基于浓密树和改进 McCHyp 算法的 Impala 查询优化

马骄阳[1] 陈 岭[1] 赵宇亮[1] 杨 谊[1] 吴 勇[2] 王敬昌[2]
[1](浙江大学计算机科学与技术学院 杭州 310027)
[2](浙江鸿程计算机系统有限公司 杭州 310009)
(majiaoyang@vip.qq.com)

摘要 针对 Impala 大数据实时查询系统在查询优化上存在的问题，提出基于浓密树和改进的 McCHyp（MinCutConservative Hypergraph）算法的 Impala 查询优化方法。首先，修改 Impala 使其支持浓密树的查询计划；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；最后，提出一种适用于 Impala 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。在 Impala 系统上实现了本文提出的查询优化方法并在 TPC-H 数据集上进行了实验，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%。同时，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 79.60%。

关键词 查询优化；Impala；代价模型；浓密树；查询计划

中图法分类号 TP311

Bushy Tree and Improved-McCHyp Algorithm Based Impala Query Optimization

Ma Jiaoyang[1], Chen Ling[1], Zhao Yuliang[1], Yang Yi[1], Wu Yong[2], and Wang Jingchang[2]
[1] (College of Computer Science and Technology, Zhejiang University, Hangzhou 310027)
[2] (Zhejiang Hongcheng Computer Systems Co., Ltd., Hangzhou 310009)

Abstract As the real-time big-data query system Impala has problem with query optimization, we proposed a bushy-tree and Improved-McCHyp (Improved-MinCutConservative Hypergraph) algorithm based Impala query optimization method. The method firstly modified Impala to support bushy-tree query plans. Then improving McCHyp algorithm with pruning strategy to reduce query optimization time. Finally, we proposed a new cost model, and integrated Improved-McCHyp algorithm into Impala to generate better query plans with user's SQL statement. The query optimization method is implemented in Impala and evaluated using TPC-H, 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%. Also, the processing time of the query, which was optimized by Improved-McCHyp algorithm and the new cost model, decreased by 79.60%.

Keywords query optimization; Impala; cost model; bushy tree; query plan

随着大数据时代的到来，对海量数据的快速查询处理成为互联网、电信、金融等类型企业的迫切需求。为了满足这类需求，大数据实时查询系统应运而生，如 Google Dremel[1]、Berkeley Shark[2]和 Cloudera Impala[3]等。Cloudera 公司的 Impala 由于不再使用 MapReduce 批处理，而是使用一种分布式查询引擎，直接从 HDFS（Hadoop Distributed File System）或 HBase 中获取数据进行查询处理，从而大幅度减少了查询时间，其查询响应速度是分布式数据仓库 Hive[4-5]的 3~90 倍[6]。Impala 1.0 版本中的查询优化主要提供

收稿日期：2014-06-19
基金项目："核高基"国家重大科技专项课题（2010ZX01042-002-003）；国家自然科学基金（60703040，61332017）；浙江省重大科技专项（2011C13042，2013C01046）；中国工程科技知识中心资助项目（CKCEST-2014-1-5）
通信作者：陈 岭（lingchen@cs.zju.edu.cn）

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。

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。

搜索空间可以使用查询树进行表示，主要分为左深树（右深树）和浓密树。Steinbrunn 等人[7]对左深树和浓密树的搜索空间进行了分析，得出 n 个单一关系的左深树有 $n!$ 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。20 世纪 90 年代之前，研究人员主要关注基于左深树的查询优化，因为当时的数据库产品大多是单机的，左深树的查询计划只能串行执行，且搜索空间小，优化时间短。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，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有许多尝试。Katsoulis[8]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。Wu 等人[9]提出 AQUA（Automatic Query Analyzer）2 阶段查询优化方法，可在基于 MapReduce 的 MPP（Massively Parallel Processing）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 MPP 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[10]中提到了支持浓密树形式的查询优化，但至今仍未实现。

搜索策略主要分 2 类：一类是自顶向下的，主要思想是由整体到局部进行优化。如 DeHaan 等人[11]提出的关于连接图的自顶向下多表连接生成算法 MinCutLazy，通过与分支定界算法相结合，提高了查询优化的效率。Fender 等人[12]在此基础上对算法进一步优化，提出了 MinCutLazyImp 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[13]还改进了剪枝策略，并提出了一个新的自顶向下算法 MinCutConservative，比 MinCutLazy 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[14]根据 DPhyp（Dynamic Programming Hypergraph）算法[15]的思想，设计了基于超图的自顶向下算法 McCHyp 以支持多元谓词，同时还支持对半连接、外连接等非内连接查询的优化。该算法是目前自顶向下查询优化算法中效率最高的算法，适用于支持浓密树形式查询计划的分布式数据库系统。另一类搜索策略是自底向上的，主要思想是由局部到整体进行优化。如 Moerkotte 等人[15]提出的 DPhyp 算法。其采用超图的思想实现，可以处理复杂的查询谓词，但由于搜索空间较大，又无法有效使用剪枝策略，使得当关系数较多时，优化时间较长。周强等人[16]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。

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执行的代价，使用合适的代价模型可以得到更好的查询优化效果。周强等人[16]提出的代价模型考虑了通信代价和参与连接的表的代价，适用于一般的分布式系统，但并未充分考虑 Impala 的查询执行特性。

针对上述问题，本文提出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 源代码，使其支持浓密树形式的查询计划；其次，在 McCHyp 算法中加入剪枝策略，提高算法的优化效率；最后，结合适用于 Impala 的代价模型，生成最佳的浓密树形式的查询计划。

本文的主要贡献有三点：第一，提出将 Impala 查询计划由左深树改为浓密树的方法；第二，提出一种适用于 Impala 的代价模型；第三，使用剪枝策略对 McCHyp 算法进行改进并集成到 Impala 中，通过对比实验验证了本文提出方法的有效性和高效性。

1 支持浓密树形式查询计划的 Impala

1.1 查询计划的形式

定义 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 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 表示一个单一的关系；若满足：（1）T_L 和 R 均存在，（2）T_L 为单一的关系，或其所有子树中右子树均为单一的关系，则称 T 为左深树。

定义 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，或为单一的关系（T_R 同理），

则称 T 为浓密树。

不难看出，当 T_R 为单一关系，且 T_L 为单一关系，或其所有的右子树也为单一关系时，浓密树 T 也是一棵左深树。

目前，Impala 生成的查询计划是左深树的形式。对于一个多表连接查询 $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$，Impala 生成的查询计划树如图 1 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。

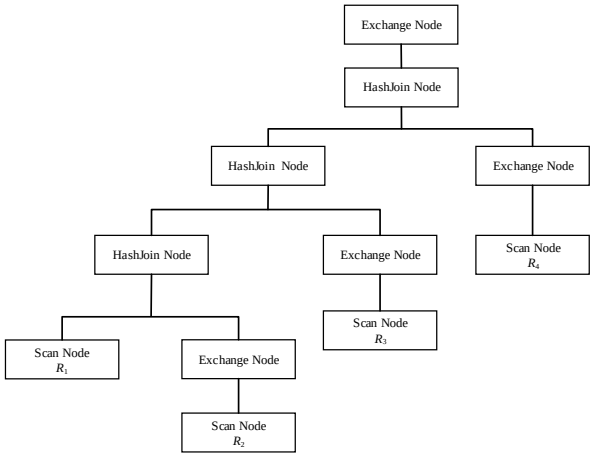


图 1 Impala 查询计划树

1.2 查询计划形式的修改方法

左深树形式的查询计划只能串行执行，对于图 1 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。

可见，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 2 展示了一种浓密树形式的查询计划。与图 1 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。

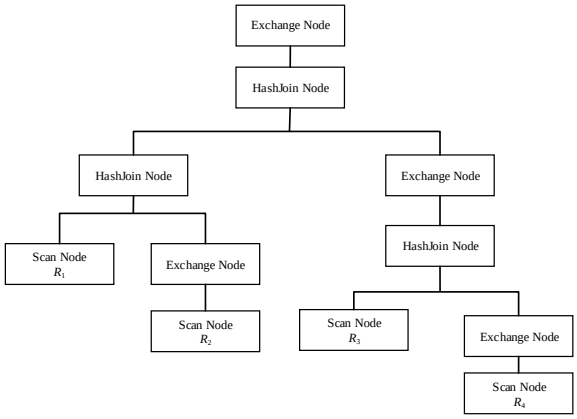


图 2 浓密树形式的查询计划树

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 HashJoinTree，用来保存根节点和查询计划树中所有叶节点（单一关系）的引用；接下来创建一个继承 TableRef 数据结构的 HashJoinNode，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。

修改后的查询计划构造方法执行步骤如下：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。

2 查询优化方法

基于浓密树和改进的 McCHyp 算法的 Impala 查询处理流程如图 3 所示：

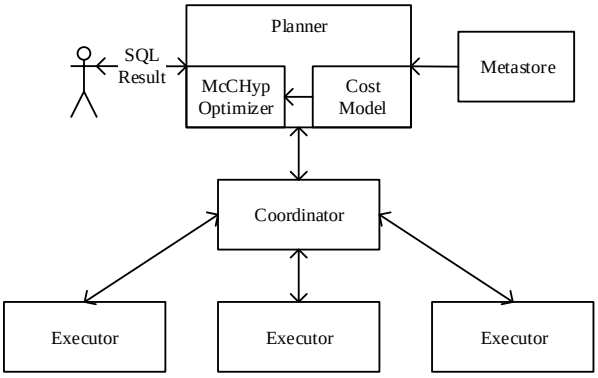


图 3 查询处理流程图

用户将查询请求发送给 Planner；Planner 解析查询语句，并将解析的结果发给查询优化模块：查询优化模块使用改进的 McCHyp 算法，根据表的元数据信

息，使用合理的代价模型，生成浓密树形式的查询计划；Coordinator 根据查询计划生成执行计划，分配执行的节点；各个执行节点上的 Executor 进行具体的查询操作，并将结果发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。

2.1 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，Lohman 等人[17]提出计算代价的通用公式如式 1 所示：

$$T = T_{\text{CPU}} \times n_{\text{insts}} + T_{\text{I/O}} \times n_{\text{I/Os}} + T_{\text{MSG}} \times n_{\text{msgs}} + T_{\text{TR}} \times n_{\text{bytes}}, \quad (1)$$

其中 T_{CPU} 为一条 CPU 指令的时间，n_{insts} 为 CPU 指令的总条数，$T_{\text{I/O}}$ 为一次磁盘读写的时间，$n_{\text{I/Os}}$ 为磁盘读写的总次数，T_{MSG} 为建立一次网络通信的时间，n_{msgs} 为网络通信的总次数，T_{TR} 为通过网络传输一个字节的时间，n_{bytes} 为传输的总字节数。该公式考虑了 CPU、磁盘 I/O、网络通信时间和数据传输时间，基本涵盖了查询涉及的组件，但由于有 8 个变量，计算较复杂，且在一些系统中部分参数无法获取，或计算代价较大，因此在实际使用时会忽略影响较小的参数，或将一些参数设为常量。

周强等人[16]在设计改进的 DPhyp 算法时提出了一种代价模型，如式 2 所示：

$$C_{opt}(i+1) = \max_{j \in [1,n]} \left| T_i \bowtie R_j \right|, \quad (2)$$

其中 $i \in [1, n-1]$，T_{i} 代表已经计算出代价的一棵子树，R_{j} 表示可以与 T_{i} 进行连接且不在 T_{i} 子树中的一个关系。该模型主要考虑数据传输的代价和左右 2 表的大小，使用该代价模型生成的多表连接大致是按表从大到小排列。该模型适用于左深树形式的查询计划，但对于浓密树形式的查询计划，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，代价估计会有偏差。

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 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。除此之外，由于改进后的 Impala 支持浓密树形式的查询计划，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改进后的 Impala 系统，对于一棵查询树 T，其左右子树分别为 L 和 R。当 L 和 R 至少有一个为单一关系时，代价模型如式 3 所示：

$$C_T = \alpha_L IO_L + \alpha_R IO_R + \beta TR_R + \gamma S_R + \delta S_{LR}, \quad (3)$$

其中 $\alpha_L + \alpha_R + \beta + \gamma + \delta = 1$，$IO$ 为磁盘读取的代价，TR 为网络传输代价，S 为数据的大小。当 L 不是单一关系时 $\alpha_L = 0$，当 R 不是单一关系时 $\alpha_R = 0$。当 L 和 R 均不为单一关系时，代价模型如式 4 所示：

$$C_T = \alpha \,\text{MAX}(C_L, C_R) + \beta TR_{\text{R}} + \gamma S_{\text{R}} + \delta S_{\text{LR}}, \quad (4)$$

其中 $\alpha + \beta + \gamma + \delta = 1$。由于 2 子树可以并行执行，因此取两者代价的最大值，同时考虑右子树的网络传输代价、右子树的大小和哈希连接结果的大小。

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。

2.2 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：

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。

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 1 所示：

算法 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。

2.3 剪枝策略的完整性和正确性

定义 4. 连接子图及其补集对。对于一个连接超图 $G = (V, E)$，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）$S_1 \bigcap S_2 = \emptyset$，且

4）$\exists(v_1, v_2) \in E \mid v_1 \in S_1 \wedge v_2 \in S_2$。

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。

2.4 改进的 McCHyp 算法在 Impala 中的集成

McCHyp 算法的输出是查询树，而 Impala 中构造的是查询计划树，为了将改进的 McCHyp 算法与 Impala 集成，需要提供额外的方法将查询树转换为查询计划树。Impala 中 Planner 类的 createSelectPlan 方法主要负责生成查询计划，因此将提供 Optimize 接口，供该方法调用。接口的输入是查询树，输出是查询计划树。Optimize 算法如算法 2 所示：

算法 2. Optimize.

输入：*queryTree*；

输出：*queryPlanTree*。

①*tables* = *queryTree*.getRealTables( )；

②get *vertices* for *tables*；

③get *edges* for *tables*；

④construct *hypergraph* with *vertices* and *edges*；

⑤*tree* = Improved-McCHyp(*hypergraph*)；

⑥*queryPlanTree* = constructTree(*tree*)；

⑦return *queryPlanTree*。

本算法的具体执行过程如下：首先获取查询树的所有基本表，如伪代码行①所示；接下来将这些基本表作为“点”，连接谓词作为“边”构建连接超图，如行②~④所示；然后调用改进的 McCHyp 算法获得优化的查询树，如行⑤所示；最后将查询树转换为查询计划树，并返回，如行⑥⑦所示。使用此方法便可将改进的 McCHyp 算法与 Impala 进行集成。

3 实验及结果分析

3.1 实验环境

本实验在开源 Hadoop 集群和 Impala 大数据实时查询系统上进行，采用 TPC-H 数据集。该数据集由 8 个基本表、22 条 SQL 语句组成，可根据用户需求生成不同数量级的数据，常用于关系型数据库的测试中。本实验使用 1GB 数据集，数据表及对应的表行数如表 1 所示：

表 1 数据表及对应的表行数

表名	表行数	表名	表行数
lineitem	600×10^4	partsupp	80×10^4
orders	150×10^4	region	5
customer	15×10^4	part	20×10^4
nation	25	supplier	1×10^4

Hadoop 集群由 9 个节点组成，硬件配置为：4GB 内存，500GB 硬盘，2.6GHz 双核四线程 CPU。操作系统版本为 Ubuntu 12.04 Desktop 64bit，通过 Cloudera Manager 进行安装部署，CDH 版本为 4.6.0-1。另外，本论文提出的查询优化算法和代价模型将基于 Impala 1.0 进行修改。各节点运行的服务如表 2 所示：

表 2 集群运行的服务

节点	运行的服务
pc1	NameNode
	JobTracker
pc2	Zookeeper
	Hive Metastore
	StateStore
pc3~pc9	DataNode
	TaskTracker
	Impalad

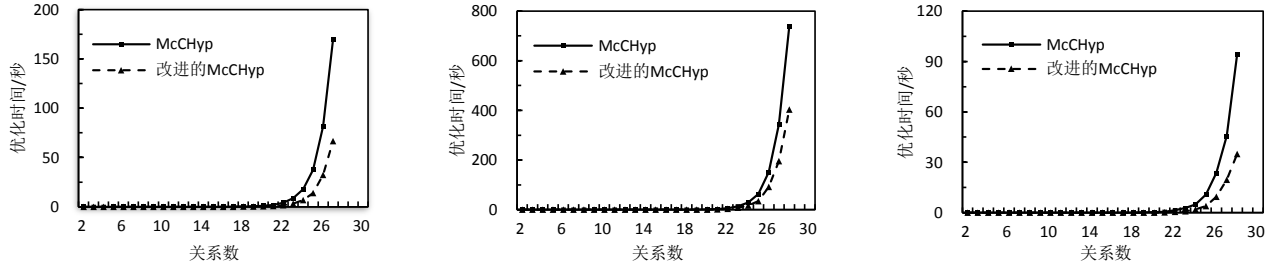
本实验主要分为 2 部分：对优化算法的比较及对集成优化算法的 Impala 的比较。优化算法比较实验主要对比 McCHyp 算法和改进的 McCHyp 算法对 2~28 张表的优化时间；集成优化算法的 Impala 比较实验主要对比原始 Impala 系统、集成改进的 DPhyp 的系统和本文提出的集成改进的 McCHyp 的系统的查询响应时间。

3.2 实验结果及分析

3.2.1 优化算法对比实验结果及分析

为了验证集成剪枝策略的 McCHyp 算法的有效性，本小节将比较 2 种算法对 2~28 张表的优化时间。同时，为了验证不同连接情况下的效果，在链式（chain）、星型（star）和随机无环（random acyclic）连接情况下对优化时间进行了比较。2 种算法采用的代价模型均为 2.1 节提出的新的代价模型。在代价模型参数的选取上，根据系统的特点，选取 $\alpha_L = \alpha_R = 0.3$，$\beta = \delta = 0.1$，$\gamma = 0.2$。实验结果如图 4 所示。

由图 4（a）可以看出，链式连接在关系数小于等



（a）链式连接　　（b）星型连接　　（c）随机无环连接

图 4 优化时间对比图

于 20 时，2 种算法的优化时间均在 1 秒以内；当关系数大于 20 时，随着关系数的增加，优化时间快速增长，改进的 McCHyp 算法的优化时间比原算法平均减少了 61.65%。图 4（b）显示了星型连接时的优化时间，在关系数小于等于 19 时，优化时间均在 1 秒以内；改进的 McCHyp 算法的优化时间平均减少了 43.82%。图 4（c）是随机无环连接时的优化时间，在关系数小于等于 21 时，优化时间均在 1 秒以内；改进的 McCHyp 算法的优化时间平均减少了 62.55%。由于链式连接的超图划分数量较多，而星型连接的超图划分数量较少，剪枝策略对链式连接的优化效果较明显，而对星型连接的优化效果较弱。实验表明，在代价模型相同的情况下，2 种算法生成的查询计划相同，改进的 McCHyp 算法的优化时间相比减少 43.82%~62.55%，更有优势。

3.2.2 集成优化算法的 Impala 对比实验结果及分析

为了验证集成改进的 McCHyp 算法的 Impala 在查询上的优势，本小节使用 TPC-H 数据集对原始 Impala 系统、集成改进的 DPhyp 的系统和集成改进的 McCHyp 的系统在查询响应时间上进行了比较，结果如图 5 所示：

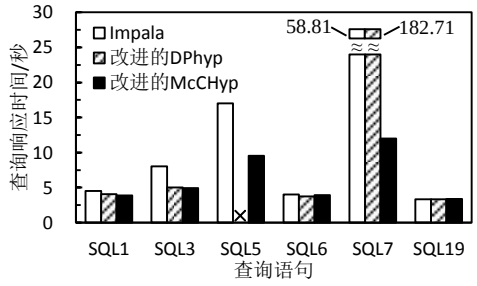


图 5 查询响应时间对比图

查询语句和对应表连接数如表 3 所示。

首先对比原始 Impala 系统和集成改进的 McCHyp 的 Impala 系统在 6 条 TPC-H 查询语句下的查询响应时间。可以看出，对于连接表数为 1 的查询语句，优化算法不进行优化，查询响应时间几乎没有变化，表明优化算法并不会影响未进行优化的语句的查询时间。虽然 SQL19 是 2 表的连接操作，但已满足大表 Join 小表的形式，优化后的查询计划与优化前相同，因此查询时间没有变化。在多表连接情况下，优化效果最明显的是 SQL7。该查询语句带有 6 表连接的 FROM 子句，原始 Impala 系统串行执行，而集成改进的 McCHyp 的 Impala 系统使用浓密树形式的查询计划在多节点上并行执行，因此优化效果明显，查询时间减少了 79.60%。

表 3 查询语句和连接表数

语句	连接表数
SQL1	1
SQL3	3
SQL5	6
SQL6	1
SQL7	6
SQL19	2

接下来比较集成改进的 DPhyp 与集成改进的 McCHyp 的 Impala 系统在 5 条 TPC-H 查询语句下的查询时间（SQL5 前者无法正确执行，在图中以×标示）。可以看出，对于连接表数为 1 和 2 的查询，由于优化结果相同，查询时间没有变化。虽然 SQL3 是 3 表的连接操作，但由于 3 表无法构造浓密树计划，只能是左深树（或右深树）计划，因此两种算法的优化结果相同，查询时间基本一样。优化效果最明显的是 SQL7，查询时间减少了 93.43%。可见除查询时间较优外，改进的 McCHyp 算法的正确率也要高于改进的 DPhyp 算法。

集成改进的 DPhyp 算法的 Impala 在 SQL7 上的查询时间甚至慢于原始 Impala 系统，这主要是由于前者的代价模型在优化时未考虑中间结果的大小，使优化后的查询计划在执行时传输的数据量过多，增加了执行的时间；而本文提出的代价模型由于考虑了中间结果的大小，因此集成改进的 McCHyp 算法的 Impala 在查询执行效率上有很大提升。

4 结束语

针对大数据实时查询平台Impala并未对多表连接查询进行优化的问题，本文提出了改进的 McCHyp 算法。首先修改 Impala 系统使其支持浓密树的查询计划；然后提出一种使用剪枝策略改进的 McCHyp 算法，并与 McCHyp 算法进行了比较，证明了在链式、星型和随机无环连接情况下均可提高优化的效率；最后将改进的 McCHyp 算法及新的代价模型集成到 Impala 中，并与原始系统、集成改进的 DPhyp 算法的 Impala 系统进行了实验对比，表明了本算法的有效性和高效性。本文算法中的代价模型只使用了表的总行数和总字节数 2 个统计信息，如何利用更多的统计信息使优化效果更好，同时控制维护统计信息的代价，是一个值得深入研究的课题。另外，查询优化算法还有改进的空间，将在后续的研究工作中进一步优化。

参 考 文 献

[1] Melnik S, Gubarev A, Long J J, et al. Dremel: Interactive analysis of web-scale databases. Proc of the VLDB Endowment, 2010, 3(1/2): 330-339

[2] Engle C, Lupher A, Xin R, et al. Shark: Fast data analysis using coarse-grained distributed memory //Proc of the 2012 ACM SIGMOD Int Conf on Management of Data. ACM, 2012: 689-692

[3] Cloudera, Inc. Impala. [2014-04-17]. https://github.com/cloudera/impala

[4] Thusoo A, Sarma J S, Jain N, et al. Hive: A warehousing solution over a map-reduce framework. Proc of the VLDB Endowment, 2009, 2(2): 1626-1629

[5] Thusoo A, Sarma J S, Jain N, et al. Hive-a petabyte scale data warehouse using Hadoop //Proc of the 26th Int Conf on Data Engineering. IEEE, 2010: 996-1005

[6] Marcel K, Justin E. Cloudera Impala: Real-time queries in Apache Hadoop, For Real. [2014-04-17]. http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/

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

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

[9] Wu S, Li F, Mehrotra S, et al. Query optimization for massively parallel data processing //Proc of the 2nd ACM Symp on Cloud Computing, 2011: 1-13

[10] Derby. Implement optimizer support for bushy trees. [2014-04-17]. https://issues.apache.org/jira/browse/DERBY-4327

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

[12] Fender P, Moerkotte G. Reassessing top-down join enumeration. IEEE Trans on Knowledge and Data Engineering. 2012, 24(10): 1803-1818

[13] Fender P, Moerkotte G, Neumann T, et al. Effective and robust pruning for top-down join enumeration algorithms. //Proc of the 28th Int Conf on Data Engineering. IEEE, 2012: 414-425

[14] Fender P, Moerkotte G. Top down plan generation: From theory to practice. //Proc of the 29th Int Conf on Data Engineering. IEEE, 2013: 1105-1116

[15] Moerkotte G, Neumann T. Dynamic programming strikes back. //Proc of the 2008 ACM SIGMOD Int Conf on Management of Data. ACM, 2008: 539-552

[16] 周强, 陈岭, 马骄阳等. 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

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