数据量的数据；最后进行了实验和分析，实验结果表明本文提出的改进的优化算法和代价模型有更好的查询性能和可扩展性。

第7章 总结与展望

7.1 总结

大数据是近几年比较热门的话题，大数据相关的企业级产品和开源项目层出不穷，其中 Impala 是大数据实时查询领域中比较有影响力的开源项目，已在多家国内外知名企业部署和应用。本文针对大数据实时查询平台 Impala 对多表连接查询优化的相关问题，提出了改进的 McCHyp 算法。首先对查询优化的相关研究背景进行了介绍，包括搜索空间、搜索策略、和代价模型，并分析了目前研究存在的问题；接着对 Impala 大数据实时查询系统的两个主要发行版本进行了介绍，并分别分析了 Impala 1.0 系统的查询过程和 Impala 2.0 系统对查询过程的改进；然后介绍了统计信息的收集方法和代价估计方法，并提出了一种适用于支持浓密树形式查询计划的分布式实时查询系统的代价模型。

本文的重点是提出一种基于超图和浓密树的大数据实时查询优化方法，首先介绍基本概念和查询超图建模的方法；接着提出使用剪枝策略改进的 McCHyp 算法并通过具体示例解释了算法的执行过程；最后对剪枝策略的完整性和正确性进行了说明。在系统实现部分，给出了系统的整体框架以及 Impala 中查询计划形式的修改方法，最后介绍了如何将改进的 McCHyp 算法集成到 Impala 中。

为了验证本文提出算法，使用了 TPC-DS 数据集从优化算法、代价模型、查询性能和可扩展性 4 个方面对系统进行了对比实验，实验结果表明了集成改进的 McCHyp 算法的 Impala 在链式、星型和随机无环连接情况下均可提高优化的效率，且查询性能优于原始的 Impala 系统和集成改进的 DPhyp 算法的 Impala 系统，同时系统也具有较好的可扩展性，可通过增加集群节点数提高查询的效率。

本文的主要贡献如下：

1) 提出适用于支持浓密树形式查询计划的分布式实时查询系统的代价模型；
2) 提出了集成剪枝策略的改进的 McCHyp 算法；
3) 修改 Impala 系统使其支持浓密树形式的查询计划，并将改进的 McCHyp

算法集成到 Impala 系统中，以支持多表连接查询的优化，提高了查询的效率。

7.2 展望

本文提出了改进的 McCHyp 算法和相应的代价模型并集成到 Impala 2.0 系统中，并通过实验验证了其有效性和高效性。本文的工作还有进一步优化的空间，主要有以下几个方面：

1) 改进的 McCHyp 算法未对超图划分过程进行优化，超图划分过程使用集合操作进行，过程较复杂，还有优化的空间。同时，剪枝策略还可进一步优化，以加速查询优化的过程，减少对非最优解的递归调用过程的次数；
2) 代价模型目前只使用了表的行数和字节数等信息，还可增加其它统计信息，如数据分布直方图，以使计算出的代价更加精确，获得更好的查询优化结果。同时，代价模型之间的参数可通过理论和实验进行调优，得到更适合当前集群环境的参数值；
3) 查询的优化过程包括查询计划的优化和执行计划的优化，本文的工作主要在查询计划的优化，而并未涉及执行计划的优化。先进行查询计划优化，再进行执行计划优化的方式在研究领域是比较普遍的，这种方式在设计和实现上都较为容易，但由于要遍历两个搜索空间，构建两次计划，通常效率较低。对于实时查询系统，如 Impala，可以考虑将 2 者的优化结合，在一个过程中进行，提高查询优化的效率。

参考文献

[1] T. Ibaraki and T. Kameda. On the optimal nesting order for computing N-relational joins. TODS, 1984, 9(3): 482-502

[2] S. Ghemawat, H. Gobioff, and Leung S T. The Google file system. SIGOPS, 2003, 37(5): 29-43

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

[4] K. Shvachko, H. Kuang, S. Radia, et al. The Hadoop distributed file system. MSST, 2010 IEEE 26th Symposium on. IEEE, 2010: 1-10

[5] A. Thusoo, J. S. Sarma, N. Jain, et al. Hive: a warehousing solution over a map-reduce framework. PVLDB, 2009, 2(2): 1626-1629

[6] S. Melnik, A. Gubarev, J. J. Long, et al. Dremel: interactive analysis of web-scale datasets. PVLDB, 2010, 3(1-2): 330-339

[7] Cloudera, Inc. The Leading Open Source Analytic Database for Apache Hadoop. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html

[8] Cloudera, Inc. Cloudera Impala: Real-Time Queries in Apache Hadoop. http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/

[9] Cloudera, Inc. Scalability Considerations for Impala. http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/latest/topics/impala_scalability.html#spill_to_disk_unique_1

[10] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. JVLDB, 1997, 6(3): 191-208

[11] K. Ono, and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. VLDB, 1990: 314-325

[12] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, et al. Access path selection in a relational database management system. SIGMOD, 1979: 23-34

[13] A. Swami and A. Gupta. Optimization of large join queries. SIGMOD, 1988: 8-17

[14] A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. SIGMOD, 1989: 367-376

[15] R. S. G. Lanzelotte, P. Valduriez, and M. Za ŧ. On the effectiveness of optimization search strategies for parallel execution spaces. VLDB, 1993: 493-504

[16] D. DeHaan and F. W. Tompa. Optimal top-down join enumeration (extended version). University of Waterloo, Tech. Rep., 2007

[17] P. Fender and G. Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. ICDE, 2011: 864-875

[18] S. Katsoulis. Implementation of parallel hash join algorithms over Hadoop. Master of Science. School of Information University of Edinburgh, 2011

[19] S. Wu, F. Li, S. Mehrotra, et al. Query optimization for massively parallel data processing. SOCC, 2011: 1-13

[20] Implement optimizer support for bushy trees. https://issues.apache.org/jira/browse/DERBY-4327?page=com.atlassian.jira.plugin.system.issuetabpanels:comment-tabpanel

[21] G. Goetz. The cascades framework for query optimization. IEEE Data Eng. Bull., 1995, 18(3): 19-29

[22] M. Stonebraker, G. Held, E. Wong, et al. The design and implementation of INGRES. TODS, 1976, 1(3): 189-222

[23] B. Vance and D. Maier. Rapid bushy join-order optimization with Cartesian products. SIGMOD, 1996: 35-46

[24] 江贺. 超启发式算法: 跨领域的问题求解模式. 中国计算机学会通讯, 2011: 63-70

[25] P. Fender and G. Meorkotte. Reassessing top-down join enumeration. TKDE, 2012, 24(10): 1803-1818

[26] P. Fender, G. Moerkotte, T. Neumann, et al. Effective and robust pruning for top-down join enumeration algorithms. ICDE, 2012: 414-425

[27] P. Fender and G. Moerkotte. Top down plan generation: From theory to practice.

ICDE, 2013: 1105-1116

[28] G. Moerkotte and T. Neumann. Dynamic programming strikes back. SIGMOD, 2008: 539-552

[29] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. VLDB, 2006: 930-941

[30] B. Vance. Join-order optimization with Cartesian products. PhD thesis, Oregon Graduate Institute of Science and Technology, 1998

[31] T. Neumann. Query simplification: graceful degradation for join-order optimization. SIGMOD, 2009: 403-414

[32] A. Nica. A call for order in search space generation process of query optimization. ICDEW, 2011: 4-9

[33] KESSLER Software GmbH & Co KG. SQL Anywhere 16. http://www.kessler.de/prd/sybase/sqlanywhere.htm

[34] 赵鹏, 王守军, 龚云. 基于改进蚁群算法的数据仓库多连接查询优化. 计算机工程, 2012: 168-173

[35] T. Dökeroğlu and A. Coşar. Dynamic programming with ant colony optimization metaheuristic for optimization of distributed database queries. ISCIS, 2012: 107-113

[36] L. Golshanara, S. M. T. R. Rankoohi, and H. Shah-Hosseini. A multi-colony ant algorithm for optimizing join queries in distributed database systems. KAIS, 2013: 1-32

[37] E. Sevinç and A. Coşar. An evolutionary genetic algorithm for optimization of distributed database queries. Comput. J., 2011, 54(5): 717-725

[38] T. Dokeroglu, U. Tosun, and A. Cosar. Particle swarm intelligence as a new heuristic for the optimization of distributed database queries. AICT, 2012: 1-7

[39] 谢晓峰, 张文俊, 杨之廉. 微粒群算法综述. 控制与决策, 2003: 129-134

[40] G. M. Lohman, C. Mohan, L. M. Haas, et al. Query processing in R*. Springer Berlin Heideberg, 1985: 31-47

[41] S. Khoshafian and P. Valduriez. Parallel execution strategies for declustered

databases. Springer US, 1988: 458-471

[42] S. Ganguly, A. Goel, and A. Silberschatz, Efficient and accurate cost models for parallel query optimization. PODS, 1996: 172-181

[43] 周强，陈岭，马骄阳，赵宇亮，吴勇，王敬昌. 基于改进 DPhyp 算法的 Impala 查询优化. 计算机研究与发展, 2013, 50(增刊): 114-120

[44] Cloudera, Inc. Releases. https://github.com/cloudera/Impala/releases

[45] Cloudera, Inc. SELECT Statement. http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/latest/topics/impala_select.html

[46] Cloudera, Inc. New Features in Impala. http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/latest/topics/impala_new_features.html

[47] Gerwin Klein. JFlex. http://www.jflex.de

[48] Technische Universität München. CUP. http://www2.cs.tum.edu/projects/cup/

[49] Elliot Berk. JLex: A Lexical Analyzer Generator for Java(TM). http://www.cs.princeton.edu/~appel/modern/java/JLex/

[50] Tomas Hurka. BYACC/J. http://byaccj.sourceforge.net/

[51] Terence Parr. ANTLR. http://www.antlr.org/

[52] Cloudera, Inc. Components of the Impala Server. http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/latest/topics/impala_components.html#intro_catalogd_unique_1

[53] Cloudera, Inc. Performance Considerations for Join Queries. http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/latest/topics/impala_perf_joins.html#perf_joins_examples_unique_1

攻读硕士学位期间主要的研究成果

[1] 浙江大学, 基于浓密树和自顶向下的大数据实时查询优化方法: 中国, 专利号: 201410765313.6, 2014.12.15

[2] 马骄阳, 陈岭, 赵宇亮, 杨谊, 吴勇, 王敬昌. 基于浓密树和改进 McCHyp 算法的 Impala 查询优化. 计算机研究与发展(增刊, 中国计算机学会第 2 届大数据学术会议), 2014

致谢

至此，论文已接近完成，研究生的学习生涯也接近尾声，在此，对曾经给予我指导和帮助的导师、实验室的小伙伴们以及父母表示感谢。

陈岭导师工作认真细致，关心学生的发展，每周都有例会检查学生们的工作情况并且与大家讨论研究或是项目上的难点，使我们每周都有新的收获。每个月还有博士生研讨会，让我们除了自己的研究领域，还能接触到其他领域的研究成果，开阔了我们的眼界。陈老师对工作的认真态度对我有很深的影响，从代码的注释到文档的写作，都学到了很多，也养成了很好的习惯。记得曾经某个劳动节，陈老师放弃休息带着我们几个学生一起撰写课题文档，并一字一句的更正，使我现在写文档时也很注意英文字体、行距、空格等容易忽视的问题。还要感谢陈老师给我机会让我当上了实验室查询处理组的组长，通过两年多的组长工作，不仅开发能力得到了提高，在项目管理能力以及与组员的沟通能力上也有了很大的进步。再一次对陈岭导师表示感谢。

接下来要感谢实验室的全体师兄师姐师弟师妹们（肖敏，涂鼎，王礼文，郭敬，Ibrar Hussain，邵维，徐振兴，袁翠丽，范阿琳，Memon Imran，董苗淼，彭梁英，赵宇亮，沈延斌，蔡雅雅，余小康，杨谊，钱坤，郭浩东，王俊凯，顾伟东，韩保礼，林言，吴晓杰，应鸳凯，蔡晓东，莫树聪，孔星，何旭峰，侯仓键，周强，范长军，涂游，刘荣游，唐燕琳，吕明琪，闯跃龙，李卓豪，刘颖，冯普超，郑君正，朱江烽，金鑫，余斌，韩成，马进，马超，王耀光，郑福真，乔奇，余宙，蔡青林，何颖鹏，雷鹏，吴峻，李旭，秦祎明，白剑飞，毛立花，刘成昊，刘坚，申亚鹏，蒋静远，张芸，陈逸宇，孔德江），和大家一起学习、工作、生活很开心，实验室课题和项目的成功离不开大家的努力，我能写下这篇毕业论文，也得到了大家的帮助和支持。特别要感谢 Impala 查询优化小组的几位同学（周强、赵宇亮、杨谊、林言、王俊凯），从 2012 年 11 月开始研究 Impala 以来，已有两年多的时间，在这段时间里，遇到过难题也遇到过挫折，大家互相鼓励互相帮助，

才有了今天的成果。Impala 的查询优化研究还将继续下去，祝愿师弟师妹们能取得更大的成就。

同时还要感谢我的父母，感谢他们二十多年来对我的养育之恩，千里迢迢来浙大读书也得到了他们的支持。他们在生活上给予了我很大的帮助，关心我、鼓励我，家里老人们身体不好，他们在繁忙的工作之余还要照顾老人。读研以来，每年陪伴他们的时间非常少，毕业之后，我会多陪陪他们，努力报答他们对我的养育之恩。

最后，感谢各位评审老师审阅本文并提出宝贵的意见。

马骄阳

2014-12-28