超大型压缩数据仓库上的CUBE算法*

高 宏, 李建中

(哈尔滨工业大学 计算机科学与工程系,黑龙江 哈尔滨 150001)

E-mail: lijz@banner.hl.cninfo.net

摘要: 数据压缩是提高多维数据仓库性能的重要途径,联机分析处理是数据仓库上的主要应用,Cube操作是联机分析处理中最常用的操作之一.压缩多维数据仓库上的Cube算法的研究是数据库界面临的具有挑战性的重要任务.近年来,人们在Cube算法方面开展了大量工作,但却很少涉及多维数据仓库和压缩多维数据仓库.到目前为止,只有一篇论文提出了一种压缩多维数据仓库上的Cube算法.在深入研究压缩数据仓库上的Cube算法的基础上,提出了产生优化Cube计算计划的启发式算法和3个压缩多维数据仓库上的Cube算法.所提出的Cube算法直接在压缩数据上执行Cube操作,无须反压缩,提高了Cube的处理速度.这些算法适用于一大类常用的数据压缩方法.对这些算法的I/O时间复杂性和CPU时间复杂性进行了细致分析,并对这3种算法和其他Cube算法进行了大量的实验研究,对性能进行了分析比较.理论分析和实验结果都表明,所提出的Cube算法的性能高于目前所有的Cube算法的性能.

关键词: 数据仓库;压缩的数据仓库;OLAP (on line analysis processing);Cube

中图法分类号: TP311 **文献标识码**: A

近年来,多维数据仓库引起了人们的重视.一个多维数据仓库由多个多维数据集合组成.一个d-维数据集合可以表示为$R(D_1,D_2,\ldots,D_d;M_1,M_2,\ldots,M_k)$,其中$D_i$是维属性,$M_j$是度量属性.多维数据仓库系统使用多维数组存储多维数据集合.多维数据集合的维属性值无须存储,仅用作数组的维索引,以确定多维数据集合中每个数据项在多维数组中的位置.多维数组仅存储度量属性值.多维数据集合通常十分稀疏,使得数据压缩技术成为存储和管理多维数据仓库的重要工具.导致多维数据仓库稀疏的主要原因是:由多个维属性值域的笛卡尔积构成的多维空间具有固有的稀疏性.例如,由于低于40岁的人很少得癌症,癌症数据集合C(年龄,性别,癌症种类;癌症人数)构成的三维数据空间是十分稀疏的.当年龄小于40时,相应子空间上的数据项非常稀少.目前,人们已经提出了很多可用于多维数据仓库数据压缩技术[1,2].

联机分析处理(on line analysis processing,简称OLAP)是数据仓库的主要应用.Cube[3]是常用的OLAP操作.设计有效的Cube算法是建立高性能多维数据仓库系统的关键.由于多维数据仓库一般都需要压缩,研究直接在压缩数据上执行Cube操作、无须反压缩的Cube算法是数据库界面临的挑战性任务.自从Gray等人提出Cube操作[3]以来,人们在关系型数据仓库的Cube算法方面开展了大量研究工作,提出了很多Cube算法,但是基于多维数据仓库和压缩多维数据仓库的Cube算法的研究还十分少见.据我们所知,到目前为止仅提出了一种基于压缩多维数据仓库的Cube实现算法[2].而该算法也仅适用于一种称为chunk-offset的压缩比较低的数据压缩方法.该算法的性能与多维数据集合的维序等因素密切相关,并且需要大容量的主存空间.当多维数据集合中维属性个数较少时,该算法具有很高的效率.反之,当多维数据集合中维属性个数较多时,该算法

* 收稿日期: 2000-09-04; 修改日期: 2001-03-15

基金项目: 国家自然科学基金资助项目(69873014);国家重点基础研究发展规划973资助项目(G1999032704)

作者简介: 高宏(1956—),女,河北故城人,博士,副教授,主要研究领域为数据库,数据仓库;李建中(1950—),男,黑龙江哈尔滨人,教授,博士生导师,主要研究领域为数据库,并行计算.

的效率很低,其原因是内存容量不能满足算法的要求,算法需要执行大量I/O操作,I/O时间复杂性迅速增长.

本文深入研究了压缩多维数据仓库上的Cube算法,提出了3个压缩多维数据仓库上的Cube算法.这些算法直接在压缩多维数据集合上执行Cube操作,不需要在执行Cube操作之前对其进行反压缩.这些算法适用于一大类常用的数据压缩方法.本文对这3个算法的I/O时间复杂性和CPU时间复杂性进行了细致的分析.另外,还对这3种算法和其他Cube算法进行了大量的实验研究,对算法的性能进行了分析和比较.理论分析和实验结果都表明,本文提出的Cube算法的性能高于目前所有其他Cube算法的性能.

1 超大型数据仓库的压缩

设$R(D_1,\ldots,D_d;M_1,M_2,\ldots,M_m)$是一个多维数据集合.我们使用如下3个步骤来压缩存储$R(D_1,\ldots,D_d;M_1,M_2,\ldots,M_m)$.

步骤1. 使用多维数组存储多维数据集合.

设$R(D_1,\ldots,D_d;M_1,M_2,\ldots,M_m)$第$i$维属性$D_i$的值域的基数为$d_i$.为了存储$R(D_1,\ldots,D_d;M_1,M_2,\ldots,M_m)$,我们建立了$m$个$d$-维数组$A_1,A_2,\ldots,A_m$.$R$的维属性$D_i$的值域是这$m$个数组第$i$维索引的值域.度量属性$M_i$的值存储在$A_i$中.显然,$R$的维属性值不需要被存储,仅用于确定度量属性在数组中的位置.

步骤2. 多维数组的线性化.

我们可以通过数组线性化函数把n-维数组变换为一个线性化数组.数组线性化函数定义如下:

定义1. 假定用$\{0,1,\ldots,d_i-1\}$表示多维数组R的第i个维属性$(1\leqslant i\leqslant n)$的值域,$R$的数组线性化函数定义为:$LINEAR(x_1,x_2,\ldots,x_n)=x_1d_2d_3\ldots d_n+x_2d_3d_4\ldots d_n+\ldots+x_{n-1}d_n+x_n$.

对于任意的维属性值组合$(i_1,i_2,\ldots,i_n)$,它所对应的度量属性值在线性化数组中的位置$P(i_1,i_2,\ldots,i_n)$是$LINEAR(i_1,i_2,\ldots,i_n)=i_1d_2d_3\ldots d_n+i_2d_3d_4\ldots d_n+\ldots+i_{n-1}d_n+i_n$.反过来,也可以利用逆数组线性化函数,根据一个度量属性值在线性化数组中的位置P,推算出其各维的索引值$(i_1,i_2,\ldots,i_n)$.

定义2. 令R是使用定义1中的数组线性化函数所得到的线性化数组,该数组中位置为P的度量属性值的各维索引值$(i_1,i_2,\ldots,i_n)$可以通过如下函数得到:$R-LINEAR(P)=(i_1,i_2,\ldots,i_n)$,其中$i_n=P \bmod d_n$,$i_{n-1}=\lfloor P/d_n\rfloor \bmod d_{n-1}$,$\ldots$,$i_2=[[\ldots[[P/d_n]/d_{n-1}]\ldots]/d_3] \bmod d_2$,$i_1=[[\ldots[[P/d_n]/d_{n-1}]\ldots]/d_2]$,$[X]$为$X$的整数部分.

步骤3. 多维数组的压缩.

使用线性化数组存储稀疏多维数据集合,数组中必然包含很多空单元,需要压缩.数据压缩既可以减少存储空间,又能够提高系统性能.近十几年来,数据库研究者提出了一些满足数据库和数据仓库要求的数据压缩方法.这些压缩方法一般都具有两个映射.一个是向前映射(forward mapping).另一个是向后映射(backward mapping).向前映射根据给定数据项在原始数据集合中的位置计算该数据项在压缩数据集合中的位置.向后映射根据给定数据项在压缩数据集合中的位置推算出该数据项在原始数据集合中的位置.一个数据压缩方法是映射完全的当且仅当它同时具有向前映射和向后映射.许多数据压缩方法都是映射完全的[1,2].本文提出的Cube算法适用于所有映射完全的数据压缩方法.为了具体而且易懂,我们使用header compression方法来压缩线性化多维数组.Header compression方法是一种映射完全的数据压缩方法.它把一个线性化多维数组文件压

缩成为一个物理文件和一个 header. 有关 header compression 方法的细节参见文献[1].

2 Cube 算法

Cube 算法包括两个阶段:(1) 生成 Cube 的计算计划;(2) 按照 Cube 计算计划完成压缩数据集合上的 Cube 的计算. 不失一般性,在下面的讨论中,我们假定每个多维数据集合仅具有一个度量属性,并存储在第 1 节所介绍的由 header compression 方法压缩的线性化多维数组中.

定义 3. 设 $R(D_1,D_2,\ldots,D_d;M)$是一个多维数据集合. R 上的聚集操作定义为 Group-by(R,β,F),其中 $\beta=\{D_{i_1},\ldots,D_{i_n}\}\subseteq\{D_1,D_2,\ldots,D_d\}$称为聚集属性,$F$ 是定义在 M 的幂集合上的函数(称为聚集函数). 令

$$S_{b_1,\ldots,b_n}=\{z\mid(d_1,\ldots,d_d;z)\in R(D_1,\ldots,D_d;M),d_{i_1}=b_1,\ldots,d_{i_n}=b_n\},$$

Group-by(R,β,F) 的结果为 $R_\beta(\beta;F(M))=\{(b_1,\ldots,b_n;F(S_{b_1,\ldots,b_n}))\mid\exists\ (d_1,\ldots,d_d;z)\in R(D_1,\ldots,D_d;M)\}$.

定义 4. 设 $R(D_1,D_2,\ldots,D_d;M)$是一个多维数据集合. R 上的 **Cube** 操作定义为 Cube(R,α,F),其中 $\alpha\subseteq\{D_1,\ldots,D_d\}$称为 Cube 属性集合,$F$ 是定义在 M 的幂集合上的函数(称为聚集函数). Cube(R,α,F)的结果是 $2^{|\alpha|}$ 个多维数据集合$\{R_\beta(\beta;F(M))\mid\forall\ \beta\subseteq\alpha,R_\beta(\beta;F(M))=$Group-by$(R,\beta,F)\}$. $\forall\ \beta\subseteq\alpha,R_\beta(\beta;F(M))$称为一个 **Cuboid**,简记为 β.

不失一般性,以下假设 Cube(R,α,F)操作的 α 包含了 R 的所有维属性,F 为求和函数.

定义 5. 设 $L=\{c_1,c_2,\ldots,c_m\}$是数据集合 $R(D_1,D_2,\ldots,D_n;M)$关于 Cube 操作 Cube(R,α,F)的 cuboid 集合. 如果 $c_i\supseteq c_j$ 而且 $|c_i|-|c_j|=1$,我们说 c_i 和 c_j 满足直接计算关系(即 c_j 可以由 c_i 直接计算出来,表示为 $c_i\rightarrow c_j$). 如果 $c_i\rightarrow c_j$,我们称 c_i 是 c_j 的直接前辈或 c_j 是 c_i 的直接后裔. $\langle L,\rightarrow\rangle$ 称为 Cube 操作 Cube(R,α,F)的 **Cube** 代数格.

如果$\langle L,\rightarrow\rangle$是 Cube 操作 Cube$(R,\alpha,F)$的 Cube 代数格. $\langle L,\rightarrow\rangle$可以由一个有向图 $T=(V,E)$表示,其中 $V=L,E=\{(c_i,c_j)\mid c_i,c_j\in L,c_i\rightarrow c_j\}$. T 称为 Cube(R,α,F)的代数格图.

2.1 Cube 计算计划的生成

给定一个 n-维数据集合 $R(D_1,D_2,\ldots,D_n;M)$,R 上的 Cube 操作需要计算 2^n 个 Cuboid. Cuboid 的计算顺序以及如何计算每个 cuboid 对 Cube 操作的效率具有重大的影响. 在开始执行 Cube 操作之前,需确定 cuboid 的计算顺序和计算方法,即产生一个 **Cube** 计算计划. 如何生成一个优化的 Cube 计算计划是 Cube 算法首先需要解决的问题. Agarwal 等人曾提出过一个 Cube 计算计划的生成算法[2]. 但是,他们的算法存在以下问题:(1) 没有充分考虑利用前缀和后缀来减少排序次数,降低计算复杂性;(2) 通过排序父结点来计算子结点,增加了排序代价,其原因是父结点的数据量大于子结点的数据量. 为了解决这些问题,我们给出一种新的 Cube 计算计划生成算法 Plan-Generation. 该算法是一个启发式算法. 它以 Cube 操作的代数格图为输入,使用一组启发式规则,为 Cube 代数格中的每个 cuboid 选择与之具有前缀、部分前缀或后缀关系且计算代价最小的直接前辈,确定各 cuboid 的计算顺序,生成一个高效率的 Cube 计算计划,使得整个 Cube 的计算代价最小. Plan Generation 算法分两个阶段产生优化的 Cube 计算计划. 第 1 阶段是优化 Cuboid 树生成阶段. 第 2 阶段是 Cuboid 树拆分阶段.

第 1 阶段. 生成优化 **Cuboid** 树.

定义 6. 设 Cube(R,α,F)是一个 Cube 操作,$\langle L,\rightarrow\rangle$是 Cube$(R,\alpha,F)$的 Cube 代数格,$G=(V,E)$是 Cube$(R,\alpha,F)$的代数格图. Cube$(R,\alpha,F)$的 **Cuboid** 树是一个满足下列条件的有向树

$T=(V,E')$:① T 的根是由 α 的所有维属性构成的Cuboid;② $E'\subseteq E$.

Cube(R,α,F)的Cubiod树确定了Cube(R,α,F)的所有Cuboid的计算顺序和计算方法,即从树根到叶的顺序逐级计算树上的每个Cuboid,而且如果Cuboid$_1\rightarrow$Cuboid$_2$,则使用Cuboid$_1$计算Cuboid$_2$.一个Cube操作具有多个Cuboid树.我们的目标是选择一棵优化Cuboid树.下面我们给出优化Cuboid树的定义.

定义7. 设Cube(R,α,F)是一个Cube操作,Ψ是Cube(R,α,F)的所有Cuboid树集合,Cost(T)是按照Ψ中的Cubiod树T规定的顺序和计算方法计算T上的所有Cuboid所需要的代价,可以是计算时间,也可以是其他测度.Cube(R,α,F)的优化**Cuboid**树是Ψ中满足如下条件的Cuboid树$T_{\min}$:Cost$(T_{\min})=\min\{\text{Cost}(T)\mid T\in\Psi\}$.

定义8. 对任意两个Cuboid c_i和c_j,如果c_i是c_j的前缀或c_j是c_i的前缀,则称c_i与c_j之间具有前缀关系;如果c_i与c_j之间存在公共前缀,但不具有前缀关系,则称c_i与c_j之间具有部分前缀关系.如果c_i是c_j的后缀或c_j是c_i的后缀,则称c_i与c_j之间具有后缀关系.

下面我们简单讨论一下如何利用这3种关系来计算Cuboid.第3节将详细讨论这些算法.

(1) 利用后缀关系计算**Cuboid**.如果Cuboid $c=d_2d_3\ldots d_k$,Cuboid $c'=d_1d_2\ldots d_k$,即c和c'之间具有后缀关系.我们可由c'如下计算c:首先把c'按d_1的值划分为最多$|d_1|$个有序段(每段内的$d_2d_3\ldots d_k$组合值自然有序),然后合并(同时执行聚集计算)这$|d_1|$个有序段即可完成c的计算,避免了其他算法所需的c'排序,节省了计算时间.

(2) 利用部分前缀关系计算**Cuboid**.如果Cuboid $c_i=d_{i_1}d_{i_2}\ldots d_{i_k}$与Cuboid $c_j=d_{j_1}d_{j_2}\ldots d_{j_t}$之间具有部分前缀关系,$t>k$,$d_{i_1}d_{i_2}\ldots d_{i_r}$为$c_j$与$c_i$之间的公共前缀,且在Cube代数格中$c_j$是$c_i$的前辈,我们可由$c_j$如下计算$c_i$:首先按公共前缀$d_{i_1}d_{i_2}\ldots d_{i_r}$的组合值把$c_j$划分为多段;然后在每段内执行聚集计算,聚集属性为$d_{i_{r+1}}d_{i_{r+2}}\ldots d_{i_k}$;最后按$d_{i_{r+1}}d_{i_{r+2}}\ldots d_{i_k}$的组合值排序各段的聚集结果即可得到$c_i$.该计算方法避免了其他算法所需的整个$c_j$的排序.

(3) 利用前缀关系计算**Cuboid**.如果Cuboid $c_i=d_{i_1}d_{i_2}\ldots d_{i_k}$与Cuboid $c_j=d_{i_1}d_{i_2}\ldots d_{i_k}d_{i_{k+1}}$之间具有前缀关系,且在Cube代数格中$c_j$是$c_i$的前辈,我们可由$c_j$如下计算$c_i$:顺序扫描$c_j$一遍,以$d_{i_1}d_{i_2}\ldots d_{i_k}$为聚集属性直接执行聚集计算,即可得到$c_i$.这种方法不需要排序.

根据以上讨论,我们可以由Cuboid之间的前/后缀关系给出如下启发式规则:

规则1. 若Cuboid $c_1,c_2,\ldots,c_k$是Cuboid c的直接前辈,则优先选择与c具有前缀关系的c_i来计算c,$1\leqslant i\leqslant k$.如果$c_1,c_2,\ldots,c_k$中有多个Cuboid与c具有前缀关系,则选择使用其中具有最小数据量的Cuboid来计算c.

规则2. 若Cuboid $c_1,c_2,\ldots,c_k$是Cuboid c的直接前辈,且$c_1,c_2,\ldots,c_k$都不与c具有前缀关系,则优先选择与c具有后缀关系的c_i来计算c,$1\leqslant i\leqslant k$.若$c_1,c_2,\ldots,c_k$中有多个Cuboid与c有后缀关系,则选择使用其中具有最小数据量的Cuboid来计算c.

规则3. 若Cuboid $c_1,c_2,\ldots,c_k$是Cuboid c的直接前辈,且$c_1,c_2,\ldots,c_k$都不与c具有前缀关系或后缀关系,则选择与c具有部分前缀关系的c_i来计算c,$1\leqslant i\leqslant k$.若$c_1,c_2,\ldots,c_k$中有多个Cuboid与c有部分前缀关系且公共前缀中属性个数相等,则选择其中具有最小数据量的Cuboid来计算c.若$c_1,c_2,\ldots,c_k$中有多个Cuboid与c有部分前缀关系且数据量相等,则选择其中与c具有最大公共前缀的Cuboid来计算c(可以减少排序时间).若$c_1,c_2,\ldots,c_k$中有多个Cuboid与c有部分前缀关系且数据量和公共前缀大小不同,则选择其中具有最小数据量的c_i和具有最大部分前缀的c_j,使用c_i和c_j计算c的代价cost(c_i)和cost(c_j),若cost$(c_i)>$cost(c_j),则使用c_j计算c,否则使用c_i计算c.

定义9. 设$G=(V,E)$是Cube(R,α,F)操作的代数格图.V中任意一个结点都可以视为R的维属性序列.如果G满足对于任意$(c_1,c_2)\in E$,c_2是c_1的子序列,则称G是Cube(R,α,F)的规范化代数格图.

任意一个代数格图都可以通过调换各 Cuboid 的属性顺序变换为规范化代数格图. 不难证明,在一个规范化代数格图 $G=(V,E)$ 中,如果 $(c_1,c_2)\in E$,则 c_1 和 c_2 必然满足前缀关系、后缀关系和部分前缀关系之一. 于是,上述 3 个规则是产生优化 Cuboid 树的有效启发式规则. 下面给出算法 Plan-Generation-Heuristic 的定义:

Plan Generation-Heuristic(G)
输入:Cube(R,α,F)的 Cube 代数格对应的有向图 $G=(V,E)$.
输出:优化 Cubiod 树 Min$T=(V,E')$.
(1) Min$T=G$;
(2) FOR $i=1$ TO n DO /* 设 G 具有 n 层,从根(即 α)到叶(即 ALL)编号为 1 到 n */
(3) FOR G 第 i 层上的每一个结点 j DO
(4) 确定 j 的所有父结点集合 Parent(j)=$\{p_1,p_2,\ldots,p_m\}$;
(5) 使用规则 1、2、3 从 Parent(j)中选择一个用来计算 j 的 p_i;
(6) 从 Min$T$ 中删除边集合$\{(p,j)\mid p\neq p_i\}$.

Plan-Generation-Heuristic 算法的时间复杂性为 $O(|V|)$,$|V|$ 是 V 中的结点个数.

第 2 阶段. 拆分 **Cuboid** 树.

这一阶段进一步加工第 1 阶段生成的优化 Cuboid 树,把优化 Cuboid 树拆分为若干 Cuboid 子树,形成一个森林,即 Cube 计算计划,使得对于森林中的每棵子树,只需扫描一遍该树的根结点所对应的 Cuboid(该 Cuboid 已经计算出)就能以流水线方式计算出该树上所有的 Cuboid. 森林中的每一个子树具有如下特点:叶结点与根具有前缀关系、部分前缀关系或后缀关系,而除根以外的所有内结点都与根有前缀关系. Cuboid 树的拆分算法描述如下:

Split-Cuboid-Tree(T)
输入:优化 cuboid 树 $T=(V,E)$.
输出:一组 cuboid 子树集合 $Q=\{T_1,T_2,\ldots,T_k\}$.
(1) Q=Null;
(2) FOR $i=1$ TO n DO /* 设 T 具有 n 层,从根(即 α)到叶(即 ALL)编号为 1 到 n */
(3) FOR T 第 i 层上的每一个结点 c DO
(4) IF 存在 c 的某个子结点 c' 未作过标记
(5) THEN 以 c 为根构造子树 T_c,将 c 的所有未标记子结点插入 T_c 中作为 c 的子结点,同时标记这些结点"处理过";
(6) 将所有与 c 有前缀关系且无标记的 c 的后裔按在 T 中父子关系插入到子树 T_c 中,并标记这些结点"处理过";
(7) $Q=Q\cup\{T_c\}$;
(8) Return Q.

在拆分后得到的一组 cuboid 子树中,我们称以优化 cuboid 树的根为根的子树为最大 cuboid 子树.

2.2 Cube 计算方法

设一个 cuboid 子树的根为 c_{root},其维属性顺序为 $D_1D_2\ldots D_k$,且维 D_i 的基数为 $d_i(1\leqslant i\leqslant k)$,$c_{root}[l]$表示 c_{root}对应的多维数组中逻辑位置为 l 的元素. 如果 cuboid c_j 与 c_{root}具有前缀关系,即 c_j 的维属性为 $D_1\ldots D_p(1\leqslant p\leqslant k)$,则在 c_j 的多维数组中与 $c_{root}[l]$对应的元素的逻辑位置为

$$l_{com\text{-}pref}=\lfloor l/(d_{p+1}*\ldots*d_k)\rfloor. \tag{1}$$

如果 cuboid c_j 与 c_{root}具有部分前缀关系,c_j 的维序为 $D_1\ldots D_qD_{q+2}\ldots D_p$,$c_j$ 与 c_{root}的公共前缀为 $D_1\ldots D_q(1\leqslant q\leqslant k-1)$,则在 c_j 的多维数组中与 $c_{root}[l]$对应的元素的逻辑位置为

$$l_{pref}=(\lfloor l/(d_{q+1}*\ldots*d_k)\rfloor)*(d_{q+2}*\ldots*d_k)+l \bmod (d_{q+2}*\ldots*d_k). \tag{2}$$

如果 cuboid c_j 与 c_{root}具有后缀关系,即 c_j 的维序为 $D_2\ldots D_k$,则 l 对应的数据项在 c_j 的多维数组中

的逻辑位置为

$$l_{out}=l \bmod (d_r * \ldots * d_k). \tag{3}$$

2.2.1 G-Cube算法

G-Cube算法是一个在任何条件下都可以使用的算法.G-Cube算法以Cube计算计划为输入,逐个计算每个cuboid子树(以下简称子树)上的所有cubiod.如前所述,Cube计算计划的每个子树中的cuboid必与其根(已经计算完毕)存在前缀、部分前缀或后缀关系.G-Cube分两个阶段计算每个子树.在第1阶段,对于了树中任意一个与根存在前缀关系的cuboid,算法直接计算出其结果;对于子树中任意一个与根存在部分前缀或后缀关系的cuboid,算法计算出其中间结果,并且分别根据部分前缀或后缀关系对中间结果进行分段,其他工作在第2阶段内完成.在第2阶段,算法处理第1阶段得到的中间结果.如果中间结果对应的cuboid与根存在部分前缀关系,则排序中间结果的各个分段;如果中间结果对应的cuboid与根存在后缀关系,则合并中间结果各个分段.如果一个cuboid子树包含k个cuboid,G-Cube算法在第1阶段需要k个内存缓冲区,其中一个用作根cuboid的输入缓冲区(记作Buff_in),其余$k-1$个用作欲计算的各$k-1$个cuboid的输出缓冲区(记作Buff_out[i],$1\leqslant i\leqslant k-1$).

我们用一个实例来说明G-Cube的计算过程.假定多维数据集合为$R=\{A,B,C;M\}$.由算法Plan-Generation-Heuristic和Split Cuboid Tree生成的Cube计算计划包含cuboid子树T_1和T_2,如图1所示.G-Cube算法首先扫描ABC一遍,计算出T_1上的所有cuboid,即AB,AC,BC,A.然后,G-Cube算法扫描BC一遍,计算出B,C,ALL.图2给出了R的数据实例和G-Cube算法计算子树T_1的过程.为了说明算法的基本思想,图2给出了各cuboid中每个数据项的逻辑位置和维属性值,忽略了数据压缩的细节.实际上,各cuboid的线性化数组中只存储度量属性,并且使用Header数据压缩方法进行了压缩.对于ABC中的每一个数据项V,算法使用backward-mapping及前面给出的公式(1)、(2)或(3),推算出AC,AB,BC,A的线性化数组中与V对应的元素的逻辑位置,并对逻辑位置相同的度量值做聚集计算(这里是求和).例如,对于ABC中逻辑位置为0、1、4的元素,由于对应的维属性A和B的值都为0,所以它们在AB中的逻辑位置都为0.这3个元素形成了AB中逻辑位置为0的一个元素,即图中的"0,0,5",度量属性值5是这3个数据项的聚集结果.同理,ABC中逻辑位置为0、5的数据项产生了AC中逻辑位置为0的元素"0,0,5".由于AC与ABC之间存在部分前缀关系,算法根据AC和ABC的公共前缀A的值把AC的中间结果划分为3段,即图2中的段1、段2和段3.算法的第2阶段对每个段进行排序,即得到最终的AC结果.由于BC与ABC之间存在后缀关系,算法根据A的值把BC的中间结果划分为3段,即图2中的段1、段2和段3.每个段内的数据项是有序的.第2阶段合并这3个数据段即得到最后的BC结果.G-Cube算法的详细定义如下:

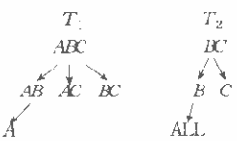


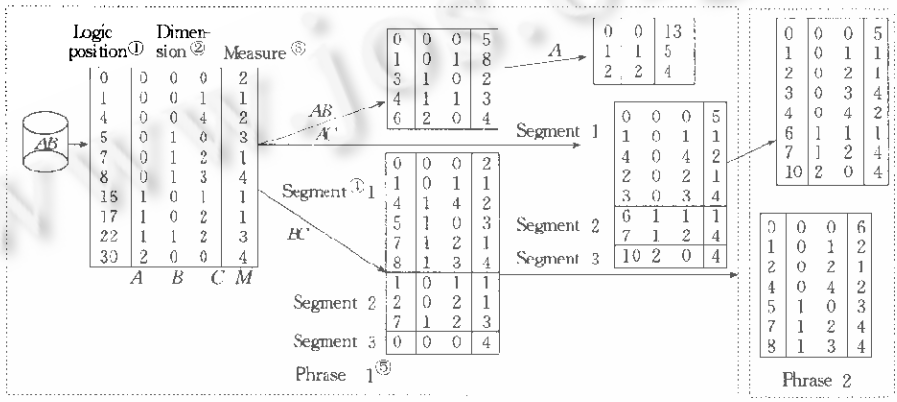
Fig. 1
图1

输入:压缩的多维数据集合、Header文件、Cube计算计划α.
输出:Cube结果(一组压缩的多维数组).

```
(1)  WHILE α不空 DO
(2)    从α中选择一个根cuboid已经被计算出来的cuboid子树T; /* 设T中除根以外的cuboid的个数为s */
(3)    WHILE T的根root仍有数据未被读入过主存缓冲区Buff_in DO /* 第1阶段 */
(4)      读入root的压缩多维数组中部分未被读入过主存缓冲区的数据到Buff_in;
(5)      FOR Buff_in中的每一个数据项v DO
(6)        使用backward-mapping恢复v在root对应的多维数组中的逻辑位置l;
```

(7) 使用公式(1)~(3)计算 root 的后裔 cuboid 对应的多维数组中与 v 对应数据项的逻辑位置 $c_1,c_2,\ldots,c_s$;
(8) FOR $r=1$ TO s DO
(9) IF Buff_out[r]中存在逻辑位置为 c_r 的数据项(c_r,x) THEN 修改(c_r,x)为$(c_r,x+v)$; /* 聚集函数为求和 */
(10) ELSE (c_r,v)写入 Buff_out[r]; /* 必要时,按照公共前缀属性或非公共后缀属性加分段标记 */
(11) IF Buff_out[r]满 THEN 将 Buff_out[r]内容写入磁盘文件;
(12) FOR 每个与 root 具有部分前缀、后缀关系的 cuboid C DO /* 第 2 阶段 */
(13) 排序或合并在第 1 阶段形成的 C 的各个段,同时去掉逻辑位置,并生成相应的 Header 文件;
(14) FOR 每个与 root 具有前缀关系的 cuboid DO 去掉逻辑位置,生成相应的 Header 文件;
(15) $\alpha=\alpha-\{T\}$.

限于篇幅,这里省略了 G-Cube 算法的复杂性分析,有兴趣的读者可以参见文献[6].



①逻辑位置,②维,③度量,④段,⑤第1阶段.

Fig. 2
图2

2.2.2 M Cube 算法

M-Cube 算法是一个主存算法.它所需要的主存空间小于 Cube 计算计划中最大和任意一个 Cuboid 子树中所有 Cuboid 的存储容量之和.M-Cube 算法的基本思想是:首先扫描输入压缩多维数组,按前缀、部分前缀、后缀关系计算最大 cuboid 子树中的所有 cuboid,主存中保留了最大 cuboid 子树中所有非根 cuboid 的计算结果;然后,把非其他子树根的 cuboid 保存到磁盘,释放其占用的主存缓冲区;其次,用主存缓冲区中的 cuboid 继续计算其他子树;这样,每计算完一棵子树,就将所计算出的 cuboid 中那些非其他子树根的 cuboid 写回到磁盘,并释放内存;如此下去,直到完成所有 cuboid 的计算.M-Cube 算法的详细定义如下:

输入:压缩的多维数据集合、Header 文件、Cube 计算计划 α.
输出:Cube 结果(一组压缩的多维数组).
(1) 从 α 中选择最大 cuboid 子树 T;
(2) WHILE T 的根 root 仍有数据未被读入过主存缓冲区 root_Buff DO
(3) 读入 root 的压缩多维数组中部分未被读入过主存缓冲区的数据到 root_Buff;
(4) FOR root_Buff 中的每一个数据项 v DO
(5) 使用 backward-mapping 恢复 v 在根对应的多维数组中的逻辑位置 l;
(6) 使用公式(1)~(3)计算与 root 具有前缀、部分前缀或后缀关系的 cuboid 中与 v 对应的元素的逻辑位置 $c_1,c_2,\ldots,c_s$;
(7) FOR $r=1$ TO s DO
(8) IF Buff out[r]中存在逻辑位置为 c_r 的数据项(c_r,x) THEN 修改(c_r,x)为$(c_r,x+v)$; ELSE (c_r,v)写入 Buff_out[r];

(9) 释放 root_Buff 所占内存;
(10) 将 T 中非其他 cuboid 子树根的那些 cuboid 结果写入磁盘,同时为其生成 Header 文件,释放其所占用的内存;
(11) $\alpha=\alpha-\{T\}$;
(12) WHILE α 不空 DO
(13) 从 α 中选择一个 cuboid 子树 T (T 的根 root 对应的 cuboid 已在内存);
(14) root_Buff=root 对应的 cuboid 结果地址;
(15) 为 T 中除 root 外的每个 cuboid 分配缓冲区 Buff_out[r];
(16) FOR root_Buff 中的每一个数据项 v DO
(17) 使用公式(1)~(3)计算与 root 具有前缀、部分前缀或后缀关系的 cuboid 中与 v 对应的元素的逻辑位置 $c_1,c_2,\ldots,c_s$;
(18) FOR $r=1$ TO s DO
(19) IF Buff_out[r]中存在逻辑位置为 c_r 的数据项(c_r,x) THEN 修改(c_r,x)为($c_r,x+v$);
ELSE (c_r,v)写入 Buff_out[r];
(20) 释放 root_Buff 所占内存;
(21) 将 T 中非其他 cuboid 子树根的那些 cuboid 结果写入磁盘,同时为其生成 Header 文件,释放其所占用的内存;
(22) $\alpha=\alpha-\{T\}$;
(23) 将内存中 cuboid 结果写入到磁盘,同时为其生成 Header 文件.

容易看到,当内存满足 M-Cube 要求时,算法只需扫描原始压缩多维数据集合一次,即可在内存中计算出所有的 cuboid,具有最小的 I/O 代价.限于篇幅,本文省略了 M-Cube 算法的复杂性分析,详见文献[6].

2.2.3 Hybrid-Cube 算法

前面给出的两种 Cube 算法分别有着各自的优点和问题.M-Cub 算法的性能很高,但要求较大的主存缓冲区,G-Cube 算法对内存没有严格的需求,但是效率低于 M-Cube 算法.在 G-Cube 算法的计算过程中,每次计算的 cuboid 子树中所含有的 cuboid 个数会随着时间的推迟越来越少,而且每个 cuboid 结果也逐渐变得越来越小.因此,必存在一个时刻 t,在 t 时刻以后,主存可以容纳多个子树的结果.此时,如果仍以 cuboid 子树为单位进行计算,会产生一些不必要的 I/O 开销.本节给出一种具有动态调节能力的 Cube 计算方法 Hybrid-Cube 算法,使得在 Cube 计算过程中,算法能够根据当前内存容量和每个子树中所有 cuboid 的大小之和动态地选择使用 G-Cube 和 M-Cube 算法以高效率地完成 Cube 的计算.Hybrid-Cube 算法需要估计 cuboid 结果的大小.目前有很多方法来估计 cuboid 结果的大小[5],在此我们不对其做详细的讨论.Hybrid-Cube 算法定义如下:

输入:压缩的多维数据集合、Header 文件、Cube 计算计划 α.
输出:Cube 结果(一组压缩的多维数组).
(1) WHILE α 不空 DO
(2) 从 α 中选择一个子树 T(T 的根 cuboid 已经被计算出来);
(3) $F=\{t\mid t\in\alpha, t$ 的根是 T 中的结点$\}$;
(4) Parallel-SET=$\{T\}$;
(5) WHILE F 非空 DO
(6) 从 F 中选择一个子树 t;
(7) IF 主存缓冲区可以存储 Parallel-SET 中所有子树及 t 中的全部 cuboid 结果;
(8) THEN Parallel-SET=$\{t\}\cup$Parallel-SET;$F=F\cup\{$以 t 中 cuboid 为根的其他 cuboid 子树$\}$;
(9) $F=F-\{t\}$;
(10) IF |Parallel-SET|=1 /* 内存不能满足同时计算多个子树的要求 */
(10) THEN 采用 G-Cube 方法计算子树 T;
(11) ELSE 采用 M-Cube 方法计算 Parallel-SET 中的子树;
(12) $\alpha=\alpha-$Parallel-SET.

限于篇幅,本文省略了 Hybrid-Cube 算法的复杂性分析,详见文献[6].

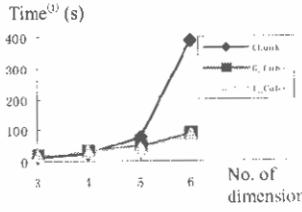
3 实验结果

我们在 P350 微型计算机上实现了 G-Cube、M-Cube 和 Hybrid-Cube 算法. P350 微型计算机的主存器容量为 64M 字节,磁盘容量为 6G 字节,磁盘的顺序读写速度为 2.5M/s. 为了与惟一一个现存的压缩多维数据仓库上的 Cube 算法[2](以下简记 Chunk 算法)比较,我们在相同计算机环境下实现了 Chunk 算法. 影响 Cube 算法的主要因素有 4 个:(1) 数据集合维属性个数;(2) 压缩比,即$\frac{\text{压缩后数据集合的大小}}{\text{压缩前数据集合的大小}}$;(3) 数据集合中的数据量;(4) 主存缓冲区大小. 我们进行了 5 组实验考察上述 4 个因素对 G-Cube,M-Cube,Hybrid-Cube,Chunk 算法的影响,并对这 4 个算法的性能进行了比较. 限于篇幅,这里仅给出实验结果.

第 1 组实验(4 个 64 万个数据项的实验数据集合)考察维属性个数对 G-Cube,Hybrid-Cube,Chunk 算法性能的影响,实验结果如图 3 所示.

第 2 组实验(3 个 5 维 102.4 万个数据项的实验数据集合,变化一维的基数)考察多维数据集合维属性基数的改变对 G-Cube,Hybrid-Cube,Chunk 算法性能的影响,结果如图 4 所示.

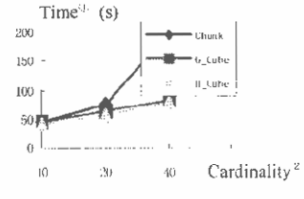
第 3 组实验(3 个 6 维实验数据集合,数据量变化)考察多维数据集合的数据量对 G-Cube,Hybrid-Cube和 Chunk 算法性能的影响,结果如图 5 所示.



①时间,②维个数.

Fig. 3

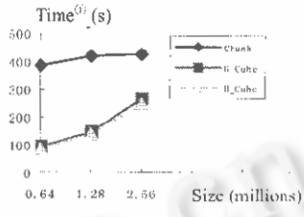
图 3



①时间,②基数.

Fig. 4

图 4



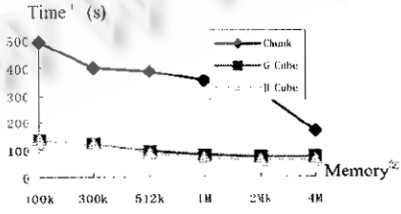
①时间,②大小(百万).

Fig. 5

图 5

第 4 组实验(1 个 6 维 64 万个数据项实验数据集合,主存大小变化)考察多维数据集合的主存大小对 G-Cube,Hybrid-Cube,Chunk 算法性能的影响,结果如图 6 所示.

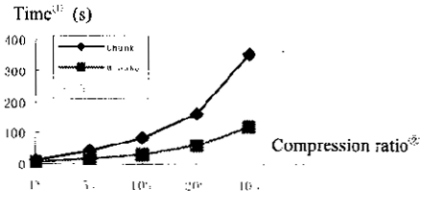
第 5 组实验(1 个 3 维实验数据集合,压缩比变化)比较 M-Cube 和 Chunk 算法的性能,结果如图 7 所示.



①时间,②内存大小.

Fig. 6

图 6



①时间,②压缩比.

Fig. 7

图 7

4 结 论

本文提出 3 种压缩数据仓库上的 Cube 算法.这些算法直接在压缩数据仓库上执行 Cube 操作,无须数据解压缩,适用于所有映射完全的压缩方法.实验和理论分析表明,本文的 Cube 算法的效率高于其他 Cube 算法.

References:

[1] Eggers, S., Shoshani, A. Efficient access of compressed data. In: Vijayaraman, T., Buchmann, M., Mohan, A. P., *et al.*, eds. Proceedings of the 6th International Conference on Very Large Data Bases. Montreal, Canada: IEEE Computer Society Press, 1980. 205~211.

[2] Zhao, Y., Deshpande, P. M., Naughton, J. F. An array-based algorithm for simultaneous multidimensional aggregations. In: Peckham, J., ed. Proceedings of the ACM SIGMOD International Conference on Management of Data. Tucson, Arizona, USA: ACM Press, 1997. 159~170.

[3] Gray, J., Bosworth, A., Layman, A., *et al.* Data cube: a relational operator generalizing group-by, cross-tab and sub-totals. In: Stanley, Y. W. Su, ed. Proceedings of the 12th International Conference on Data Engineering. New Orleans, Louisiana: IEEE Computer Society Press, 1996. 152~159.

[4] Agarwal, S., Agrawal, R., Deshpande, P. M., *et al.* On the computation of multidimensional aggregates. In: Vijayaraman, T., Buchmann, M., Mohan, A. P., *et al.*, eds. Proceedings of the 22th International Conference on Very Large Data Bases. Mumbai, India: Morgan Kaufmann Publishers, Inc., 1996. 506~521.

[5] Shukla, A., Deshpande, P. M., Naughton, J. F., *et al.* Storage estimation for multidimensional aggregates in the presence of hierarchies. In: Vijayaraman, T., Buchmann, M., Mohan, A. P., *et al.*, eds. Proceedings of the 22th International Conference on Very Large Data Bases. Mumbai, India: Morgan Kaufmann Publishers, Inc., 1996. 522~531.

[6] Gao, Hong, Li, Jian-zhong. Cube algorithms for very large compressed data warehouses. Technology Report, Department of Computer Science and Engineering, Harbin Institute of Technology, 2000. http://202.118.239.124/doc/Cube_Algorithms.

Cube Algorithms for Very Large Compressed Data Warehouses*

GAO Hong, LI Jian-zhong

(*Department of Computer Science and Engineering, Harbin Institute of Technology, Harbin* 150001, *China*)

E-mail: lijz@banner.hl.cninfo.net

Abstract: Data compression is an effective approach to improve the data wharehouses. On line analysis processing (OLAP) is the most important application on the data warehouses, and Cube is one of the most operators in OLAP. Thus, it is a big challenge to develop efficient algorithms for compressed data warehouses. Although many algorithms to compute Cube have been developed recently, there is little to date in the literatures about Cube algorithms for compressed data warehouse. To the authors' knowledge, there is only one paper that presented a Cube algorithm for compressed data warehouses with a special compression method called chunk offset. A set of Cube algorithms for very large and compressed data warehouses are proposed in this paper. These algorithms operate directly on compressed datasets without the need of decompressing them first. They are applicable to a variety of data compression methods. The detail analysis of I/O and CPU cost are also given, and compared with the existed algorithms by experiment. The analytical and experimental results show that the algorithms proposed in this paper are more efficient than other existed ones.

Key words: data warehouse; compressed data warehouse; OLAP (on line analysis processing); Cube

* Received September 4, 2000; accepted March 15, 2001

Supported by the National Natural Science Foundation of China under Grant No. 69873014; the National Grand Fundamental Research 973 Program of China under Grant No. G1999032704