ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.17, No.8, August 2006, pp.1743−1752
DOI: 10.1360/jos171743

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

大型压缩数据仓库上的 Iceberg Cube 算法*

骆吉洲+, 李建中, 赵 锴

(哈尔滨工业大学 计算机科学与技术学院,黑龙江 哈尔滨 150001)

An Iceberg Cube Algorithm for Large Compressed Data Warehouses

LUO Ji-Zhou+, LI Jian-Zhong, ZHAO Kai

(School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China)

+ Corresponding author: Phn: +86-451-86415280 ext 20, E-mail: luojizhou@hit.edu.cn, http://db.cs.hit.edu.cn/~luojizhou

Luo JZ, Li JZ, Zhao K. An Iceberg Cube algorithm for large compressed data warehouses. *Journal of Software*, 2006,17(8):1743−1752. http://www.jos.org.cn/1000-9825/17/1743.htm

Abstract: Iceberg Cube is meaningful for OLAP (on-line analysis processing) and compression techniques play more and more important role in reducing the storage of data warehouse and improving the efficiency of data operations. It is really a problem to compute Iceberg Cube efficiently in the compressed data warehouse. The compression techniques of data warehouse are introduced concisely in this paper, and an algorithm to compute Iceberg Cube in compressed data warehouse by mapping-complete methods is proposed. Experimental results show that this algorithm outperforms the direct method that selects Iceberg Cube tuples from the complete computed cube.

Key words: Iceberg Cube; compressed data warehouse; data cube; bottom-up computation

摘 要: Iceberg Cube 操作是 OLAP(on-line analysis processing)分析中的一种重要操作.数据压缩技术在有效减小数据仓库所需的数据空间和提高数据处理性能方面的作用越来越明显.在压缩的数据仓库上,如何快速、有效地计算 Iceberg Cube 是目前亟待解决的问题.简要介绍了数据仓库的压缩,然后给出了在压缩数据仓库中计算 Iceberg Cube 的算法.实验结果表明,该算法的性能优于先在压缩数据上计算 Cube 再检查 having 条件这种方法.

关键词: Iceberg Cube;压缩数据仓库;数据立方体;自底向上计算
中图法分类号: TP311 文献标识码: A

OLAP(on-line analysis processing)分析是数据仓库的主要应用,其中最重要的操作是 Cube 操作.Cube 操作是 group by 操作的推广,一个完整的 Cube 操作是对数据集合中的 n 个属性的任意组合计算其 group by 操作的结果,一个基本的 group by 操作称为一个 cuboid,2^n 个 group by 操作的结果的并称为数据立方体(data cube).Cube 操作的基本问题是快速而有效地计算完整的数据立方体.文献[1−5]给出了几个有效计算数据立方体的算法.然

* Supported by the National Natural Science Foundation of China under Grant No.60273082 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2001AA415410 (国家高技术研究发展计划(863)); the National Grand Fundamental Research 973 Program of China under Grant No.G1999032704 (国家重点基础研究发展规划(973)); the Natural Science Foundation of Heilongjiang Province of China under Grant Nos.F0208, zjg03-05 (黑龙江省自然科学基金)

Received 2005-06-24; Accepted 2005-11-09

而一些研究者指出[5−8]:当数据集的维数和数据量增大时,计算数据立方体所需要的时间开销和空间开销都呈指数增长.为了克服这个困难,一些研究者提出了有效削减数据立方体体积的方法,如 Condensed Cube[4],Dwarf Cube[5],Quotient Cube[6]和基于图结构的 Free Cube[7];另一些研究者则有选择地计算并存储某些 cuboid 的结果或只计算和存储 Iceberg Cube[7,8].

简单地说,Iceberg Cube 操作就是带有 having 条件的 Cube 操作,即 Iceberg Cube 就是在计算数据立方体时在每个基本 group by 操作中添加同一个 having 条件后计算得到的结果.Iceberg Cube 除了可以削减数据立方体的体积之外,其本身在 OLAP 分析中也十分有用.事实上,所有 cuboid 组成了一个代数格[1],属性较多的 cuboid 位于格的较高层次中,属性较少的 cuboid 位于格的较低的层次中,相邻层次的两个 cuboid 有偏序关系当且仅当层次较低的 cuboid 可以由层次较高的 cuboid 来计算,如后文图 1(a)所示;如果 Iceberg Cube 中的 having 条件是对聚集结果的支持度进行限制,则格中层次较高的 cuboid 的结果很可能是原始数据,其聚集效果不明显,它们在 OLAP 分析中的意义也不大[7];故计算 Iceberg Cube 时忽略这些 cuboid,既可减小存储空间又可提高 OLAP 分析效率.计算 Iceberg Cube 最直接的方法是先计算出完整的数据立方体,再用 having 条件过滤结果以删除不满足 having 条件的结果;然而,文献[7]指出这种做法的时间开销非常大.因此,很多研究者提出了直接计算 Iceberg Cube 的算法[7−9],最基本的算法是 BUC(bottom-up computation)和 TDC(top-down computation);此外,文献[9]还提出用 star_cube 方法来计算 Iceberg Cube.

另一方面,人们还引入数据压缩技术来削减大型数据仓库的空间开销[1,3,5,10−12].大型数据仓库有两种主要的存储结构:基于关系表的数据仓库(relational data warehouse,简称 RDW)和基于多维数组的数据仓库(multi-dimensional data warehouse,简称 MDW).当数据仓库中的数据比较稀疏且数据量非常大时,人们常采用基于多维数组的存储结构,这样做避免了对维属性的存储,极大地节省了数据仓库所需的空间[11].在压缩方法被引入数据仓库后,人们开始研究如何在压缩数据上有效地计算数据立方体.一些研究者提出了在压缩数据仓库中计算数据立方体的算法[1,3].文献[3]讨论了基于 chunk_offset 方法压缩的多维数据仓库上的 Cube 算法,该算法用大量的内存空间以流水方式同时计算多个 cuboid 的聚集结果,并仅在属性维较少的情况下能够获得较高的性能,其压缩方法也仅对稀疏数据立方体有效.文献[1]根据 cuboid 之间的前缀、中缀和后缀关系给出了启发式规则,用这些规则确定 cuboid 的计算次序,同时生成 Cube 的计算计划,然后根据计算计划读入相应的数据并利用压缩方法的特性自底向上无须解压缩地计算 Cube 结果;由于该方法在计算过程中无须解压缩操作,因此其性能远优于先解压缩再计算 Cube 这一传统方法.据我们所知,文献[1,3]是仅有的在压缩数据上计算 Cube 的工作,还没有人提出在压缩数据仓库中直接计算 Iceberg Cube 的算法.本文将在压缩的 MDW 中讨论 Iceberg Cube 的计算问题.

本文第 1 节介绍大型数据仓库的压缩算法.第 2 节给出大型压缩数据仓库中直接计算 Iceberg Cube 的算法.第 3 节是实验结果.最后给出结论和需要进一步研究的问题.

1 大型数据仓库的压缩

数据仓库由多个多维数据集构成,一个 n-维数据集可以表示为 $R(D_1,\ldots,D_n,M_1,\ldots,M_m)$,其中 D_i 是维属性,M_j 是度量属性.将多维数据集用多维数组存储,所得数据仓库称为多维数据仓库(MDW).本文只讨论多维数据仓库压缩及其上的 Iceberg Cube 算法.

将 n-维数据集 $R(D_1,\ldots,D_n,M_1,\ldots,M_m)$转化成多维数组需要如下 3 个步骤:

1. 用多维数组存储多维数据集合.先构建 n 维数组 $A_1,\ldots,A_m$,使得第 i 维的大小等于$|D_i|$;设$(d_1,\ldots,d_n,m_1,\ldots,m_m)$是 $R(D_1,\ldots,D_n,M_1,\ldots,M_m)$的一个元组,则用 d_i 索引第 i 维,找到多维数组的一个存储位置,再将$(m_1,\ldots,m_m)$依次存储到由该位置确定的存储单元 $A_1,\ldots,A_m$ 中.这种存储方式无须存储维属性值,节省了大量的空间.

2. 将多维数组线性化.即选择适当的线性化函数,将多维数组线性化为一维数组.我们给出一个简单、常用的线性化函数 LINEAR.设$(d_1,\ldots,d_n,m_1,\ldots,m_m)$是多维数据集的一个元组,则该元素在一维数组中的位置是 $LINEAR(d_1,\ldots,d_n)=d_1|D_2|\ldots|D_n|+d_2|D_3|\ldots|D_n|+d_n$.这个简单的函数有逆函数,故我们可以由数据在一维数组中的位

置来计算它在多维数组中的位置.

3. 数据压缩,即选择适当的数据压缩算法来压缩线性化过程得到的数组.

对这 3 个过程采取不同策略就得到不同的压缩算法.文献[3]提出了 chunk_offset 压缩方法:在步骤 1 之后,将每个维连续划分为若干区间(多维数据空间被划分为多个 chunk);步骤 2 分别线性化每个 chunk;步骤 3 对稠密 chunk 不作进一步压缩,仅对稀疏 chunk 用 offset 方法进行压缩.这种压缩方法对比较稀疏的多维数据集合能够取得较好的压缩效果,但对比较稠密的多维数据集合效果不明显,而且维的数值化过程会严重影响 chunk 的稠密性,进而影响压缩效果.文献[11]给出了一种映射完全的压缩方法,其中在步骤 2 中使用了前面的例子所给出的线性化函数,在步骤 3 中采用了称为 Header 的压缩技术[13].其优点在于:它存在前向函数和后向映射,根据前向映射可以由未压缩数据计算该数据在压缩数据文件中的位置,根据后向映射可由压缩数据计算该数据在原始数据集中的位置.

本文所给出的 Iceberg Cube 算法适用于所有由映射完全的压缩技术产生的压缩数据仓库,但为了叙述方便,在行文时采用文献[11]所给出的方法来压缩数据仓库,实验也是基于这种压缩技术来实现的.该压缩方法有如下性质:

定义 **1**. 线性化函数 LINEAR 中采用的属性序列 $D_1,\ldots,D_n$ 称为线性化属性序列.

命题 **1**. 如果 LINEAR 中线性化属性序列 $D_1,\ldots,D_n$ 满足$|D_1|>\ldots>|D_n|$,而$\langle a_1,\ldots,a_n\rangle,\langle b_1,\ldots,b_n\rangle\in D_1\times\ldots\times D_n$ 且存在 $k(<n)$使得当 $i\le k$ 时,$a_i=b_i$ 且 $a_{k+1}<b_{k+1}$,则 $LINEAR(a_1,\ldots,a_n)<LINEAR(b_1,\ldots,b_n)$.

尽管其证明很简单,命题 1 却有着重要意义.它表明,适当选取线性化属性序列后,线性化过程相当于一个 group by 操作,使得元组有一定的顺序,且由于 Header 压缩方法使数据逻辑位置的次序与物理位置的次序保持一致,故整个压缩方法使数据的顺序与线性化后的顺序保持一致.在计算 Iceberg Cube 时,这个性质将被用来划分数据.

命题 **2**. 将 $D_1\times\ldots\times D_n$ 中的元组 $a=\langle a_1,\ldots,a_n\rangle$投影去掉维 $D_k(1\le k\le n)$得到 $D_1\times\ldots\times D_{k-1}\times D_{k+1}\times\ldots\times D_n$ 中的元组 $a'=\langle a_1,\ldots,a_{k-1},a_{k+1},\ldots,a_n\rangle$,并记 $w=|D_{k+1}|\ldots|D_n|$.如果分别取 $D_1,\ldots,D_n$ 和 $D_1,\ldots,D_{k-1},D_{k+1},\ldots,D_n$ 为这两个数据集的线性化属性序列,则 $LINEAR(a')=[LINEAR(a)-a_k*w-LINEAR(a) \bmod w]/|D_i|+LINEAR(a) \bmod w$.

其中,$a \bmod b$ 是将 a 对 b 取模.该命题表明,数据在一个(或多个)维上的投影操作可以方便地进行,即用 Header 方法的后向映射得到数据的逻辑位置,再用命题 2 得到新的逻辑位置,然后用 Header 方法的前向映射计算新数据的物理位置.整个操作可在压缩状态下完成,数据无须解压缩过程.此过程被广泛用于我们的算法中.

2 压缩数据仓库中的 Iceberg Cube 算法

2.1 基本思想

在论述中,我们将待计算的 Iceberg Cube 记为 *ICube*,将 *ICube* 的输入数据集记为 $D(A_1,\ldots,A_n,M_1,\ldots,M_k)$.我们的算法要求 *ICube* 的 Having 条件中的聚集函数满足如下定义的反单调性,且 *ICube* 中的所有聚集函数满足如下定义的分布性或代数性.

设 f 是 *ICube* 中的一个聚集函数,$A,B\subseteq\{A_1,\ldots,A_n\}$,$S$ 是 D 的任意子集且 S 被划分成 $S_1,\ldots,S_k$,使得$\cup_i S_i=S$,$S_i\cap_{i\ne j}S_j=\varnothing$.我们将 f 在 S 上对 A 的聚集结果表示为 $f_A(S)$.

定义 **2**. 设 f 是 *ICube* 的 Having 条件中的一个聚集函数.对任意 $A,B\subseteq\{A_1,\ldots,A_n\}$,如果 f 在 S 上对 A 的聚集结果 $f_A(S)$不满足 *ICube* 的 Having 条件,蕴含 f 在 S 上对 $A\cup B$ 的聚集结果 $f_{A\cup B}(S)$也不满足 *ICube* 的 Having 条件,则称 f 对这个 Having 条件满足反单调性.

min(对≤条件),max(对≥条件),*sum*(对非负度量操作时对≥条件),*count*(对≥条件)等函数均满足反单调性.但某些聚集函数(如 *average*)不满足反单调性.文献[8]讨论了将某些复杂聚集函数(如 *average*)转换成反单调聚集函数的方法,并保证不丢失 *ICube* 中的元组.基于此,我们假设 *ICube* 中 having 条件的聚集函数均满足反单调性.

定义 **3**. 设 f 是 *ICube* 的聚集函数,$A\subseteq\{A_1,\ldots,A_n\}$.若存在函数 g 使 $f_A(S)=g(f_A(S_1),\ldots,f_A(S_k))$,则称 f 满足分布性.

许多聚集函数均满足分布性.如 min,max,*sum*,*count* 等,它们对应的函数 g 分别是 min,max,*sum*,*count*.

定义 **4**. 设 f 是 *ICube* 的聚集函数,$A\subseteq\{A_1,\ldots,A_n\}$.若存在多值函数 g 和单值函数 h 使 $f_A(S)=h(g_A(S_1),\ldots,g_A(S_k))$,则称 f 满足代数性.

代数性聚集函数如 *average*,它所对应的多值聚集函数为$\langle count,sum\rangle$,单值函数 $h=sum(sum_i)/sum(count_i)$.

聚集函数的分布性和代数性使得我们在内存不足以容纳全部数据时可将数据划分成若干个部分,然后对每部分数据分别聚集,再通过运算得到正确的聚集结果.

根据处理 cuboid 的顺序,*ICube* 算法可分为自顶向下算法(TDC)和自底向上算法两类(BUC).在由所有 cuboid 构成的代数格中(如图 1(a)所示),TDC 算法从包含属性最多的 cuboid 开始计算,向下逐步计算各个层次的 cuboid,这类方法常用于 RDW,它往往需要生成复杂的中间结果.此外,这类算法无法利用聚集函数的反单调性来提高计算效率.TDC 的处理过程如图 1(b)所示.BUC 算法从包含属性最少的 cuboid 开始计算,向上逐步计算各个层次的 cuboid,其处理过程如图 1(c)所示.这一类算法的核心思想是对输入数据逐步进行划分,逐次处理各个部分的数据.如果在计算过程中发现某个 cuboid 在某个数据子集上不满足 Having 条件,则利用聚集函数的反单调性可知其上层的 cuboid 在相应的数据子集上也不能满足 Having 条件,由此对计算过程进行剪枝以提高计算效率.BUC 算法的缺点是要求 Having 条件中的聚集函数满足反单调性.我们给出的算法是 BUC 算法,这是由于映射完全的压缩方法便于数据划分,同时又可以利用聚集函数的反单调特性来提高计算效率.

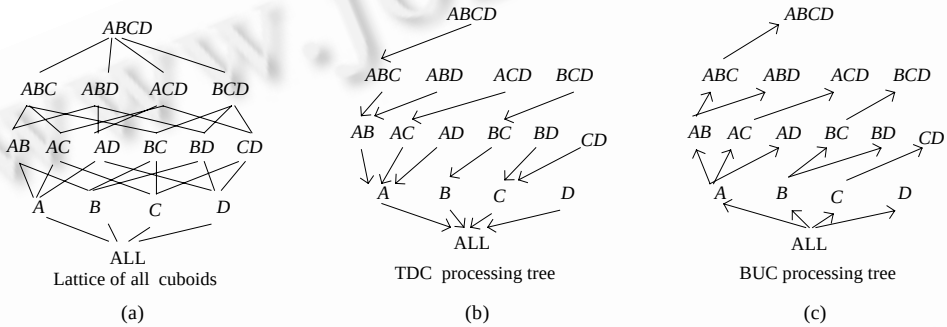

Fig.1 The procedure of computing Iceberg Cube

图 1 Iceberg Cube 的计算过程

在给出算法之前,我们先描述算法的基本思想.对于多维数据集 $D(A_1,\ldots,A_n,m_1,\ldots,m_k)$,我们先根据其属性集合生成如图 1(c)所示的 BUC 处理树,然后将这棵树划分成 n 个分支,第 i 个分支包含以 A_i 为前缀的所有 cuboid,如图 2 所示.该划分得到的分支称为一级分支,并将所有一级分支依次表示为 $T_1,\ldots,T_n$.算法在运行时,逐次处理各个一级分支,处理每个一级分支的数据集合各不相同,处理分支 T_i 所需的数据集 D_i 需要在处理 T_{i-1} 时生成,且处理分支 T_1 所需的数据 D_1 即为输入数据 D.在处理分支 T_i 时,数据 D_i 被划分成若干个互不相交的部分 $D_{i1},\ldots,D_{is_i}$,每个部分在该分支的公共前缀 A_i 上具有相同的取值,每一部分数据被称作一个数据块.这样,同一个分支在不同数据块上的聚集结果不会得到相同的元组.因此,处理分支 T_i 时可以依次处理每个数据块 D_{ij},同时生成处理下一个分支所需的部分数据,放入临时文件 F_{ij}.当分支 T_i 处理后,利用第 1 节所给出的性质合并 $F_{ij}(1\le j\le s_i)$ 得到 D_{i+1}.重复上述过程,直到所有分支被处理完毕为止.注意:临时数据也是以压缩形式存在的,整个计算过程均无需显式的解压缩过程.

当内存不足以容纳分支的一个数据块时,我们将较大的一级分支划分成如图 3 所示的二级分支.在处理每个一级分支时,用前面类似的过程来依次处理每个二级分支.我们将在第 2.3 节中阐述此过程.

2.2 基于一级分枝的**Iceberg Cube**算法

我们先假设内存足够大,在处理任何一级分支 T_i 时,内存能够容纳该分支的每个完整数据块 D_{ij}.在处理一级分支 T_i 时,算法依次将各个数据块 D_{ij} 读入内存;当数据被读入内存时,利用后向映射得到数据在线性化数组中的位置,因此,内存中所有数据项都形如$(Tag,value_1,\ldots,value_k)$,其中 *tag* 是线性化函数作用到该数据项上得到的值,而 $value_j$ 是该数据项的度量值;对于内存中的数据,自底向上地计算该分支中各个 cuboid 的聚集值.再利用第

1 节命题 2 中的方法将内存中的数据进行投影,去掉下一个一级分支中无关的维 A_i,并进行必要的聚集操作;最后,将投影后的数据依据下一个分支的公共属性进行排序,压缩后写到临时文件 F_{ij} 中.当所有数据块被处理完之后,合并所有 F_{ij},得到处理下一个分支所需的数据 D_{i+1}.重复上述过程,直到完成所有分支的处理.

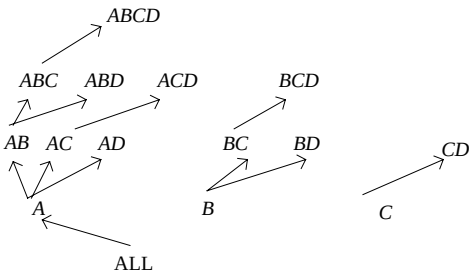

Fig.2 An example of branches of 1th order

图 2 一级分支示例

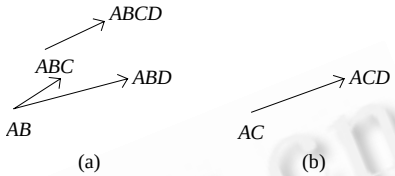

Fig.3 An example of branches of 2th order

图 3 二级分支示例

在使用内存中的数据 D_{ij} 来计算分支 T_i 中所有 cuboid 的聚集值时,我们以深度优先的方式自底向上地遍历 T_i,依次处理各个 cuboid.在处理当前 cuboid,即 *cuboid* 时,所有数据项的 *Tag* 值保持不变,利用命题 2 中的方法计算所有数据项在 *cuboid* 中的逻辑位置 *new_tag*,然后根据 *new_tag* 值将所有数据项排序;这样,*new_tag* 值相同的数据项被放在一起;对具有相同 *new_tag* 值的数据进行聚集,得到 *cuboid* 的一个元组.如果该元组满足 Having 条件,则将它输出;否则,根据 Having 条件中聚集函数的反单调性,我们知道 D_{ij} 中产生该元组的数据在处理这个 cuboid 的后继 cuboid 时是不需要的,因此,将这些数据用 *cuboid* 标记之后入栈;如果当前 *cuboid* 没有后继 cuboid 时,则回溯到达下一个要处理的 cuboid,即 *cuboid′*,这时,要将栈内所带标记不是 *cuboid′* 的前缀的所有数据项弹出到数据区重新纳入考虑;重复上述过程,直到整个分支内的所有 cuboid 都被计算完毕为止.

下面的例子将说明上述过程.为了清楚起见,例子中的数据是以未压缩形式给出的.设输入数据集有属性 *A,B,C,D*,计算 *ICube* 时的处理过程如图 2 所示.我们首先处理第 1 个分支:先将输入数据划分成两个数据块,每个数据块在该分支的所有 cuboid 的公共前缀上具有相同的取值(如图 4(a)所示);之后,将第 1 个数据块读入内存,自底向上地计算该分支的所有 cuboid 的结果;然后将所有数据进行投影以去掉属性 *A*,排序、聚集、压缩之后写到临时文件.类似地处理其余数据块.当所有数据块都被处理之后,我们得到了 *ICube* 在第 1 个分支上的结果和一些压缩临时数据文件(如图 4(b)所示).将这些临时文件归并,得到处理下一个分支所需要的数据,然后就可以进行类似的处理直至完成整个计算.值得注意的是,为了减小磁盘 I/O 操作,临时文件的归并操作无须独立完成,只需在数据进入内存时直接完成归并,同时完成对数据的划分操作,但这样做将导致打开的文件句柄过多.因此,我们可以对临时文件进行部分归并,再利用上述策略.另一方面,由于计算过程中的投影操作,处理各个分支时的数据逐步减少.为了使数据减少的速度尽可能快,以提高计算效率,我们在压缩数据时,将线性化属性序列中的属性按其值域大小的递减顺序排序,即$|A_1|>|A_2|>\ldots>|A_n|$,这样就可以利用第 1 节给出的性质很方便地完成上述各个操作.

假设 *ICube* 的 Having 条件是 *count*(*)≥3 且内存中的数据如图 4(a)的第 2 个数据块所示,我们来考察算法如何计算图 2 中第 1 个分支中的所有 cuboid.我们首先根据内存中的元组数计算得到 *Agr*(*A*),此时,内存中数据区和栈的情况如图 5(a)所示;再用内存的数据计算 *Agr*(*AB*),此时,有些聚集值不满足 having 条件,产生这些聚集值的数据被压入栈,因为计算 *Agr*(*AB**)时不需要它们,数据区和栈的情况如图 5(b)所示.类似地,计算 *Agr*(*ABC*)的数据之后,数据区和栈的情况如图 5(c)所示.由于此时数据区为空,我们不再计算 *Agr*(*ABCD*),进行回溯直接计算 *Agr*(*ABD*).这时,我们将栈内标记不是 *ABD* 前缀的数据弹出,并让它们进入数据区.这个操作完成后,数据区和栈的情况如图 5(d)所示;完成 *Agr*(*ABD*)的计算后,数据区和栈的情况如图 5(e)所示.其他计算不再赘述.值得注意的是,数据区和栈无须独立使用各自的空间,只需一个辅助结构记录数据所带的标记和带有该标记的元组的个数,通过数据的移动,即可在数据区内实现栈操作.但为了叙述方便,我们将它分开加以叙述.

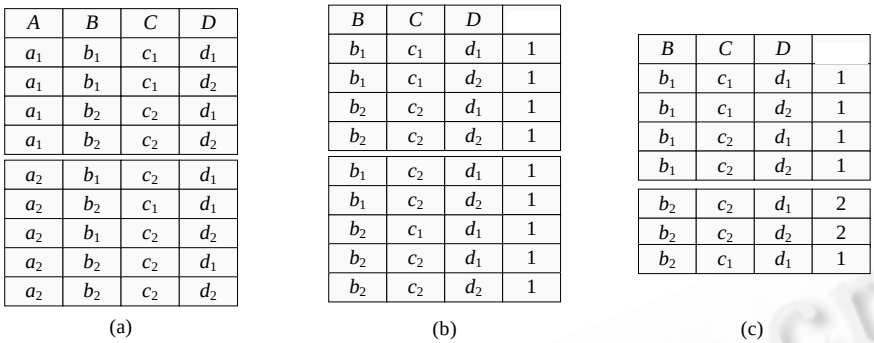

Fig.4 An example for the changing dataset in FIC (first-branch based Iceberg Cube algorithm) algorithm

图 4 FIC(first-branch based Iceberg Cube algorithm)算法中数据集变化过程示例

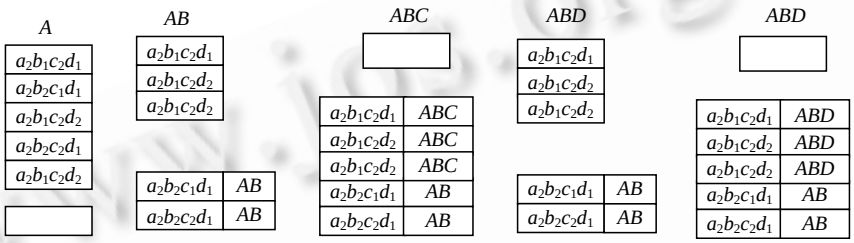

Fig.5 An example for the changing data in memory in FIC algorithm

图 5 FIC 算法的内存数据变化过程示例

设 $f_1,\ldots,f_h$ 是计算 *ICube* 过程中需要计算的全部聚集函数.由于此时我们无须利用聚集函数的分布性和代数性,因此,这些聚集函数包括 *ICube* 中需要输出的聚集函数、Having 条件中的聚集函数以及进行投影操作时需要计算的聚集函数.下面是算法的形式化描述.

算法 **1**(**FIC** 算法). *FIC*($D(A_1,\ldots,A_n,m_1,\ldots,m_k)$, $f_1,\ldots,f_h$,*condition*).

输入:$D(A_1,\ldots,A_n,m_1,\ldots,m_k)$, $f_1,\ldots,f_h$,Having 条件 *condition*;

输出:数据集 $D(A_1,\ldots,A_n,m_1,\ldots,m_k)$上的 Iceberg Cube 结果 *ICube*.

1. 构造所有一级分支 $T_1,\ldots,T_n$¬,D_1←$D(A_1,\ldots,A_n,m_1,\ldots,m_k)$;
2. 对 $T_1,\ldots,T_n$ 中的每个分支 T_i DO
3. WHILE (数据集 D_i 未处理完)
4. 读数据 D_i 的下一个块 D_{ij} 到内存,由后向映射计算每个数据的线性化值 *Tag*;
5. *S*←*root*(T_i);
6. WHILE (*S*!=NULL)
7. 将栈 *stack* 中标记不为 *S* 的前缀的数据弹出到数据区;
8. 根据每个数据的 *Tag* 值计算它在 *S* 下的 *new_tag* 值;
9. 根据 *new_tag* 值将数据区中的数据排序;
10. FOR 每个 *new_tag* 值 *NEW_TAG* //扫描数据区中的数据
    f_j←0 for *j*=1 to *h*;
11. 在满足 *new_tag*=*NEW_TAG* 的所有数据项上计算 $f_1,\ldots,f_h$;
12. IF (*condition*($f_1,\ldots,f_h$)=TRUE) THEN 输出〈*NEW_TAG*, $f_1,\ldots,f_h$ 中需输出的值〉;
    ELSE 将第 11 步中的所有数据项标记为 *S* 并压入 *stack* 的栈顶;
13. *S*←*Next*(T_i,*S*);
14. 生成有序数据集 $D(A_{i+1},\ldots,A_n)$,使其在属性 A_{i+1} 上有序,写入临时文件 F_{ij};

15. 合并第 11 步生成的临时文件 F_{ij},使其在属性 A_{i+1} 上有序,得到 D_{i+1}.

算法第 1 步无须显式地构造所有一级分支,只需给出该一级分支的根即可;第 5 步的 $root(T_i)$用来取得 T_i 的根;第 13 步根据分支 T_i 的当前 cuboid S,以深度优先方式构造下一个 cuboid,若 T_i 的所有 cuboid 均已被产生,它将返回空;第 4 步计算 *Tag* 值的过程由映射完全的压缩方法保证;第 8 步计算 *new_tag* 的过程由命题 2 给出.

2.3 一般算法

一级分支对应的数据块可能太大,不能完整地被内存容纳.因此,我们还需要对较大的一级分支进行细分,得到其他形式的一些分支,使得所有分支的数据块均能完整地容纳到内存中.我们将细分一级分支得到的分支称为二级分支,如图 3 所示.我们将由一级分支 T_i 细分得到的所有二级分支表示为 $T_{i1},\ldots,T_{ib_i}$.第 1 个二级分支 T_{i1} 中所有 cuboid 的公共前缀是线性化属性序列的前缀,计算该二级分支中 cuboid 的数据就是计算一级分支 T_i 的数据 D_i,这时,我们可以将 T_{i1} 等同于一级分支 T_i.处理其他二级分支 $T_{ij}(j>1)$的数据要在处理二级分支 T_{ij-1} 时生成.

按上述要求将 BUC 处理树划分成一些分支以后,我们逐次地处理每个一级分支 T_i.对于 T_i 的任意数据块 D_{ij},如果 D_{ij} 能够被内存容纳,则在 D_{ij} 上用与 FIC 算法同样的方法自底向上计算 T_i 中所有的 cuboid;否则,依次处理 T_i 的每个二级分支 T_{ik},以计算得到 T_i 中所有 cuboid 在 D_{ij} 上的聚集结果.D_{ij} 是处理 T_{i1} 所需的输入数据 $D_{ij}^{(1)}$, 处理 $T_{ik}(k>1)$所需的输入数据 $D_{ij}^{(k)}$ 需要在处理 T_{ik-1} 时生成.现在考虑在数据 $D_{ij}^{(k)}$ 上处理 T_{ik}:首先,根据 $root(T_{ik})$将 $D_{ij}^{(k)}$ 划分成一系列子数据块 $B_1,\ldots,B_l$,每个子数据块在 $root(T_{ik})$上具有相同的取值且能够被内存容纳;然后,依次处理每个数据子块 $B_t(1\le t<l)$.在内存中处理数据子块 B_t 时,先完整计算 $root(T_{ik})$在 B_t 上的聚集结果,写入到临时文件 F,同时将满足 Having 条件的元组输出;然后,自底向上地计算 T_{ik} 中其他 cuboid 的 Iceberg Cube 结果,同时生成处理 T_{ik+1} 所需的部分数据,写入临时文件 f_{kt} 中;如果 $t=1$,还要生成处理 T_{i+1} 所需的数据写入临时文件 F_{it} 中.当所有数据子块处理完之后,由 F 自顶向下计算所有可以由 $root(T_{ik})$的完整聚集结果计算得到的 cuboid 的 Iceberg Cube 结果,再合并所有 f_{kt},得到处理 T_{ik+1} 所需的数据 $D_{ij}^{(k+1)}$;如果 $k=1$,则还要合并 F_{it},得到处理 T_{i+1} 所需的部分数据,写入临时文件 F_i.当所有 D_{ij} 被处理完之后,合并 F_i,得到处理 T_{i+1} 所需的完整数据 D_{i+1}.一般算法形式化描述如下:

算法 2 (GIC(general Iceberg Cube algorithm)). 根据 BUC 处理树分支来计算 *Icube*.

输入:$D(A_1,\ldots,A_n,m_1,\ldots,m_k)$,having 条件 *condition*;

输出:数据集 $D(A_1,\ldots,A_n,m_1,\ldots,m_k)$上的 Iceberg cube 结果 *ICube*.

1. 构造所有一级分支 $T_1,\ldots,T_n,D_1\leftarrow D(A_1,\ldots,A_n,m_1,\ldots,m_k)$;
2. FOR i=1 TO n DO
3. WHILE (D_i 的数据未被处理完)
4. IF D_i 的数据块 D_{ij} 能被内存容纳 THEN 用 FIC 的方法计算 T_i 的每个 cuboid;
   ELSE
5. $D_{ij}^{(1)}\leftarrow D_{ij}$;
6. FOR T_i 的每个二级分支 T_{ik} DO
7. WHILE ($D_{ij}^{(k)}$ 未被处理完)
8. 读取下一个数据子块 B_t 放入内存;
9. 在 B_t 上计算 $root(T_{ik})$的完整聚集结果,写入文件 F,输出满足 *condition* 的元组;
10. IF (k=1) THEN 生成 T_{i+1} 对应的部分数据,写入 F_{it};
11. 用 FIC 中的方法计算 T_{ik} 中除 $root(T_{ik})$以外的 cuboid 的 *ICube* 结果;
12. 生成 T_{ik+1} 所需数据,并写入临时文件 $f_{k,t}$;
13. 合并第 11 步中生成 $f_{k,t}$,得到计算 T_{ik+1} 所需的数据 $D_{ij}^{(k+1)}$;

14. 根据 F 自顶向下计算可以由 $root(T_{ik})$生成的 Cuboid 结果并输出;

15. IF k=1 THEN 合并 F_{it},得到处理 T_{i+1}所需的部分数据 F_j;

16. 合并 F_j,得到处理 T_{i+1}所需的数据 D_{i+1}.

与 FIC 算法一样,GIC 算法的第 1 步无须显式地生成所有一级分支.第 4 步的实现很简单,就是将内存的数据区读满,同时检查它是否包含一个完整的数据块;内存中其他数据块的数据以后无须重读,保留在数据区直到被使用;如果内存中没有包含一个完整的数据块,我们根据内存中的数据即可确定第 1 个二级分支的根.因此,二级分支的划分无须显式地进行,计算过程可以直接产生所需的二级分支.算法第 8 步和第 14 步中用到了聚集函数的分布性或代数性,即在各个子数据块上分别计算 $root(T_{ik})$的聚集结果之后,可以得到该 cuboid 在整个数据块上的聚集结果;为说明第 14 步的过程,假设 $root(T_{ik})$=$ABCD$,根据 F 中 $ABCD$ 在数据块 D_{ij} 的完整聚集结果(未检查 Having 条件),我们可以依次计算 ABD,ACD,AB,AC,AD,A 等 cuboid 在数据块 D_{ij} 上的 Iceberg Cube 结果.第 9 步和第 11 步的计算过程与 FIC 算法中产生临时数据的过程相同.算法 GIC 的特点是,D_i 中数据量的大小随着 i 的增加迅速减小,且在每个数据块 D_{ij} 上依次处理二级分支时, $D_{ij}^{(k+1)}$ 中的数据量随着 k 的增加也迅速减小.

3 实 验

为了本节论述方便,我们将先用文献[1]中的方法在压缩 MDW 上计算完整的 Cube,然后用 having 条件对所有元组进行过滤来产生 Iceberg Cube 这种方法称为 Cube-ICube 策略.

我们实现了 FIC 算法和 GIC 算法,并研究了它的性能.我们将 FIC 算法和 GIC 算法的性能与 Cube-ICube 的性能进行了比较.所有的算法均用 C++实现,其运行环境是内存为 256M,主频为 2.4G 的奔腾机器,机器上运行的操作系统是 XP 专业版.在实验中,我们分别在维属性的值域大小为 15,12,10,8,7,5,5,5,3,2,2 的人工合成的 10 维数据集上运行第 2.2 节例子中所给出的 Iceberg Cube(只是将该维数扩展为 10).我们从 4 个方面考察了算法的性能:(1) 考察算法 FIC 和 GIC 处理各个分支所需的时间;(2) GIC 算法与 Cube-ICube 策略的性能比较;(3) Having 条件中最小支持度对算法执行时间的影响;(4) 考察数据集的稀疏程度对算法压缩数据的时间、计算时间和 I/O 时间的影响.

在实验 1 中,我们在每个维属性的值域上产生随机值分别生成了由 20 万条记录构成的数据集和由 200 万条记录构成的数据集,并以 10M 内存空间在前者上运行 FIC 算法而在后者上运行 GIC 算法.处理各个一级分支花费的时间如图 6 所示.我们看到,算法处理各个一级分支的时间迅速减小,这主要是因为投影操作使得各个分支的数据集迅速减小.

在实验 2 中,我们穷举每个维属性的取值生成了由约 1 350 万条记录构成的数据集,并以不同的内存设置运行了 GIC 算法和文献[1]中的 G-Cube 算法,然后在 G-Cube 的结果上用 Having 条件过滤得到 Iceberg Cube 的结果,统计总的时间开销(如图 7 所示).我们看到,GIC 算法的性能优于 Cube-ICube 策略.

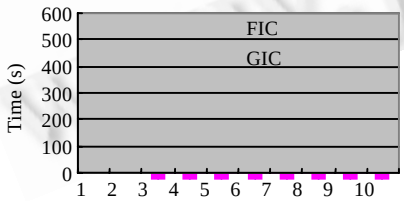

Fig.6 Time for the computation of each branch

图 6 处理各个分支的时间

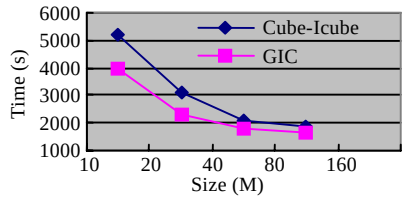

Fig.7 Comparison between GIC and Cube-ICube

图 7 GIC 与 Cube-ICube 的比较

在实验 3 中,我们用实验 2 中的数据集和 20M 内存空间运行算法 GIC,运行时修改 Having 条件中的最小支持度,总的时间开销如图 8 所示.我们看到,随着最小支持度的变化,Having 条件中聚集函数的反单调性能够有效剪除某些 cuboid 的计算.

在实验 4 中,我们生成了稀疏程度不同的 3 个数据集,其中密度为 1 的数据集由 1 350 条记录构成.在这 3

个数据集上分别运行 GIC 算法.我们统计了压缩数据的时间、计算时间和 I/O 时间占总时间的比例,结果如图 9 所示.在数据集较稀疏的情况下,数据集能够同时读入多个数据块,因此,主要开销在计算上,I/O 开销较小.当数据集的密度增大时,I/O 开销所占比例增大,而计算时间随之减小.在 3 个数据集上,压缩时间所占比例均不大.

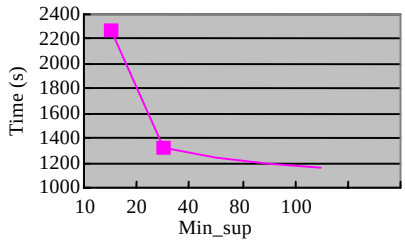

Fig.8 Impact of min_sup on run time

图 8 最小支持度对时间的影响

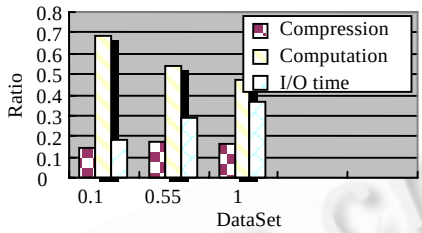

Fig.9 Impact of the density of datasets

图 9 数据集稀疏程度对算法的影响

4 结 论

Cube 操作是数据仓库中的重要操作.为了节省空间开销和计算代价,研究者们提出了一系列技术,其中包括 Iceberg Cube 操作算法和压缩技术.据我们所知,目前还没有在压缩多维数据仓库上计算 Iceberg Cube 的算法.

本文在讨论了多维数据仓库的压缩技术之后,给出了两个基于 BUC 思想的 IceBerg Cube 算法,以便在压缩数据仓库中有效地计算 Iceberg Cube.我们的算法利用 Having 条件中聚集函数的反单调性,避免了大量不必要的计算;算法的输入数据和中间临时数据均以压缩形式存在,计算过程无需显式的解压缩过程,且计算过程中数据量迅速减小.实验结果表明:该算法的性能优于先计算完整 Cube 再利用 Having 条件过滤产生 Iceberg Cube 这种方法,且 Having 条件中聚集函数能够有效剪裁掉一些 cuboid 的计算.

References:

[1] Gao H, Li JZ. Cube algorithms for very large data warehouses. Journal of Software, 2001,12(6):830−839 (in Chinese with English abstract).

[2] Li SE, Wang S. Research on closed data cube technology. Journal of Software, 2004,15(8):1165−1171 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/15/1165.htm

[3] Zhao Y, Desshpande PM, Naughton JF. An array-based algorithm for simultaneous multidimensional aggregations. In: Pechhan J, ed. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM Press, 1997. 159−170.

[4] Wang W, Feng JL, Lu HJ, Yu J. Condensed cube: An effective approach to reducing data cube size. In: Liu L, Reuter A, Whang KY, Zhang JJ, eds. Proc. of the 18th Int'l Conf. on Data Engineering (ICDE 2002). Los Alamitos: IEEE Computer Society Press, 2002. 155−165.

[5] Sismanis Y, Deligiannakis A, Roussopoulos N, Kotidis Y. Dwarf: Shrinking the PetaCube. In: Franklin MJ, Moon B, Ailamaki A, eds. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM Press, 2002. 464−475.

[6] Lakeshmanan LVS, Pei J, Han JW. Quotient cubes: How to summarize the semantics of a data cube. In: Bressan S, Chaudhri A, Lee ML, Yu JX, Lacroix Z, eds. Proc. of the 28th Int'l Conf. on Very Large Data Bases. San Fransisco: Morgan Kaufmann Publishers, 2002. 476−487.

[7] Beyer KS, Ramakrishan R. Bottom-Up computation of sparse and Iceberg Cubes. In: Delis A, Faloutsos C, Ghandeharizadeh S, eds. Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIMOD'99). New York: ACM Press, 1999. 359−370.

[8] Han JW, Pei J, Dong GZ, Wang K. Efficient computation of Iceberg Cubes with complex measures. In: Aref WG, ed. Proc. of the ACM SIGMOD Conf. 2001. New York: ACM Press, 2001. 1−12.

[9] Xin D, Han JW, Li XL, Wah BW. Star-Cubing: Computing Iceberg Cubes by top-down and bottom-up integration. In: Freytag JC, Lockemann PC, Abiteboul S, Carey MJ, Selinger PG, Heuer A, eds. Proc. of the 29th Int'l Conf. on Very Large Data Bases. San Fransisco: Morgan Kaufmann Publishers, 2003. 476−487.

[10] Kong YF, Chen H. GFSC—The storage of free cube based on graph structure. Journal of Computer Research and Development, 2004,41(10):1652−1660 (in Chinese with English abstract).

[11] Li JZ, Rotem D, Srivastava J. Aggregation algorithms for very large compressed data warehouses. In: Atkinson MP, Orlowska ME, eds. Proc. of the 29th Int'l Conf. on Very Large Data Bases. Mumbai: Morgan Kaufmann Publishers, 1999. 651−662.

[12] Feng JH, Jiang XD, Zhou LZ. An improved multi-dimensional storage structure for data warehousing. Journal of Software, 2002, 13(8):1423−1429 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/13/1423.pdf

[13] Eggers S, Shoshani A. Efficient access of compressed data. In: Vijayaraman T, *et al.*, eds. Proc. of the 6th Int'l Conf. on Very Large Data Bases. Montreal: IEEE computer Society Press, 1980. 205−211.

附中文参考文献:

[1] 高宏,李建中.超大型压缩数据仓库上的 Cube 算法.软件学报,2001,12(6):830−839.

[2] 李盛恩,王珊.封闭数据立方体技术研究.软件学报,2004,15(8):1165−1171. http://www.jos.org.cn/1000-9825/15/1165.htm

[10] 孔延凡,陈红.GFSC——一种基于图结构的 Free Cube 存储方法.计算机研究与发展,2004,41(10):1652−1660.

[12] 冯建华,蒋旭东,周立柱.用于数据仓储的一种改进的多维存储结构.软件学报,2002,13(8):1423−1429. http://www.jos.org.cn/1000-9825/13/1423.pdf

骆吉洲(1975－),男,重庆铜梁人,博士,讲师,主要研究领域为压缩数据库技术.

赵锴(1981－),男,硕士,主要研究领域为压缩数据库技术.

李建中(1950－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为数据库系统实现技术,数据仓库,半结构化数据,传感器网络,压缩数据库技术,Web 数据集成,数据挖掘,计算生物学.