列存储数据仓库中启发式查询优化机制

严秋玲[1)] 孙 莉[1)] 王 梅[1)] 乐嘉锦[1)] 刘国华[1),2)]

[1)](东华大学计算机科学与技术学院 上海 201620)

[2)](南京大学计算机软件新技术国家重点实验室 南京 210093)

摘 要 研究和实践表明列存储更加适合于大规模数据集上的即席查询的“读优化”应用需求.然而由于列存储的处理对象是列,此时传统的基于规则的查询优化方法并不完全适用.文中首先比较了列存储系统中查询优化与行存储系统的不同,在此基础上提出适合于列存储的启发式查询优化机制,其中包括启发式优化策略、重写规则、左深连接树结构和相关算法.实验表明:该文提出的启发式优化机制能有效减少候选计划的规模,排除大量不可能生成最优计划的计划,使得查询处理代价和执行时间大大减小.

关键词 列存储;查询优化;优化策略;重写规则;左深连接树

中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2011.02018

Heuristic Mechanism for Query Optimization in Column-Store Data Warehouse

YAN Qiu-Ling[1)] SUN Li[1)] WANG Mei[1)] LE Jia-Jin[1)] LIU Guo-Hua[1),2)]

[1)](*School of Computer Science and Technology, Donghua University, Shanghai* 201620)

[2)](*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210093)

Abstract It is well known that column-store architecture is more suitable for “read optimization” application in large scale dataset. However, due to the fact that data is organized in columns in column-store, the traditional rule-based query optimization methods are not fully applicable for such application. In this paper, we first compare the difference of the query optimization between the column-sotre and row-store, and then propose a heuristic mechanism for query optimization in column-store, including heuristic optimization strategy, relational algebra expression rewriting rules, left-deep join tree and relating algorithms. The experimental results show that the proposed heuristic optimization mechanism can effectively reduce the size of the candidate plan, and exclude a large number of plans which can not generate the optimal plan, so as to make the cost and implementation time of query processing greatly reduced.

Keywords column-store; query optimization; optimization strategy; rewrite rule; left-deep join tree

1 引 言

传统的事务型数据库系统是写优化(write-optimized)的,它们多数采用行存储的方式来存储数据,而分析型的数据仓库常常是短时间内写入大批新数据,然后长时间地进行 ad-hoc 查询操作,面向 ad-hoc 大数据量查询的系统应该是读优化(read-optimized)的[1],此时列存储的方式就表现出比行存储具有更加显著的性能优势.列存储系统是将同

收稿日期:2011-07-18;最终修改稿收到日期:2011-08-17.本课题得到核高基重大专项(2010ZX01042-001-003-004)、国家自然科学基金(61070031,61070032)资助.**严秋玲**,男,1986 年生,硕士研究生,主要研究方向为数据库、查询优化.E-mail: 21gxkfy@163.com.**孙 莉**,女,1964 年生,副教授,主要研究方向为数据库技术、面向对象分析与技术等.**王 梅**,女,1980 年生,博士,主要研究方向为数据库与多媒体.**乐嘉锦**,男,1951 年生,教授,博士生导师,中国计算机学会(CCF)会员,主要研究领域为数据库与数据仓库、软件工程理论与实践.**刘国华**,男,1966 年生,教授,博士生导师,中国计算机学会(CCF)会员,主要研究领域为隐私保护、文档复制检测、面向数据的业务过程管理.

一列数据连续存储，避免了将不相关列的数据读入内存，这种技术的特点是对复杂数据查询效率高、读磁盘少、存储空间小[2]，这样的列存储系统就成为了构建数据仓库的理想架构，因而引起数据库学术前沿和相关高新科技企业投入大量的人力和物力研发，相继开发出不少列存储系统如 C-Store[2]、Sybase[3]、MonetDB[4]等，它们也实际验证了列存储技术在读优先系统上的优越性. 而作为数据仓库中最常用、最重要的语句，查询语句的执行效率直接影响了数据仓库的性能，这使得在大容量的数据仓库上探索更高效的查询处理技术成为一种必然.

查询优化在现代数据仓库管理系统中一直占有重要的地位，目前，学界和业界多数在底层存储和执行上做优化处理，如压缩技术、索引技术、不可见连接技术[5-8]等等. 现有的列存储系统也多通过在存储上做改进来减少查询中的连接开销，如 C-Store 的“投影(projection)[1-2,8]”技术将属于同表的列按不同的组合存储，以此来提高查询效率；MonetDB 的“饼干图(cracker map)[9]”技术在查询时建立相关列的映射关系；由此可以看出，列存储技术在存储方面的优化已有不少研究成果.

查询重写[10](语法重新构造)作为查询语句逻辑优化的一个重要组成部分，它是将查询语句转换为另一种等价且高效的内部表示，也就是帮助查询优化器获得更好的执行计划[11]. 目前，基于规则的优化(RBO)[10,12]是查询重写中一个比较成熟的方法，它是根据指定的规则对逻辑计划进行优化. 在基于规则的模式下，优化的代价小，获得的执行计划通常也比较稳定. 大多数情况下，启发式查询优化[13-14]是基于规则的查询重写中应用最多的一种机制，它结合启发式优化策略，遵循关系代数中多个代数定律的重写规则来改写查询计划，使之转化成一个与其所需时间较小的等价的计划. 由于列存储系统与行存储系统在处理对象、选择和连接的级联形式、自然连接等方面存在较大差异，这就使得传统的启发式查询优化方法并不完全适用于列存储系统.

针对上述问题，本文提出了一种适用于列存储数据仓库的启发式优化机制. 首先，本文研究了传统的启发式查询优化的主要策略，并深入比较了列存储系统和行存储系统各自的存储特点，给出了它们在查询优化中的具体区别；然后提出了适合于列存储系统的启发式优化策略和关系代数表达式重写规则；接着论述了左深连接树在构建逻辑查询计划时的适用性，并给出了本文设计的双层左深连接树结构；再接着给出了相关的优化算法；最后，通过实验论证，本文提出的优化机制能生成高效的逻辑查询计划，使得查询效率大大提高.

本文第 2 节介绍列存储技术的研究背景以及列存储查询优化的研究现状；第 3 节基于对查询优化过程、传统启发式优化方法和列存储特点的分析研究，给出适应的优化策略和重写规则；第 4 节介绍左深连接树结构和相关的优化算法；第 5 节描述实验并对实验结果进行分析；最后是总结和展望.

2　相关工作

1985 年，Copeland 等人[15]提出了一种新的存储概念，简称 DSM，这就是列存储系统的雏形，它对列存储模型做了比较详细的介绍. 接着在以 Stonebraker Michael, Abadi Daniel J, Boncz Peter 为首的一批专家的大力提倡下，列存储的相关技术及应用快速发展，相继出现 PAX[16]、S-PAX[17]存储模型，其中 PAX 将同一元组的属性存储在一个磁盘页上，而页内所有元组属性值按列连续存放，以此来加速元组重构. 而 S-PAX 在 PAX 基础上，由若干磁盘页逻辑上构成一个超页，元组的不同属性分别存放在同一超页内的不同磁盘页上，即每个磁盘页按列连续存放元组属性值，以此来避免在查询执行过程中，将元组的无关列读入缓冲区，又能加速元组重构.

C-Stor[2]是一款开源的、运行于 Linux 系统的列存储数据库系统，于 2006 年 10 月发布，目前主要的学术研究都采用了该系统进行实验，在学术界比较流行. 它是将每列连续存储，多列的一个组合保存在一个投影(projection)中，按照其中一列排序. 因此 C-Store 查询经常基于一个投影，或者含有公共排序列的不同投影，以此减少元组重构和列的连接代价. 这样，C-Store 很好地完善了查询执行器，但是没有对初始的逻辑执行计划进行必要的优化处理.

MonetDB[4]是一款运行于 Linux 和 Windows 系统上的高性能开源列存储数据库，同时是一款内存数据库. 在存储设计上，采用 DSM 模型，同时用散列法代替 B^+ 树，以此解决建立多个 B^+ 树空间利用率低的问题；对初始的逻辑计划，MonetDB 也采用一些简单的优化策略和重写规则进行优化，如下推选择策略，使得选择操作尽早执行，减小中间关系大小，进而减少读写的 I/O 次数，以此达到优化效果. 但这些用到的策略和规则相对于列存储系统的

特点并不够深入，MonetDB 也仅仅设计和实现了一个比较粗糙的查询优化器.

此外包括其它的一些列存储系统如：Rasdaman、Sybase IQ、ParAccel 等也在查询优化方面提供了一些很好的解决方法，但它们的重点大都放在物理存储的改变上，在重写逻辑查询计划这方面涉及较少，实现过程中仅仅简单引入了传统行存储系统的一些优化策略和重写规则，这些列存储系统基本没有更深入地分析列存储系统的存储特点，提出一些列存储系统特有的优化策略和重写规则. 基于以上分析，本文结合行存储启发式查询优化的方法和列存储系统的存储特点，提出了面向列存储系统的基本优化策略、详细的重写规则、双层左深连接树结构以及相关优化算法，从而建立了一种比较完善的适用于列存储数据仓库的启发式查询优化机制.

3 优化策略和重写规则

3.1 相关概念

查询处理. 是指从数据库中检索数据的活动，其目标是将高级语言（例如 SQL）表示的查询转换为正确有效的、用低级语言表达的执行计划，即实现关系代数，并通过执行该计划来获取所需的数据.

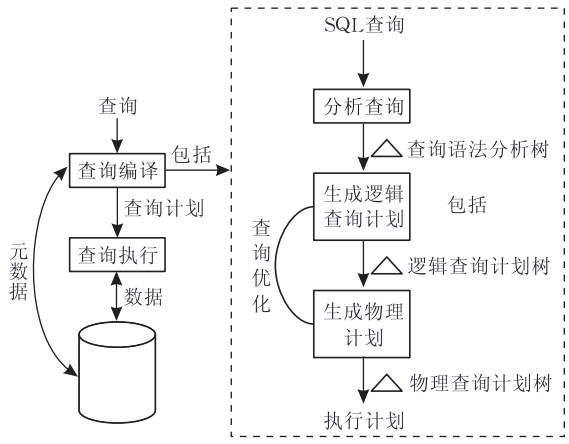

图 1 查询处理的流程

查询编译. 它是指通过语法分析、查询优化、制定物理执行策略将最初查询语句等价转化为物理执行计划的过程. 它主要分为 3 个步骤：分析查询、生成逻辑计划、生成物理计划.

查询优化. 它是指选择高效的执行计划的活动. 由于一个高级查询有多种等价的转换形式，查询优化的目标是选择其中资源占用最少的一种.

启发式查询优化. 在查询优化过程按照启发式重写规则将初始的查询计划转换成一个等价的、高效的查询计划，该查询计划可被转换成最有效的物理查询计划.

典型的启发式优化策略有

(1) 对每一个选择，利用等价变换规则使其尽可能深地靠近查询树的叶端；

(2) 对每一个投影，利用等价变换规则使其尽可能深地靠近查询树的叶端；

(3) 利用等价变换规则把选择和投影的串接合并成单个选择、单个投影或一个选择后跟一个投影.

3.2 列存储系统启发式查询优化与行存储系统的区别

在列存储系统中，为了重构元组，存储每一列数据时，每一列都附加一个伪列 *rowid*，形如〈*rowid*，*value*〉. 这样可以将每个列看作是〈*rowid*，*value*〉形式的二元表，此时列存储系统的查询优化与行存储系统的区别如下：

操作对象. 在行存储中，下推的目标对象是表；而在列存储中，下推的目标对象具体到某个列，每个列相当于一个由(*rowid*，*value*)组成的小表.

投影操作. 行存储由于是逐行存储，为了减少中间结果，需要投影查询中需要操作的列，因此在每个选择操作后，都伴随着投影操作，挑选对后面有用的属性，有相关投影的启发式重写规则. 而在逐列存储的系统中，根本不需要投影，只需操作与查询相关的列即可，所以没有相关投影的重写规则.

选择的级联形式. 行存储对于同表的选择，是级联形式的，先根据一个选择条件过滤元组，再根据另一个条件过滤经过筛选的元组. 这样能减少中间结果，也能减少代价. 而在列存储中，由于操作对象是单列，因此既可以选择级联形式减少中间结果，也可以选择单列 *rowid* 求交的形式减少中间结果，因此，列的选择相当于集合交，不会转化成固定的级联形式.

表之间连接的级联形式. 行存储系统中多表连接的处理需要得到较小的中间结果，需要估计连接的代价，然后决定连接顺序再去执行. 而列存储系统中由于连接条件具体到某两个列上，我们既可以选择级联地执行多个空间的连接条件，也可以选择求交的执行方法，同样可以减少中间结果.

自然连接. 列存储系统中由于操作对象是列，因此所有的自然连接都会转换成列与列的等值比较，也就是转化成 θ 连接，因此，没有自然连接的重写规则. 相反在行存储系统需要有关于自然连接的重写规则.

3.3　列存储系统的启发式优化策略

本文启发式优化的基本思想是：首先执行最具限制性的选择和连接操作.具体优化策略如下：

(1)把选择操作尽可能深地下推到查询树叶端(使得尽早执行选择操作.选择操作减少了关系中元组的数量从而降低了后面关系处理的复杂度，它一般可使得中间结果大大变小，常常能使执行时间节约几个数量级)；

(2)如果由多个谓词组成的选择条件，则可把该条件分解并分别将每个选择条件下推(有效地将不同表的选择条件下推到相关表的复杂操作下面)；

(3)合并同表同列的选择操作(消除重复和一次性执行完同列的所有操作)；

(4)将同表不同列的选择操作分组(一次性执行完同一张表中的所有相关列操作，大大减少后面表之间连接操作的中间关系)；

(5)两列之间的比较谓词操作或数学操作，无论是否来自同一张表，都是二元连接操作；

(6)把某些选择运算同在其前面执行的笛卡尔积结合起来成为一个连接运算(连接特别是等值连接的执行时间远远低于笛卡尔积的执行时间).

3.4　启发式重写规则

本文重点研究如下形式的 sql 查询：

select L from R where $\wedge/\vee(A_1,\cdots,A_n)$.

R 是关系的集合，L 是关系 R 的属性集，$A_1,\cdots,A_n$ 是由 and 或者 or 连接的谓词.在列存储系统中，查询处理的对象是列，所以此类查询可以作以下转化：

$$\text{select } L \text{ from}(K_1\times K_2\times\cdots) \text{ where } \wedge/\vee(A_1,\cdots,A_n)$$

K_i 是查询相关的列，再根据前面提出的优化策略，本文制定了如表 1 所示的关系代数表达式重写规则.

表 1　重写规则

序号	内容
Rule1	在列存储系统中处理的对象是列，将每个列看作是〈$rowid$，$value$〉形式的二元表，所以可以在传统的处理对象是表的基础上具体化到列上来： $\sigma_{\theta_i}(R)=\sigma_{\theta_i}(K_i)$，$\theta_i$ 是涉及引用关系R中列 K_i 的选择条件； $\sigma_\theta(R_1\times R_2)=\sigma_\theta(K_i\times K_j)$，$\theta$ 是涉及引用关系 R_1、R_2 中列 K_i、K_j 的选择条件；
Rule2	$\sigma_{\theta_i}(K_i)\wedge/\vee\sigma_{\theta_j}(K_j)=\sigma_{\theta_j}(K_j)\wedge/\vee\sigma_{\theta_i}(K_i)$， $\sigma_{\theta_i}(K_i)\bowtie_\theta\sigma_{\theta j}(K_j)=\sigma_{\theta j}(K_j)\bowtie_\theta\sigma_{\theta_i}(K_i)$， θ 是连接条件，θ_i，θ_j 是涉及同一引用关系中列 K_i、K_j 的选择条件，$i\neq j$；
Rule3	$(\sigma_{\theta_i}(K_i)\wedge\sigma_{\theta_j}(K_j))\wedge\sigma_{\theta_x}(K_x)=(\sigma_{\theta_i}(K_i)\wedge\sigma_{\theta_x}(K_x))\wedge\sigma_{\theta_j}(K_j)$；
Rule4	$\sigma_{\theta_i\wedge\theta_j}(R_1\times R_2)=\sigma_{\theta_j}(K_i)\wedge\sigma_{\theta_j}(K_j)$， θ_i，θ_j 分别是引用关系 R_1、R_2 中列 K_i、K_j 的选择条件；
Rule5	$\sigma_{\theta_i}(K)\wedge/\vee\sigma_{\theta_j}(K)=\sigma_{\theta_i\wedge/\vee\theta_j}(K)$， θ_i，θ_j 都是涉及引用列 K 的选择条件；
Rule6	$\sigma_{\theta_i}(K_i)\wedge/\vee\sigma_{\theta_j}(K_j)=\sigma_{\theta_i\wedge/\vee\theta_j}(K_i\times K_j)$；
Rule7	$\sigma_\theta(K_i\times K_j)=K_i\bowtie_\theta K_j$，$\theta$ 是涉及列 K_i、K_j 的比较条件，且不可分解； $\sigma_{\theta_i\wedge/\vee\theta_j}(K_i\times K_j)=\sigma_{\theta_i}(K_i)\bowtie_\theta\sigma_{\theta_j}(K_j)$，$\theta$ 是两列的连接条件：$rowid$ 相等；
Rule8	$\sigma_\theta(K_i\times K_j)=R_{k_i}\bowtie_\theta R_{k_j}$ θ 是列 K_i、K_j 的连接条件，且不可分解；R_{ki}、R_{kj} 分别是列 K_i、K_j 的引用关系；
Rule9	◇：代表笛卡尔积(×)或者 θ 连接($\bowtie_\theta$)， $\sigma_{\theta_i}(K_i)\wedge(R_i\diamond R_j)=\sigma_{\theta_i}(K_i)\diamond R_j$，$R_i$ 是列 K_i 的引用关系； $\sigma_{\theta_i}(K_i)\wedge\sigma_{\theta_j}(K_j)\wedge(R_i\diamond R_j)=\sigma_{\theta_i}(K_i)\diamond\sigma_{\theta_j}(K_j)$， R_i，R_j 分别是列 K_i、K_j 的引用关系； $\sigma_{\theta_i}(K_i)\wedge\sigma_{\theta_j}(K_j)\wedge(R_i\diamond R_j)=\sigma_{\theta_i}(K_i)\diamond R_2\times\sigma_{\theta_j}(K_j)$， R_i 是列K_i的引用关系，K_j 的引用关系既不是R_i，也不是R_j

3.5　查询重写举例

对于查询：

select $A.a$，$B.b$ from A，B，C

where $A.a>1$ and $B.b>3$ and $A.a>4$ and $B.b<5$ and $A.e=7$ and $B.f=9$ and $C.a=2$ and $A.b=5$ and $A.c=B.c$ and $C.g=B.g$；

此查询用关系代数表达式可表示为

$$Q_{ex}=\Pi_{A.b,B.b}(\sigma_{\substack{A.a>1 \text{ and } B.b>3 \text{ and } A.a>4 \text{ and}\\ B.b<5 \text{ and } A.e=7 \text{ and } B.f=9 \text{ and}\\ C.a=2 \text{ and } A.b=5 \text{ and } A.c=B.c \text{ and}\\ C.g=B.g}}(A\times B\times C)).$$

根据上面提出的优化规则对 Q_{ex} 进行重写：首先使用 Rule1、Rule4 将 Q_{ex} 处理对象由表具体化到引用列，就可得到如表 2 中所示的 T_1(Step1)；然后根据 Rule2、Rule5 合并 A 表中涉及 a 列的选择操作得到 T_2，合并 B 表中涉及 b 列的选择操作得到 T_3(Step2)；(Step3)接着将涉及 A、B 表的同表不同列的选择条件合并为笛卡尔积(Rule2、Rule6)，再根据 Rule7 将笛卡尔积转化为同表不同列的 θ($rowid$ 相等的连接条件)连接 T_4、T_5，同时根据 Rule8 将 A 与 B、B 与 C 的列之间的笛卡尔积转化为表之间的 θ(有关两表列之间的比较条件)连接 T_7、T_8；最后根据 Rule9 将关于 A、B、C 表中列的选择条件 T_4、T_5、T_6 下推到 θ 连接 T_7、T_8 下，使得对每张表尽早执行选择操作，再做表之间的连接操作(见表 2).

表 2 重写 Q_{ex}

步骤	操作
Step1	$Q_{ex}=\Pi_{A.a,B.b}(T_1)$, $T_1=\sigma_{A.a>1}(A.a)\wedge\sigma_{B.b>3}(B.b)\wedge\sigma_{A.a>4}(A.a)\wedge$ $\sigma_{B.b<5}(B.b)\wedge\sigma_{A.e=7}(A.e)\wedge\sigma_{B.f=9}(B.f)\wedge$ $\sigma_{C.a=2}(C.a)\wedge\sigma_{A.b=5}(A.b)\wedge\sigma_{A.c=B.c}(A.c\times B.c)\wedge$ $\sigma_{C.g=B.g}(C.g\times B.g)$.
Step2	$T_1=T_2\wedge T_3\wedge\sigma_{A.e=7}(A.e)\wedge\sigma_{B.f=9}(B.f)\wedge$ $\sigma_{C.a=2}(C.a)\wedge\sigma_{A.b=5}(A.b)\wedge\sigma_{A.c=B.c}(A.c\times B.c)\wedge$ $\sigma_{C.g=B.g}(C.g\times B.g)$, $T_2=\sigma_{A.a>1}(A.a)\wedge\sigma_{A.a>4}(A.a)=\sigma_{A.a>4}(A.a)$, $T_3=\sigma_{B.b>3}(B.b)\wedge\sigma_{B.b<5}(B.b)=\sigma_{B.b>3\wedge B.b<5}(B.b)$.
Step3	$T_1=T_4\wedge T_5\wedge T_6\wedge T_7\wedge T_8$, $T_4=T_2\wedge\sigma_{A.e=7}(A.e)\wedge\sigma_{A.b=5}(A.b)$ $=\sigma_{A.a>4}(A.a)\bowtie_\theta\sigma_{A.e=7}(A.e)\bowtie_\theta\sigma_{A.b=5}(A.b)$, $T_5=T_3\wedge\sigma_{B.f=9}(B.f)=\sigma_{B.b>3\wedge B.b<5}(B.b)\bowtie_\theta\sigma_{B.f=9}(B.f)$, $T_6=\sigma_{C.a=2}(C.a)$, $T_7=\sigma_{A.c=B.c}(A.c\times B.c)=A\bowtie_{A.c=B.c}B$, $T_8=\sigma_{C.g=B.g}(C.g\times B.g)=C\bowtie_{C.g=B.g}B$.
Step4	$T_1=T_4\bowtie_{A.c=B.c}T_5\bowtie_{C.g=B.g}T_6$.

4 左深连接树和算法实现

4.1 左深连接树

一般来说,一个二叉树如果所有右边的子女都是叶子结点的话,它就是左深树;一个二叉树如果所有左边的子女都是叶子结点的话,它就是右深树;既不是左深树,也不是右深树,称之为紧密树;一个左深连接树[18]的叶结点是带有除了连接之外的操作符的内部结点.

随着关系数目的增长,可形成的左深连接树的数目的增长几乎不会像可形成的所有树(包括左深树、右深树、紧密树)的数目增长那样快,这样有利于代价估计过程对给定关系数目形成的不同连接树的考虑;同时基于对连接顺序考虑,这种结构还具有以下优点:

(1)查询计划的搜索将可以用于比较大的查询.

(2)用于连接的左深连接树可以和通用的连接算法很好地交互——尤其是嵌套循环连接和一趟连接算法.

(3)基于左深连接树生成的执行计划将会比非左深连接树更有效.如果用的是一趟连接,并且"建立关系"是在左边,则任何时候所需要的内存都将比同样关系用非左深树的情况要小;如果是用迭代的方法实现的嵌套循环连接,则可以避免多次构建任意中间关系.

本文提出一种双层左深连接树结构(如图 2 所示),表层为不同表($R_1\cdots R_n$)之间的左深连接树,$\theta_1\cdots\theta_{n-1}$是不同表的列之间的连接条件,$\theta_i$($1\leq i\leq n-1$)可以为空,即表示为笛卡尔积的连接关系;内层为同表不同列($K_1\cdots K_i$)之间的左深连接树,$f_1\cdots f_{i-1}$是同表列之间的连接条件,f_j($1\leq i\leq i-1$)不可空,它是涉及同表不同列之间的比较操作连接条件或者 *rowid* 相等的连接条件.

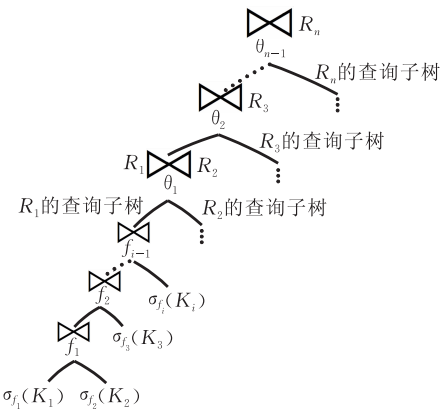

图 2 双层左深连接树

4.2 算法实现

在查询处理过程中,经过词法语法分析得到语法树,再对语法树进行预处理可以生成初始的查询计划,它仅仅是对查询语句的简单翻译,其树结构是比较粗糙的,也还没有经过优化重写,是不能作为最终的逻辑查询计划.本文对初始的查询计划树使用规则重写和构建双层左深连接树的实现过程如下:

步骤 1. 建立表之间的左深连接树.

从得到的查询树根结点遍历整棵树,不断地访问孩子结点,找出关于表之间连接操作的二元结点,同时对其结构作适当调整后,将它压入目标左深树 R_tree 中.其算法描述如下:

```
Function: LeftTree
Input: 查询树根结点 root _node,空树 R_tree
Output: R_tree
LeftTree(root_node,R_tree)
1. begin
2.   遍历查询树 root_node;
3.   if 遍历到的结点是"and"操作的二元结点 then
4.      R_tree=LeftTree(left_child,R_tree);//递归
5.      R_tree=LeftTree(right_child,R_tree);
6.      return R_tree;
7.   else if 遍历到的结点是连接操作的二元结点 then
8.         R_tree=Push(R_tree,connect_node);
           //将该连接结点压入 R_tree 中
9.         return R_tree;
10.    end if
11. end if
12. end
```

步骤2. 建立同表的列之间的左深连接树.

根据每一张表的表号，遍历整棵查询树，找出属于同一张表的选择操作，并将其构建成一个同表列之间的左深连接树 L_tree，接着对 L_tree 进行常量转化、消除重复、合并同列选择操作等规则优化操作，最后就生成了优化的同表列之间的左深连接树 L_tree. 其算法描述如下：

```
Function: Opt_L_tree
Input: 查询树根结点 root _node,空树 L_tree,表号 id
Output: L_tree
Opt_L_tree(id,root_node,L_tree)
1. begin
2.  遍历查询树 root_node;
3.  if 遍历到的结点是"and"操作的二元结点 then
4.    L_tree=Opt_L_tree(id,left_child,L_tree);
5.    L_tree=Opt_L_tree(id,right_child,L_tree);
6.    L_tree=Opt_onetable(L_tree);
      //合并同表同列选择操作、消除重复等
7.    return L_tree;
8.  else if 遍历到的结点是选择操作的一元结点 then
9.      if 该选择结点的表号==id then
10.       L_tree=Push(L_tree,select_node);
          //将该选择结点压入 L_tree 中
11.       return L_tree;
12.     end if
13.   end if
14. end if
15. end
```

步骤3. 下推选择.

遍历目标左深树 R_tree，根据其叶子结点的表号，将对应的 L_tree 下推到表连接操作结点下面. 其算法描述如下：

```
Function: Pushdown
Input: 表之间的左深树 R_tree,查询树根结点 root_node
Output: R_tree
Pushdown(R_tree,root_node)//下推
1. begin
2.  遍历 R_tree 中的每个内结点;
3.  对遍历到的每个结点,根据右孩子表号获得 L_tree;
4.  right_child=L_tree;
    //将 L_tree 压入到结点的右孩子位置,即将有关该
    //表的查询子树下推到连接操作下面
5.  if 左孩子不是叶子结点
6.    left_child=Pushdown(left _child,root_node);
      //递归
7.  else
8.    根据左孩子表号获得 L_tree
9.    right_child=L_tree;
10. end if
11. return R_tree;
12. end
```

这里以上一节提到的 Q_{ex} 为例，在查询处理的过程，它初始的查询计划树如图3. 首先调用 LeftTree 函数，从树的根结点开始遍历整棵查询树，若结点是逻辑操作"and"的二元结点，则左右孩子递归调用 LeftTree 函数；遍历到 "$A.c=B.c$" 二元结点，则将该结点取出并通过 Push 函数压入预先建立的空树 R_tree 中，同样遍历到"$B.g=C.g$"二元结点，该结点也会被压入 R_tree 树中，此时得到一棵关于 A、B、C 三表且带有连接条件的左深连接树(1)(见图4). 然后，通过 Pushdown 函数遍历(1)，首先遍历到的结点的右孩子的表号代表的是 A，则根据 A 的表号调用 Opt_L_tree 函数，在查询树中遍历到涉及 A 表的选择条件一元结点"$A.a>1$"，则将该结点取出并通过 Push 函数压入预先建立的空树 L_tree 中，同样"$A.a>4$"、"$A.b=5$"、"$A.e=7$"会依次被压入 L_tree 中，再调用 Opt_onetable 函数对 L_tree 作一些优化处理(合并同表同列选择操作、常量计算、常量转换、消除重复等)，这里是将"$A.a>1$"和"$A.a>4$"合并为"$A.a>4$"，从而得到列之间的左深树(2)，接着将(2)压入(1)中 A 的位置，也就是将 A 表的所有选择操作(2)下推到了关于 A 表的连接操作下面，由此生成的查询执行计划，将使得(2)会优先于关于 A 表的连接操作"$A.c=B.c$"执行. 同理可通过 B、C 的表号得到经过优化处理的(3)、(4)，并将其下推到左深树的对应叶子结点位置.

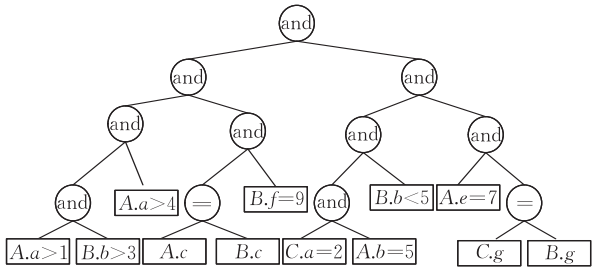

图3　初始的查询计划

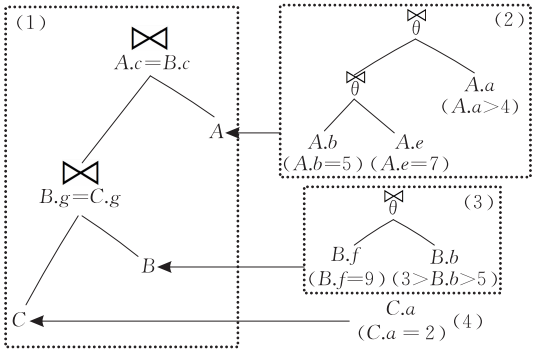

图4　基于左深连接树的查询计划

4.3 算法分析

本文的二叉树采用链式存储结构，设 n 为遍历的树中结点总数. 在前面介绍的算法中，步骤 1 建立表之间的左深连接树，需从查询树根结点遍历所有的内部结点(连接操作设计为二元结点，即有在初始的查询树中肯定为内部结点，所以不需要遍历到叶子结点)，再根据初始的查询树中的每个结点的出度是 0 或者 2 的特点，内部结点数目 $(n-1)/2$ 是随 n 线性变化的，所以算法 LeftTree 的时间复杂度为 $O(n)$；同理在步骤 2 中只需遍历到 and 二元结点下面的选择叶子结点即可，算法 Opt_L_tree 的时间复杂度也为 $O(n)$；而步骤 3 则需遍历所有的查询树结点，所以算法 Pushdown 的时间复杂度为 $O(n)$；由此可知，3 个算法的总的复杂度为 $3O(n)$，即本文所提方法总的时间复杂度为 $O(n)$.

5 实 验

5.1 实验环境

实验运行的硬件环境为 Intel(R)Core(TM) i3 CPU M380@2.53GHz，内存 2.00GB，操作系统为 Windows 7，开发环境为 Visual C++6.0.

实验以列存储数据仓库管理系统 DBMS3.0 为平台，它是以 $(rowid, value)$ 形式存放每列数据，$rowid$ 是为了重构元组，在每一列都附加一个行号，这样可以将每个列看作是 $\langle rowid, value\rangle$ 形式的二元表；其优化器也就可以采用本文提出的优化策略、重写规则和双层左深连接树结构来对初始的查询计划进行优化处理.

5.2 数据集

实验采用的数据来源于数据仓库基准数据集 SSB[19]，数据库的模型是星型模型，我们使用基准测试提供的数据产生器生成了 SSB 的数据集实例. 每个实例数据集的大小是用一个增量因子控制的，记为 SF. 实验选用 $SF=1$，事实表 lineorder 的数据量为 6000000 行，维表 part 的数据量为 2000000 行、Customer 的数据量为 30000 行、Supplier 的数据量为 2000 行.

5.3 实验结果与分析

(1) 从前面的启发式重写规则介绍可知，查询语句中能合并和下推的选择条件直接影响着重写规则对性能优化的显著性，为了测试重写规则在优化过程的有效性，在此设计了 10 例查询语句进行实验，其中前 5 例选取表 Part 为引用表，逐渐增加关于列 partkey、size 和 color 的选择条件的个数；后 5 例选取表 Lineorder、Part 为引用表，并带有 Lineorder.partkey=Part.partkey 连接操作，逐渐增加关于两个引用表中列的选择条件. 前 5 例和后 5 例分别在相同的条件下，重写查询计划前后的执行时间对比如表 3 所示.

表 3 基于重写规则的执行时间比较

Query	执行时间/s	
	重写规则前	重写规则后
1	0.0951	0.0873
2	0.1156	0.0927
3	0.1325	0.1081
4	0.1497	0.1205
5	0.1653	0.1270
6	0.3950	0.2521
7	0.4198	0.2075
8	0.4205	0.1862
9	0.4337	0.1689
10	0.4563	0.1591

从表 3 可看到 10 例查询的各自的执行时间，先来对比观察前 5 例查询的执行时间，Query1 到 Query5 的重写后的执行时间对比重写前的有显著的减少，且减少程度递增，而 Query1 到 Query5 是逐渐增加同表同列的选择条件，这表明本文提出的重写规则能有效合并同表同列选择条件，使得查询计划在执行中能消除重复和一次性处理完一列的所有选择操作，减少查询的执行时间，从而达到优化的目的；再来观察后面 5 例查询语句，Query6 到 Query10 重写后的执行时间比较重写前有大大的减少，并随着引用表相关列选择条件的增多(可下推选择条件的增多)，重写后的执行时间相对重写前减少的更多，这也较好地验证了通过将选择条件下推到复杂的操作前面，可尽早执行选择操作，有效地减少中间关系的大小，使得查询的执行时间大大减小；此时再对比前面 5 例重写前后执行时间上的变化，可看出下推选择后带来的时间效率提升的更高，表明下推策略的优化程度比同列合并要高，体现了它的必要性；因为连接操作普遍比其它操作的代价更高，也造成了后 5 例的执行时间基本比前 5 例的偏高的现象；通过分析这些重写规则，不难看出它们都会大大减小中间结果，使得内存尽早释放对查询结果无用的数据，从而提高了空间利用率，鉴于重写规则对查询时间和空间性能的提升，表明本文提出的规则在查询优化过程中的必要性和有效性.

(2) 前文论述过左深连接树结构的适用性，它的优化性能与查询相关表和列的个数有着密切联

系.为了测试双层左深连接树结构能够带来查询性能提升,实验设计 4 种查询计划树结构:表之间连接和同表列之间都采用非左深连接树(RR)、表之间采用左深连接树而同表列之间采用非左深连接树(LR)、表之间采用非左深连接树而同表列之间采用左深连接树(RL)、表之间和同表列之间都采用左深连接树(LL),在此设计了逐步增加引用表和列的个数(当引用表和列的个数超过 2 个才能构建典型的左深连接树结构)的 10 例查询语句,其中查询 1 的引用表只有 Lineorder 表,在此基础上逐步加入引用表 Part、Customer、Supplier 表,同时增加每张表的相关列的选择条件.这 10 例查询语句在查询处理过程分别采用这 4 种结构,在相同的条件下其执行时间实验结果如图 5 所示.

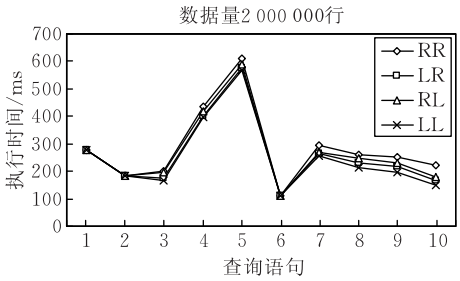

图 5　基于左深连接树的执行时间比较

从图中观测 4 条线的高低位置,对于相同的查询语句,RR 位置多数偏高,花费的时间最多,LL 位置多数偏低,花费的时间最少,实践论证了左深连接树结构在本文的优化机制中能进一步促进查询优化.从图中也可看到四条线有时会有重合的点,这是因为当查询中引用表的个数或者列的个数没有超过 2 个的时候,左深连接树和非左深连接树的结构是一样的,也就不会带来执行时间上的变化,但当表的个数或者列的个数超过 2 个的时候,左深连接树和非左深连接树的结构差异就直接影响着各自的查询执行时间,正如图中 LL 这条线除去重合的点,一直处于其余三条线下方,表明了采用左深连接树结构的查询计划的执行效率更高.

(3)结合前面两个实验提到的重写规则和左深连接树结构,在此提出下面 4 种情况:使用重写规则和左深连接树结构(TL);使用重写规则和非左深连接树结构(TR);不使用重写规则,但使用左深连接树结构(FL);既不使用重写规则,也不使用左深连接树结构(FR).实验设计了 10 例查询语句,查询 1 到查询 10 逐步增加可合并、下推的选择条件以及查询引用表和列的数量.在相同的条件下,这 4 种情况下的执行时间如图 6 所示.

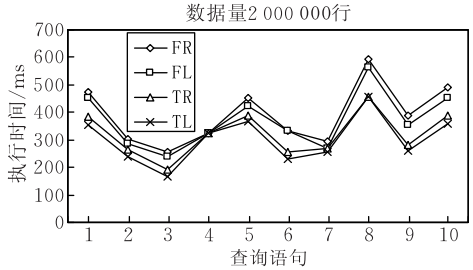

图 6　基于重写规则和左深连接树的执行时间比较

从图中可看出,FR 执行时间最多,TL 执行时间最少,由此充分验证了同时使用重写规则和左深连接树结构能更大提升优化器的有效性,使得查询效率更高,也就达到了本文提出这些优化方法的目的.

6　总结与展望

基于传统的行存储的启发式查询优化方法,本文分析了列存储查询优化与行存储的区别,重点抓住列存储数据仓库的处理对象已经具体到每列的关键点,进而提出了适合于列存储系统优化策略、重写规则和双层左深连接树结构.通过实验验证了查询语句通过本文的启发式优化机制,能得到可生成最优计划的候选树,大大提高了查询效率.未来的工作将继续完善本文提出的启发式优化机制,重点将转向将 goup_by 聚集的一些选择操作上推到连接操作下面和将子查询转化为半连接的重写规则,同时考虑相关列上是否建有索引来调整该列在左深连接树中位置的策略,使得本文提出的优化机制得到进一步完善,以获得查询效率的最大提升.

参　考　文　献

[1] Abadi Daniel J. Query execution in column-oriented database systems[Ph. D. dissertation]. Department of Electrical Engineering and Computer Science, MIT, Boston, 2008 (in America)

[2] Stonebraker Mike, Abadi Daniel J et al. C-Store: A column-oriented DBMS//Proceedings of the 31st VLDB Conference. Trondheim, Norway, 2005: 553-564

[3] MacNicol R, French B. Sybase IQ multiplex-designed for analytics//Proceedings of the 30th VLDB Conference. Toronto, Canada, 2004: 1227-1230

[4] Boncz P A. Monet: A next-generation DBMS kernel for query-intensive applications[Ph. D. dissertation]. Universiteit van Amsterdam, Amsterdam, 2002(in Netherlands)

[5] Abadi D J. Integrating compression and execution in column oriented database systems//Proceedings of the SIGMOD. Chicago, IL, USA, 2006: 671-682
[6] Gupta A, Davis K C. Performance comparison of property map and bitmap indexing//Proceedings of the 5th ACM International Workshop. McLean, USA, 2002: 65-71
[7] Neil P O, Quass D. Improved query performance with variant indexes// Proceedings of the ACM SIGMOD International Conference on Management of Data. New York, 1997: 38-49
[8] Abadi Daniel J, Madden Samuel R, Hachem Nabil. Column-stores vs. row-stores: How different are they really?//Proceedings of the 2008 ACM SIGMOD International Conference. Vancouver, BC, Canada: ACM, 2008: 967-980
[9] Idreos Stratos, Kersten Martin L, Manegold Stefan. Self-organizing tuple reconstruction in column stores//Proceedings of the 35th SIGMOD International Conference. Providence, Rhode, Island, 2009: 297-308
[10] Pirahesh H, Hellerstein J M, Hasan W. Extensible/rule based query rewrite optimization in starbust//Proceedings of ACM SIGMOD. New York, USA, 1992: 39-48
[11] Hellerstein J M, Stonebraker M. Predicate migration: Optimizing queries with expensive predicates//Proceedings of the ACM SIGMOD. New York, USA, 2005: 267-375
[12] Finance B, Gardarin G A. Rule-based query rewriter in an extensible DBMS//Proceedings of the 7th Conference on Data Engineering. Kobe, Japan, 1991: 248-256
[13] Fegaras L. A new heuristic for optimizing large queries// Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA'98). London, 1998: 726-735
[14] Swami A. Optimization of large join queries: Combining heuristics and combinatorial techniques//Proceedings of the ACM SIGMOD International Conference on Management of Data. Portland, Oregon, 1989, 18(2): 367-376
[15] Copeland George P et al. A decomposition storage model// Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data. New York, NY, USA, 1985: 268-279
[16] Ailamaki A, Dewitt D J et al. Weaving relations for cache performance//Proceedings of the 27th VLDB Conference. San Francisco, 2001: 169-180
[17] Bößwetter Daniel. SPAX-PAX with super-pages//Proceedings of the ADBIS. Riga, Latvia, 2009: 362-377
[18] Hector Carcia-Molina, Ullman Jeffrey D, Widom Jennifer. Database System Implementation. Upper Saddle River, New Jersey: Prentice Hall, 2000: 238-302
[19] O'Neil Pat, O'Neil Betty, Chen Xuedong. Star Schema Benchmark Revision 3. June 5, 2009[EB/OL]. 2009[2010-2-9]. http: //www. cs. umb. edu/~poneil/StarSchemaB. PDF

YAN Qiu-Ling, born in 1986, M. S. candidate. His research interests include database, query optimi- zation.

SUN Li, born in 1964, associate professor. Her main research interests include database and object-oriented technology.

WANG Mei, born in 1980, Ph. D.. Her research interests include database, image semantic analysis, and information retrieval.

LE Jia-Jin, born in 1951, professor, Ph. D. supervisor. His research interests include database and data warehouse, software engineering theory and practice.

LIU Guo-Hua, born in 1966, professor, Ph. D. supervisor. His main research interests include database theory, uncertain database and business process management.

Background

This work is supported by the Research Project on Core Electronics Device, General-purpose Chips for High-end Usage and Basic Software (2010ZX01042-001-003-004) and the National Natural Science Foundation of China (61070031, 61070032). The research on column-oriented data warehouse management system is a hot topic in recent years and the purpose of this project is to conduct a deep investigation on this field. Query rewriting is an important part of the query optimization in data warehouse. It can transform initial query plan into an equivalent and more efficient logical query plan. However, most of the existing column-store systems pay more attention on changing of physical storage in the query optimization. There are a few studies in query rewriting. These work only introduce some simple optimization strategy and rewrite rule of row-store system. Thus, we propose a heuristic mechanism for query optimization in column-store, including heuristic optimization strategy, relational algebra expression rewriting rules, left-deep join tree, and relating algorithms.