CHUNK 中的多维数据压缩*)

Compress Multi-Dimensional Data in Chunk

田新锋　李战怀　朱　岩
(西北工业大学计算机系　西安710072)

Abstract　The data explosion is a problem which must be solved in OLAP on basis of object-relational technology. The Aggregation in multiple relational cubes is an important operation. A compressing method in which the bit-map is used to compress multi-dimensional data in chunk is advanced in this paper to solve the data explosion.

Keywords　Data cube, MOLAP, ROLAP

在 OLAP 应用中,计算多个相关 cubid 聚集是一个极其重要的操作,包括如何有效计算 CUBE。在文[1]中,为关系操作引入了 CUBE 操作符,丰富了关系语言对多维数据计算的处理能力。已有很多文章对此作了深入讨论,主要分为两类:一是对以关系构造的多维数据进行优化;一是对多维数组构造的多维数据进行优化。由于关系数据库的成熟,及应用范围的广泛,对以关系构造的多维数据(即 ROLAP)进行 CUBE 计算的研究明显多于对多维数组构造的多维数据(即 MOLAP)的 CUBE 计算研究。

数据仓库主要用于决策支持。无论关系数据库还是多维数据库,提高其性能的一个重点就是如何更有效地执行 CUBE 聚集。我们在 ORDBMS 中实现多维数据时,涉及到将度量数据转化为数组,在多维计算及数据存贮转化时都涉及到数组。数组在 CUBE 计算时可大大加快数据计算速度,而稀疏数组又是多维数据的另一个问题。本文首先提出一个在 chunk 中采用位图进行多维数据压缩的方法,然后与 chunk 中的 offset 压缩进行比较。

1. 多维数组压缩

由于 MOLAP 牵涉到稀疏数组的问题,所以为了减小稀疏数组的大小,并方便多维数据的计算,我们首先讨论多维数组的压缩问题。

1.1　Chunk 压缩

1.1.1　Chunk 的产生

在 ROLAP 中,多维数据以星型模式表示,多维数据中的每个单元用一个元组表示,一个元组既包含度量数据,又包含各维的标识,其存储以元组或属性为单位进行,而不能直接针对度量对应的数据单元,而且不能一次定位到数据单元。

在 MOLAP 中,数据以数组的形式存储。我们知道,存储器是线性存储,而不是多维方式存储,多维数组的存储最终也必须转化为线性存储,即按不同的维序顺序存储,在读取时,只能按照维序进行,所以计算时也只能按照维序进行,对于数据读取及计算都不是很有效,文[4]给出了有关 chunk 的构造情况。Chunk 以 I/O 块为单位,每个 chunk 都是由数据解析后再组合成的 n 维立方体,是原立方体中具有相同维数的一个子立方体。原立方体以 chunk(子立方体)为单位按维序进行线性存储,每个 chunk 再以子立方体的维序存储。每次 I/O 都是原立方体的一块,所以可以不按照次序读取。如一个三维数组 $D_1D_2D_3$, D_1, D_2, D_3表示数据的三个维,用 $|D_i|(1\leqslant i\leqslant 3)$ 表示 D_i 维的势。

*)本文研究得到国家自然科学基金、国家863计划项目资助.田新锋　博士生,主要研究领域为数据库,数据仓库技术.李战怀　教授,主要研究领域为数据库理论与技术,数据仓库技术.朱　岩　讲师,主要研究领域为数据库理论与技术.

$D_1D_2D_3$被分解为一系列的子立方体 $d_1d_2d_3$。每个子立方体都是一个三维数组，只是每维的势比原立方体的势小，即有：$\Sigma|d_i|=|D_i|(1\leqslant i\leqslant 3)$。子立方体 $d_1d_2d_3$按(D_1,D_2,D_3)的顺序存储，子立方体内部按(d_1,d_2,d_3)的顺序存储。

每次 I/O 可以读入或写回一个子立方体，每读入一次后可对该子立方体进行计算，因此，这样的 chunk 立方体对于输入、输出及计算都提高了效率。

下面我们给出存取数据单元的方法。

设 offset 是在 chunk 子立方体中该数据项线性化后的位置。如某个 chunck 中具有子立方体 $d_1d_2d_3$，当以 $O=(d_1,d_2,d_3)$的维序存储时，则子立方体 $d_1d_2d_3$的坐标(i,j,k)唯一对应子立方体中的一个数据单元，可以将坐标(i,j,k)转化为子立方体内部线性存储的位置，即有：

$$offset=i+j\times|d_1|+k\times|d_1|\times|d_2|$$

更一般地，设 q_i 是维序中第 i 维的坐标，所以对于子立方体中的坐标$(q_1,q_2,\cdots,q_n)$，可得 chunk 内偏移量 offset 为：

$$offset=q_1+\sum_{i=1}^{n}(qi\cdot\prod_{j=1}^{i-1}|Dj|) \tag{1}$$

给定原立方体的坐标$(Q_1,Q_2,\cdots,Q_n)$，需要经过一些转换才能取得对应的数据单元。可把 chunk 看作原立方体的坐标单位，则可得虚拟坐标$(V_1,V_2,\cdots,V_n)$，其中：

$$V_i=\begin{cases}\left[\dfrac{Q_i}{C_i}\right]+1 & \text{当}\left[\dfrac{Q_i}{C_i}\right]\neq\dfrac{Q_i}{C_i}\text{时}\\ \left[\dfrac{Q_i}{C_i}\right] & \text{当}\left[\dfrac{Q_i}{C_i}\right]=\dfrac{Q_i}{C_i}\text{时}\end{cases} \tag{2}$$

并且，

$$q_i=Q_i \bmod C_i \tag{3}$$

根据虚拟坐标用公式(1)可得出对应数据单元虚拟位置 OFFSET，即数据单元所在 chunk。

$$OFFSET=V_1+\sum_{i=2}^{n}\left(V_i\cdot\prod_{j=1}^{i-1}\frac{|D_j|}{V_j}\right) \tag{4}$$

将坐标$(q_1,q_2,\cdots,q_n)$代入公式(1)，可取出坐标$(Q_1,Q_2,\cdots,Q_n)$对应的数据单元 c。

1.1.2 Chunk 的压缩

由于大型数据仓库中用多维方式组织数据，每个立方体可能具有很多个维，而每一维的域会很大，因此这些维组合在一起以后将迅速膨胀，产生具大的数据空间。一些维的域中有很多值的组合对于现实世界可能是无意义的，即在与之对应的数据单元中根本就不存在数据。数据膨胀增大了磁盘成本，若再加上 summary 的预计算，数据膨胀会进一步加剧。所以压缩多维数据以减少数据所占用的空间是很必要的。

文[3]中提出了 chunk-offset 的压缩方法，将数组分解后再组合放入一个 chunk 中，chunk 中的数据是小的 n 维立方体。每个有值的数据项存入一对(offset, data)，offset 即是数据单元在未压缩 chunk 中的位置，无值或值为0的数据不存储。这样当计算时，如果 offset 相同则可进行聚集计算，不计算为空的数据单元，从而节约了计算时间。

chunk 算法存在着几个不足：首先是 chunk-offset 的压缩方式引入数组入口点偏移，这种方法虽然压缩掉了为空或0的数组单元，但却倍增了不为空或0的数组单元大小，因此对一个不太稀疏或很大的数组来说并不理想。

1.2 位图压缩

chunk-offset 压缩方式适用于多维数组极端稀疏的情况时比较有效，即稀疏度小于6%时。对于稀疏度大于6%的多维数组，则效果不佳。我们借用索引中的位图索引采用的方法，将它应用到 chunk 压缩中，以获得好的压缩效果，作为 chunk-offset 压缩方法的弥补。

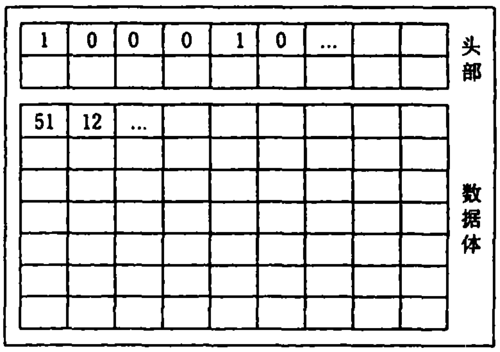

图1 位图压缩

将 chunk 分为两部分：头部和数据体。头

部放置位图数据，用二进制位标识未压缩数组中每个数据单元的状态。二进制位为1表明与该位对应的数据单元存在合法值，为0表明不存在合法值。数据体按先后顺序放置合法数据。

在获取 chunk 中某个位置 offset 的数据单元时，如果头部对应位置的位值为0，则无合法数据。如果头部对应位置的位值为1，则先需要将头部中的从0到 offset 的位图数据累加。由此累加值可以从数据体中直接取得相应单元数据。如图1所示。在头部 offset＝0的们值为1，则其数据单元值在数据体中的位置为0×*SizeOfCell*，即其值为51。在头部偏移为4的位值为1，则数据单元值在数据体中的位置为4×*SizeOfCell*，即其值为12。

这种压缩方式适用于多维数组并不是很稀疏的情况时效果明显。因为给定 chunk 中放置的数据数目，头部的大小就已确定下来，即其大小与数组的稀疏度无关，只与 chunk 中数据单元数有关。

1.3 多维数据对象的改进位图压缩

位图压缩中的头部占用固定大小的长度，在数据稀疏度很小时效果很好。但在数据稀疏度很大时，效果不理想。例如，一个 chunk 对象中只有一个有合法值，而仍需为头部耗费大量的空间。而恰恰是由于头部所占用的空间造成了浪费。为了弥补位图压缩方式的这种不足，我们需要做一些改进。

当数据稀疏度很大时，所有数据单元中很少有合法值。比如64个数据单元中只有一个数据单元具有合法值。这样如果一个 chunk 中有64个数据单元，每个数据单元占用4个字节。则用 chunk-offset 方法压缩后为5个字节(偏移量占用1个字节)；用位图压缩后为12个字节。可以看出数据稀疏度很大时位图压缩方式较 chunk-offset 差。

我们可以对位图压缩作一改进。数据稀疏度大时，效果较差的原因出在头部。将头部分为两部分：标志区和头数据区。可以为每8个数据单元设一个标志，将连续的8位用一位来表示，放在标志区。为1则表示相应的8个数据单元中至少有一个合法值，为0表示相应的8个数据单中没有合法值。如果值为1，则在头部数据区存放一个字节(8位)，该字节标记 chunk 中的8个相应数据单元是否有合法值。如图2所示。

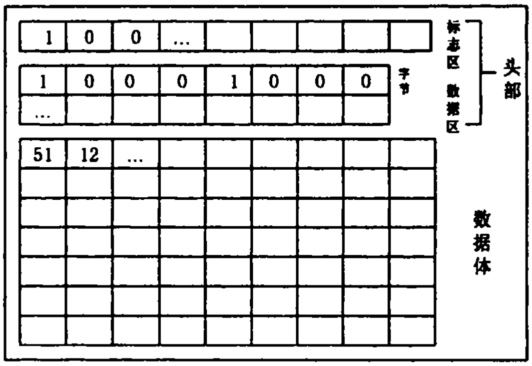

图2 改进位图压缩

在前面所说的那个 chunk 中，用改进位图压缩后为6个字节。只比 chunk-offset 多一个字节。当 chunk 中有两个合法值时，chunk-offset 占用10个字节。而改进位图压缩则需要分为两种情况，当这两个合法值不在同一个连续的8位中时占用11个字节，在同一个连续8位中时占用10个字节。因此效果也随之较位图压缩增强，且不比 chunk-offset 差多少。每个 chunk 中改进位图压缩最多较 chunk-offset 压缩多一个字节(当然，只是在该例中)。但是位图压缩在 CUBE 计算中远优于 chunk-offset，这是因为在 chunk-offset 压缩中，我们根本不知道哪个单元有合法值，而哪个单元没有，必须对整个 chunk 进行扫描，而在位图压缩中头部则可看作是位图索引，在改进位图压缩中的头部可看作二级索引，并且这两种位图压缩中的索引都是位操作，因此有很快的速度。

特别地，当 chunk 中没有一个合法数据单元时，改进位图压缩占用1个字节，而 chunk-offset 不占用任何字节。当一个 chunk 作为一个对象存在时，长度为0的对象就会出现一定的问题。需要一定的手段加以控制。因此最终也不可以使长度为0。

从 chunk-offset 压缩与位图压缩可以看出，压缩后的 chunk 大小是变化的。其大小取

决于 chunk 中的合法值数。位图压缩中头部长度是固定的，数据区长度是变化的。在改进位图压缩中不但数据区长度是变化的，头部长度也因头部数据区的不确定而变化。

2. 压缩方法比较

下面我们对两种压缩方法及无压缩进行比较，以找出对压缩方法的选择尺度，便于在实现时，选择最有效的压缩方法。

设多维数组 A 的体积为 V，压缩后 A 的体积为 T，chunk 大小为 C(即磁盘 I/O 块的大小为 C)，数据 A 的稀疏度 $S=T/V$。不失一般性，我们假设数据均匀分布。

2.1 chunk-offset 与位图压缩的比较

对于多维数组的任一个 chunk，我们分别讨论 chunk-offset 与位图压缩的计算。

在 chunk-offset 压缩中，设每个 offset 占用 x_0 个字节，每个数据单元占 u 个字节，数据单元数为 y 个，有值数据为 z 个。则有下面方程：

$$\begin{cases}(x_0+u)z=C\\ 2^{8x_0}=y\end{cases}$$ 解得：

$$x_0=\frac{\lg y}{8\lg 2},z=\frac{C\lg 2}{\lg y+8u\lg 2}$$

在位图压缩中，设位图占用 x_b 个字节，每个数据单元占 u 个字节，数据单元数为 y 个，有值数据为 z 个。则有下面方程：

$$\begin{cases}x_b+uz=C\\ 8x_b=y\end{cases}$$ 解得：

$$x_b=\frac{y}{8},z=\frac{8C-y}{8u}$$

现在我们可以对这两种压缩方法进行比较。当 *chunk* 中的数据单元数相等时，*chunk-offset* 压缩浪费的空间为 x_0z 个字节，而位图压缩浪费的空间为 x_b 个字节。假设 *chunk-offset* 优于位图压缩，则有：$x_0z<x_b$，即有 $\frac{z}{y}=S<\frac{lg2}{lgy}$，否则当 $S>\frac{lg2}{lgy}$ 时，位图压缩优于 *chunk-offset* 压缩。

从上面的讨论可以看出，是选择 *chunk-offset* 还是位图压缩是由稀疏度 S 与 *chunk* 中数据单元数共同决定的。

2.2 改进位图压缩与位图压缩的比较

在改进位图压缩中，设标志区占用 x_f 个字节，头部数据区占用 x_d 个字节，每个数据单元占 u 个字节，数据单元数为 y 个，有值数据为 z 个。

当很极端地任两个有值数据都不处在同一个连续8个标志位中，则有下面方程：

$$\begin{cases}x_f+z+uz=c\\ 8\times 8x_f=y\end{cases}$$

解得：$x_f=\frac{y}{64},z=\frac{c-\frac{y}{64}}{u+1}$

此处改进位图压缩浪费的空间为：x_f+z。

当改进位图压缩优于位图压缩时，则有：

$x_f+z<x_b$，

即有 $S=\frac{z}{y}<\frac{7}{64}=0.109$

因此当稀疏度达到0.109时，改进位图优于位图压缩。当然这是在一个很极端情况下的一个临界值。当达到另一个极端：没有非法值时，改进位图压缩较位图压缩多出 x_f 个字节。

2.3 改进位图压缩与 chunk-offset 的比较

我们也在极端稀疏的情况下讨论。

当 chunk-offset 优于改进位图压缩时，则有：

$x_0z<x_f+z$，

即有 $S=\frac{z}{y}<\frac{\lg 2}{8\lg y-64\lg 2}$

因此，改进位图压缩与 chunk-offset 压缩方式的选择依赖于稀疏度 S 和数据单元数 y。

结束语 本文提出了在内存中直接对压缩数据进行有效计算的算法。并提出了更高效的 CUBE 算法，该算法用维序及 MMST 保证了最小父亲，用 MMST 分块及相应 CUBE 算法保证了缓冲结果的使用及最少磁盘扫描，而分块与 ROLAP 中的分片技术相似。数组是有序的，从而不需要共享排序。直接对压缩数据的操作也很大地节约了内存的使用。因此，该算法将 ROLAP 中各种优化方法与数组的优点结合在一起，达到较高的性能要求。

这些压缩算法的目的只是为了节约外存空间，避免数据膨胀。如果在计算的同时使用压缩，使整个计算过程中的计算对象都是针对

压缩的 chnuk，就可以用很小的内存空间计算很大的 CUBE，但我们需要根据这种需求选择合适的压缩方法。

位图压缩的压缩效果在数据极端稀疏时效果太差，在数据仓库中的数据又恰恰是极其稀疏，所以虽然位图压缩的数据定位及存取速度最快，但是也不是理想的方法。

在 chunk-offset 压缩中，我们根本不知道哪个单元有合法值，而哪个单元没有，必须对整个 chunk 进行扫描。当向上层聚集时，需要对上层的新 chunk 进行多次扫描，因此扫描量极大。虽然 chunk-offset 压缩在数据极其稀疏时稍好于改进位图压缩，但改进位图压缩在存取效率上远远高于 chunk-offset 压缩方式。

参考文献

1 Jim Gray, Adam Bosworth, Andrew Layman, Hamid Pirahesh. Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab and Sub-Totals. In: Proc. of the 12th Int. Conf. on Data Engineering, 1996. 152～159

2 Agrawal S, et al. On the Computation of Multidimensional Aggregates. In: VLDB'96. 506～5211

3 YiHong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In: SIGMOD'97

4 Sunita Sarawagi, Michael Stonebraker, Efficient Organization of Large Multidimensional Arrays. In: ICDE'94

5 Jianzhong Li, Lawrence Berkeley National Laboratory, Jaideep Srivastava. Aggregation Algorithms for Very Large Compressed Datra Warehouses. In: VLDB'99

6 Eggers, S, Shoshani A. Efficient Access of Compressed Data. In: VLDB'80

7 Sarawagi S, et al. On computing the data cube: [Technical Report]. RJ10026, IBM Almaden Research Center, San Jose, Ca, 1996

8 Deshpande P M, et al. Computation of multidimensional aggregates: [Technical Report]. 1314, University of Wisconsin, Madison, 1996

9 Ross K A, Srivastava D. Fast Computation of Sparse Datacubes. In: VLDB'97

（上接第222页）

行，从而显著提高了相似检索的效率。

参考文献

1 Bentley J L. Multidimensional binary search trees used for associative searching. Communication of ACM, 1975, 18(9): 509～517

2 Guttman A. R-tree: A Dynamic index structure for spatial searching. In: Proc. of the ACM SIGMOD International conference on management of data. 1984. 47～54

3 White D A, Jain R. Similarity indexing with SS-tree. In: Proc. of the 12th Int. Conf. on Data Engineering. 1996

4 Katayama N, Satoh S. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In: Proc. of ACM SIGMOD. 1997

5 Chakrabarti K, Mehrotra S. The hybrid tree: An index structure for high-dimensional feature spaces. In: Proc. of the 15th Int. Conf. on Data Engineering. 1999

6 Uhlmann J. Satisfying general proximity/similarity queries with metric trees. Information Processing Letters, 1991, 40: 175～179

7 Baeza-Yates R, Cunto W, Manber U, Wu S. Proximity matching using fixed-queries trees. In: Proc. of the fifth symposium on Combinatorial Pattern Matching (CNCS807, June). Spring-Verlay, New York, 1994. 198～212

8 Brin S. New neighbor search in large metric space. In: Proc. of VLDB'95. Zurich, Sept. 1995. 574～584

9 Ciaccia P, Patella M, Zezula P. M-tree: An efficient access method for similarity search in metric space. In: Proc. of VLDB'97. Athens, Greece, Aug. 1997. 426～435

10 Andrew P B, Linda G S. A flexible image database system for content-based retrieval. Computer Vision and Image Understanding, 1999, 75(1/2): 175～195

11 Szu H H, Hartley R L. Fast Simulated Annealing. Physics Letters A, 1987, 122: 157～162

12 Goldberg D E. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading MA. 1989