高效实现数据立方

Venky Harinarayan　Anand Rajaraman　Jeffrey D. Ullman

摘要

决策支持涉及非常大的数据集重要查询的应用。因为反应时间应该要短，查询的优化就很重要。使用者通常把数据看作数据立方。像总的销售额，每一个数据集的单元是一个由利益集合组成的。很多单元的值依赖于数据集中其他单元的值。普遍且有效的查询优化技术是实现某些或者全部的单元而不是每次都从原始数据中计算。商业系统和其它不同的是它实现数据集的方法。在这篇论文中，我们调查当实现所有视图的代价太昂贵的时候，要实现哪一些单元（视图）的问题。格框架被用于表达视图间的依赖关系。我们用贪心算法来清除这些网格和决定要实现的好的视图集合。贪心算法在一个小的常数优化因子内和一系列的模型下操作。于是我们考虑最普遍的超立方体格和仔细检查为超立方体实现的视图的选择，给出一些回答一个查询所用的空间和平均时间的好的折衷方法。

1 介绍

决策支持系统（DDS）正在迅速变成为商业获取竞争力优势的关键。DDS 允许商业去获取在动态数据库中封闭的数据，并且把那个数据转化成有用的信息。很多公司已经建立或者正在建立新的被叫做数据仓库的支持联合决策的数据库，使用者可以用它来执行分析。

动态数据库维护状态信息的同时，通常还维护历史信息。结果，数据仓库会变得很大而且一直在增长。DDS 的使用者通常关心趋势而不是独立的纪录。因

此，决策支持查询充分地利用集合，而且比 OLTP 查询复杂得多。

数据仓库的大小和查询的复杂程度会导致查询要花非常长的时间去完成。而查询通常所需要的执行时间最多是几秒或者几分钟。

有很多的方法去获得上述的性能目标。查询优化器和查询评估技术能够通过处理更好的［CS94］,[GHQ95],[YL95]集合，使用像位映射索引、合并索引和其它的索引这些不同的索引策略来提高。

一个普遍使用的技术是用来实现（提前运算）常常被查到的查询。在 Mervyn 的部分存储链的数据仓库，例如，有一个总共有 2400 个提前运算的表格[Rad95]来提高查询性能。选取正确的查询集合来实现是一个重要的任务，通过实现一个查询能够快速地回答其它查询。比如，虽然一个查询不是经常被查到，但它能够帮助我们快速地回答其它问题，我们也许会实现这个查询，。在这篇论文中，我们提出能够使我们实现一个好的查询的集合的框架和算法。我们的框架还能够使我们推断出实现这些查询的顺序。

1.1 数据立方

工作在数据仓库的图形化环境的使用者通常用 2 维、3 维或者更高维子数据立方的数据立方中的数据来探索发现有趣的信息。这个数据集中的每一个单元的数值是一些利益的“措施”。考虑 TCP-D 决策支持的基准来作为一个例子。

例子 1.1 TPC-D 标准模拟一个商业仓库。零件从提供者处购买，然后以价格 SP 卖给顾客。这个数据库有过去 6 年类似这样的的交易信息。

有 3 个我们关心的方面：part, supplier 和 customer。关心的“测度”是总体的价格。所以对于每一个在 3 维数据集的单元(p,s,c)，我们存储从提供者 s

中购买的、卖给顾客 c 的零件 p 的总体价格。在这个 Section 的术语维度和属性是可互换的。通常来说，一个给定的维度也许有很多在 Section 2 中的属性。

使用者通常关心固定的价格：例如，对于一个买给给定顾客的给定零件，总的价格是多少？[GBL95]表明在每一个维度中添加一个额外的值“ALL”到值域中来获得。对于一个给定的零件 p，一个给定的顾客 c，和“ALL”（所有）提供者我们想要总体的价格这个问题。这个查询通过查找单元(p, ALL, c)来回答。

在这篇论文中，我们用大小为 1GB 的 TPC-D 数据库作为运行的例子。这个标准上的更多的细节参照[TPCD]。

对于使用者，我们只讨论表现为多维数据集的数据。下述的实现选择是可能的：

1、物理地实现整个多维数据集。这个方法给出最好的查询反应时间。可是，对于大型数据库，提前计算和储存每个单元不是一个可行的选择，因为消耗的空间过多。需要记住的是，数据库消耗的空间也是一个将要创建数据库所用时间的好指标，这对于很多应用来说是很重要的。消耗的空间还会影响到索引，所以要被添加到总体的开销。

2、 什么都不实现。在这个情况下，我们需要获得原始数据，并根据要求计算需要的每个单元。这个方法丢弃了存储原始数据的数据库系统的快速查询反应。除了原始数据不需要额外的空间。

3、只实现数据库系统的一部分。在这篇论文中，我们考虑这个方法。在这个多维数据集中，很多单元的值都能通过其它单元计算得出。这种依赖和单元的值可以表达为其他单元的值的函数的电子表格相似。我们称这些单元“依赖”其它单元。举例来说，在 Example1.1 中，我们能够计算单元(p,s_1，c)，…，

(p,s_(N_supplier)，c)值的和（p, ALL, c）单元的值，其中 N_supplier 是提供者的数目。我们实现越多的单元，查询的性能越高。可是对于大型的数据库，由于空间和其它的约束，我们也许只能够实现数据库的一小部分。因此对我们来说，选择正确的单元来实现是很重要的。这种方法具有规模化而且能够很好地处理大型数据库。

任何有“ALL”值的单元作为它的地址中的一个组成部分是一个依赖单元。这个单元的值可以通过数据库的其它单元计算得出。如果一个单元的组成部分没有“ALL”，它的值不能通过其它单元计算得出，我们必须查询原始数据来计算它的值。有“ALL”作为它组成部分的单元通常需要用到数据库所有单元的很大一部分来计算独处。在 Example 1.1 中的多维度的 TCP-D 数据库中，数据库中70%的单元是具有依赖性的。

要计算数据库中的哪些单元的问题是非常现实的。有很多不同的商业系统会选择上述不同的策略的其中一个。很明显，每个策略都有它的优点。举例来说，对于那些规模化和重要的性能并不重要的应用，我们采用实现一切的策略。举例来说，Essbase 系统[ESS]实现整个数据库，而 BusinessObjects[X94]什么都不实现，MetaCube 系统［STG］只实现数据库的一部分。

还有在一个关系型系统或者一个专有的 MDDB(多维数据库)系统中，数据库的实现储存在哪里的问题。在这篇论文中，我们假设在关系型系统中数据集储存在“摘要”表格。数据库的单元的集合被分配到不同的表格。

数据库的单元被组织成基于地址中的“ALL”的位置的不同集合。因此，举例来说，所有符合(_, ALL, _)地址的所有单元被放在同一个集合中。在这里，“_”是一个表示匹配任何值的占位符。这些集合中的任一个都匹配一个不同的 SQL

查询。(_, ALL ,_)单元的集合的值是 SQL 查询的输出：

```
1. SELECT Part, Customer, SUM(SP) AS TotalSales
2. FROM R
3. GROUP BY Part, Customer;
```

在这里，R 意味着原始数据关系。对应于不同单元集合的查询只有在 GROUP-BY 子句中不同。通常来说，在单元的集合中的有“ALL”的属性，没有出现在上述的 SQL 查询的 GROUP-BY 子句中。举例来说，提供者有“ALL”值在集合描述（_, ALL ,_）。因此它没有出现在 SQL 查询的 GROUP-BY 子句中。因为不同单元的集合的 SQL 查询只有 grouping 属性不同，我们用 grouping 属性来唯一辨别查询。

决定要实现的那一个单元的集合等同于决定要实现哪一个匹配的 SQL 查询（视图）。这篇论文的其它部分为视图而工作而不是单元的集合。

1.2 促进例子

我们在 Example 1.1 考虑的 TCP – D 数据库有 3 个属性：part, supplier, customer。我们有 8 个可能的分组属性。我们列举了下面用他们结果的行的所有可能的查询（视图）。记住它足够去提及视图中的 GROUP-BY 子句的属性。

```
1. part , supplier, customer(6M, i. e.,  6 million rows )
2. part, customer (6M)
3. part  supplier(0.8M)
4. supplier  customer(6M)
5. part(0.2M)
```

6. supplier(0.01M)
7. customer(0.1M)
8. none(1)

none 表明在 GROUP – BY 子句中没有属性。Figure 1 表明这 8 个视图组织成一个我们能够讨论的 Section 2 的格的类型。在这个图中的视图，我们用 p 表示 part, s 表示 supplier, c 表示 customer。

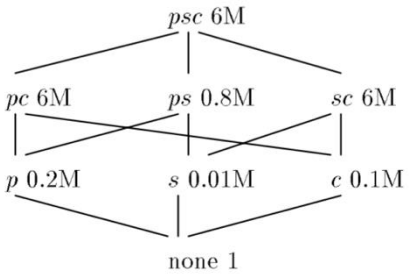

Figure 1: The eight views constructible by grouping on some of `part`, `supplier`, and `customer`

一个可能的使用者查询是一个对整个视图的需求。例如，使用者也许会问根据 part 分组的价格。如果我们实现只根据 part（view 5）分组的视图，我们只能够扫描视图和输出答案。我们还能够根据 part 和 customer 分组的视图（view 2）来回答这个查询。在这种情况下，因为我们有每个顾客、每个分组的总体价格，要们需要把所有顾客的购买价格加起来来获得这个结果。

在这篇论文中，我们假设回答一个查询的开销和检查的行数成正比。因此，如果 view5 实现了，找到根据 part 分组的总的价格的开销是处理 20 万行（视图的大小）的开销。为了用 part, customer 视图回答同样的查询我们需要处理 600 万行。

另一种查询会问一个独立零件的价格，叫“部件”。如果一个视图没有索引，那么我们仍需要扫描整个视图（或者平均为视图的一半）来回答这个问题。因此，

一样的查询，视图 5 需要 20 万行，视图 2 需要 6000 万行。可是，如果两个视图都有合适的索引可用，视图 5 需要访问 1 行来找到部件的价格，然而视图 2 平均需要访问 6M/0.2M=30 行。可是，不管实现的视图是否可索引，我们期望回答每一个查询的开销－－整个视图或者单一的单元－－和我们要回答查询的视图的大小成正比。我们会在 Section 3 讨论开销模型更多的细节。

下面有我们现在可以回答的一些有趣的问题：

1、为了获得合理的性能，我们必须要实现多少视图?

2、假设我们有空间 S，为了最小化平均查询开销，我们要实现什么样的视图?

3、如果比起完全实现的数据集，我们愿意忍受 X%的性能下降，比起完全实现的数据库，我们能够节省多少空间?

在这篇论文中，我们提供能帮我们回答下述问题和提供近似最优的结果的算法。

在上述的例子中，一个完全实现的数据库实现了所有的视图，有超过 1900 万行。

现在让我们试试看能否做得更好。为了避免处理原始数据，我们需要实现根据 part, supplier 和 customer 分组的视图（view 1），因为这个视图不能通过其它视图生成。现在考虑根据 part 和 customer 分组的视图（view 2）。用这个视图回答任何的查询都需要我们处理 600 万行的数据。一样的查询总是可以用根据 part, supplier 和 customer 分组的视图来回答，同样也需要处理 600 万行。因此实现根据 part 和 customer 分组的视图没有好处。根据相似的原因，实现根据 supplier 和 customer 分组的视图也没有好处。因此我们可以只用 700 万行获得一样的平均查询开销，就消耗的空间和创建数据库的开销而言，可以拥

有超过 60%的改善率。

因此通过聪明地选择实现数据库的一部分，可以获得最好的效果。

1.3 相关的工作

多维数据的处理（也叫 OLAP）比起先前的版本有很好的提升。有两个便利的 OLAP 的基本实现方法。第一种方法是避免 SQL 和关系型数据库，使用专有多维数据库（MDDB）系统和 OLAP 的 APIs。所以关系型数据仓库的原始数据等同于 MDDB 实现的多维数据集。使用者查询多维数据集，MDDB 通过它的地址高效地检索单元的值。为了分配原始数据中单元的空间而不是多维数据集中每一个可能的单元，我们使用单元定位哈希模式（cell-address hashing scheme）。Arbor’s Essbase [ESS]和很多其它用这种方法实现的 MDDBs。记住，这种方法仍然实现了多维数据集中原始数据的所有单元，所以会非常大。

另一种方法使用关系型数据库系统，并且让使用者可以直接查询原始数据。使用有效的索引和其它传统的关系型查询优化策略来解决查询性能的问题。有很多类似 BusinessObjects 和 Microstrategy’s DSS Agent 的产品使用这种方法。可是，MDDBs 仍然有更好的性能优势。通过把多维数据集变成摘要表格的实现，关系型数据库系统的性能可以被大幅提升。

关系型方法是非常具有规模化的并且能够处理非常大的数据仓库。另一方面，MDDBs 有好得多的查询性能，但是不具规模化。通过只实现选择的多维数据集的一部分，我们能够提高关系型数据库的性能，并且可以提高 MDDBs 的可扩展性。还有即用了关系型方法[STG]，又用了 MDDB 方法（Sinper’s Spreadsheet Connector），也就是说只实现了多维数据集的一部分。我们相信这篇论文是第

一个调查这样的细节的基本问题。

［GBLP95］讨论把 SQL GROUP-BY 运算符推广为多维数据集运算符。她们介绍了我们提及的“ALL”的概念。可是，他们宣称整个多维数据集的大小没有比对应的 GROUP-By 的大小大很多。我们的想法正相反。就像我们在 TPC – D 数据库中看到的，多维数据集通常大很多：比起对应的 GROUP-BY(part, supplier, customer)要大上三倍多。

1.4 文章结构

这篇论文按照下列的方式组织。在 Section 2 我们介绍了格框架来模拟视图间的依赖关系。我们还显示了格框架模拟比涉及的属性任意分级还要复杂的分组。然后在 Section 3，我们提出了我们这篇论文使用的查询开销模型。Section 4 提出一个普遍的技术，它产生基于任意格的实现视图的近似最优选择。在 Section 5，我们考虑重要而特别的“超立方体”格的情况，其中每个视图和分组的属性集合相关。Section 1.2 运行的例子就是这样的一个超立方体。

2 格框架

在本节中，我们定义一个符号用于描述当一个数据立方的查询可以使用另一个查询的结果的情况。我们在插入语中描述其聚合属性来表示一个视图或查询（这是一样的）。例如一个聚合属性为 part 和 customer 的查询将被表示成(part, customer)。在 1.2 节我们看到由超集定义的视图可以用于回答涉及子集的查询。

2.1 查询的依赖关系

我们可以将 1.2 节概括成如下描述。考虑两个查询 Q_1 和 Q_2，我们说 $Q_1 \preceq$

Q_2 如果 Q_1 可以仅用 Q_2 的结果回答，我们也可以说 Q_1 依赖于 Q_2。例如，在 1.2 节中，查询(part)可以只使用查询(part, customer)的结果回答，因此，(part) $\preceq$ (part, customer)。有些查询是不能使用 $\preceq$ 与其它查询做比较的，例如(part) $\not\preceq$ (customer) 以及 (customer) $\not\preceq$ (part)。

而 $\preceq$ 操作符规定了查询偏序。我们将谈论一个数据立方体问题的视图（类似形成格）。为了形成格，任意两个元件（视图或查询）根据 $\preceq$ 的排序必须有一个最小上界和一个最大下界。然而，在实践中，我们只需要假设 $\preceq$ 是偏序的，这样就一定会有一个顶部元素使得所有在其下的元素都依赖于它。

2.2 格符号

我们用元素集合 L（在本文中为查询或视图）和依赖关系由 表示一个格，记做（L, $\preceq$）。对于点阵中的元素 a, b，b 是 a 的祖先当且仅当 a $\preceq$ b。常见的描述方法是用格图来描述格，格图中用节点表示格元素，两个节点 a, b 之间有从 a 到 b 的路径时当且仅当 a $\preceq$ b。图 1 的超立方体表示 1.2 节中讨论的视图集合的格图。

3 代价模型

在这一节中，我们回顾并证明我们之前关于“线性成本模型”的假设，在这个模型中回答该查询的时间成本定义为回答该查询所占用的视图空间。接下来，我们考虑一些未实例化的点并估计它们的视图大小，然后用线性成本模型进行验证。

3.1 线性成本模型

假设（L, $\preceq$）是查询（views）的格，为了回答查询 Q 我们选择一个 Q 的祖先，记为 Q_A，Q_A 已被实例化。因此，我们需要将回答 Q_A 的处理表应用于 Q。在本文中，我们选择最简单的成本模型：即回答 Q 的代价是 Q_A 构建 Q 所用表格的行数。

正如我们在第 1.2 节所讨论的，不是所有的查询都会用到整个视图，例如一个关于销售零件的请求，很大的可能是用户只希望看到针对特定零件或几个零件的销售。如果我们有合适的索引结构，这个视图就很容易构建，然后我们可以在O(1)的查询时间里得到回答。如果我们没有合适的索引结构，那么我们将不得不搜索整个视图，并且对单个零件的查询就可能与查询整个视图的时间相同

例如，如果我们需要从一些祖先视图的查询（零件，供应商）回答一个关于单个零件的查询，我们需要检查整个视图。由此可以看出，该图的单次扫描就足以获得一个特定部分的销售。在另一方面，如果我们需要从祖先视图（零件，供应商）找到每个零件，我们需要聚集这个视图。我们可以使用散列或排序（早期聚合）[Gra93]去聚集这个视图。做聚合的成本是一个与可用内存容量与输入输出行数的比率有关的函数。在最好的情况下，输入一次就够了（例如，当哈希表与主存匹配）。在实践中，已经观察到大部分聚合体需要 1 到 2 次输入。虽然对于单个元件、数量较少的元件而不是整个视图的查询代价是复杂的，但是我们认为做均匀的假设是合理的。我们为这一假设在[HRU95]提供了理由。所以：我们假设所有查询都与给定点阵中的一些元素（视图）相同。

显然，还有其他影响查询代价的因素没有被考虑到，其中有对某些属性的实例化视图集，以及可能存在的索引集。更复杂的成本模型当然是可能的，但我们

相信我们选择的是既简单又实际的成本模型，已足够让我们设计和分析强大的算法。此外，我们在 4、5 节对算法的分析体现了它们在其他成本模型和在我们的成本模型下的性能。[CHRU96]调查纳入了更详细的指数型模型。

3.2 线性成本模型的实验测试

图 5 中显示了我们成本模型的实验验证结果，在 TPC-D 的数据，我们用视图四种不同的粒度查询某个供应商的总销售额。我们发现视图大小和查询的运行时间之间几乎呈线性关系。这种线性关系可以用下式来表示：T = M * S + C。这里 T 是基于视图 S 的查询运行时间，C 是固定成本（基于可忽略不计的视图的查询运行时间），和 m 是算上固定成本后查询时间与视图的比率。正如图 5 所示，这个比例在不同的视图大小中几乎是相同的。

Source	*Size*	*Time*	*Ratio*
From cell itself	1	2.07	-
From view *s*	10,000	2.38	.000031
From view *ps*	0.8M	20.77	.000023
From view *psc*	6M	226.23	.000037

Figure 5: Query response time and view size

3.3 确定视图大小

我们的算法要求知道出现在每个视图中行的数量。有很多方法可以不用实例化每个视图就估算出视图的大小，一个常用的方法是对原始数据中有代表性的子集运行我们的算法。在这种情况下，我们可以通过实例化该视图得到原视图的大小。我们用原始数据的子集来确定我们想实例化的视图。

我们可以使用取样和分析的方法来计算不同视图的大小，如果我们只实例化

点阵（在每个维度聚集了最大属性的视图）中最大的元素 V_1。对一个视图而言，如果我们知道分组属性在统计上是互相独立的，否则，我们可以通过 V_1 或者原数据来分析估计其它视图的大小。一个给定视图的大小是根据不同属性分组的数量，有很多可以用来确定一个关系中不同属性数量的著名取样技术 [HNSS95]。

4 优化数据立方点阵

我们最重要的目标是研发一种当执行视图格时可以优化时空权衡的技术。这个问题可以从多个角度进行考虑，因为我们可以在一个有利的时间，在另一个空间，只要我们用交换出去的东西得到好的“价值”，就有第三方愿意以时间交换空间。在本节中，我们将从一个简单的优化问题开始。其中

1. 我们希望能最小化查询与视图相同的查询集的平均时间
2. 无论在哪种空间，我们被限制只能实例化固定数量的视图

显然，项目（2）不减少空间的消耗，但在 4.5 节，我们将展示如何将我们的技术应用到最优化空间的模型上。即使是在这个简单的场景下，优化问题也是 NP 完全问题：有来自 Set-Cover 的直接减少。因此，我们积极地看待启发式产生的近似解。启发式显而易见的选择是贪心算法，在贪心算法中，我们选择了一系列的视图，其中每个都比它之前的要好。我们可以看到，这种方法总是接近最佳，并且在某些情况下，这种算法可能找到最佳用于实例化的视图。

4.1 贪心算法

假设给定与每个视图相关的数据立方格的空间成本。在本文中，空间成本是

在视图中的行数。设 C(v)是视图 v 的空间成本。我们要实例化的视图集应该总是包括顶层视图，因为没有其他视图可用于回答对于该视图的查询。假设除了顶视图，我们可以选择的视图数量最大为 k。当选择了视图集 S 后，相对于 S 视图 v 的收益记做 B(v, S)，定义如下：

1. a) 对于每个 $w \preceq v$，定义数量 B_w：令 u 是 S 中最低成本的视图，有 $w \preceq u$，注意到 S 是顶视图，则至少有一个这样的视图；

   b) 如果 C(v) < C(u)，则 $B_w = C(u) - C(v)$，否则，$B_w = 0$；

2. 定义 $B(v, S) = \sum_{(w \preceq v)} B_w$

也就是说，我们计算 v 的收益时考虑它如何提高包括它自己在内的视图的评估代价。对于 v 覆盖的每个视图 w，我们比较用 v 评估 w 的代价和用 S 中被认为有最小评估代价的视图评估 w。如果 v 有帮助，就是说，v 的成本低于它的竞争者的成本，那么这个差值代表选择 v 作为实例化视图的部分收益。总收益 B（v，s）是所有用 v 来评估 w 的收益总和，假设所有收益都是正数。现在，我们可以定义选择一组视图来实例化的贪心算法。该算法见图 6：

```
S = {top view};
for i=1 to k do begin
    select that view v not in S such
           that B(v,S) is maximized;
    S = S union {v};
end;
resulting S is the greedy selection;
```

Figure 6: The Greedy Algorithm

实例 4.1： 考虑图 7 中的格，如图所示有 8 个从 a 命名到 h 的视图有空间成本，顶层视图 a 的成本为 100，该视图必须被选中，假设我们希望选中三个以上的视

图。

为了在这个格中执行贪心算法，我们必须选择三个有继承关系的视图来实例化。图 8 中的“第一选择”一列告诉我们在 a 之下每个视图的收益。当我们计算这些收益时，我们假设每个视图都是用 a 进行估价的，则得到代价为 100。

如果我们先实例化 b，那么在 b 及 b 之下的视图 d, e, g, h 都将减少 50 个代价，因此收益为 50*5，即 250，这个计算结果正如图 8 中 b 行的第一列所示。举另一个例子，如果我们先实例化 e，则在 e 及 e 之下的视图 g, h，每个视图将从估价 100 减少到 30，即减少了 70 个代价，因此，e 的收益是 210。

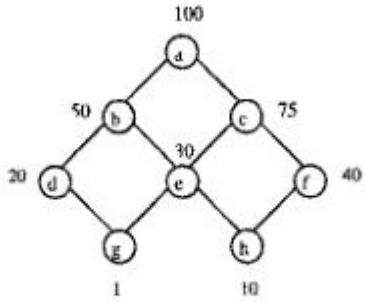

Figure 7: Example lattice with space costs

	Choice 1	*Choice 2*	*Choice 3*
b	$50 \times 5 = 250$		
c	$25 \times 5 = 125$	$25 \times 2 = 50$	$25 \times 1 = 25$
d	$80 \times 2 = 160$	$30 \times 2 = 60$	$30 \times 2 = 60$
e	$70 \times 3 = 210$	$20 \times 3 = 60$	$2 \times 20 + 10 = 50$
f	$60 \times 2 = 120$	$60 + 10 = 70$	
g	$99 \times 1 = 99$	$49 \times 1 = 49$	$49 \times 1 = 49$
h	$90 \times 1 = 90$	$40 \times 1 = 40$	$30 \times 1 = 30$

Figure 8: Benefits of possible choices at each round

显然，在第一轮的胜利者是 b，因此我们选择视图 b 作为一个实例化对象。现在，我们必须重新计算每个视图的收益，假设每个视图要么是从 b 以代价 50

被创建（如果 b 高于 V），要么是从 a 以代价 100 被创建的，每个视图的收益表示在图 8 中的第 2 列。

举个例子，目前 c 的收益是 50，它自己和 f 每个占 25，选择 c 无法再提高 e, g, h 的代价，因此我们不会将 25 的提高计算在内。举另一个例子，实例化 f 所得的收益：对 f 本身而言从 100 到 40 减少了 60 个代价，对 f 之下的 h 而言，因为选择 b 时优化了 h 的估价到 50，因此 h 的估价从 50 到 40 减少了 10 个估价。因此第二轮的胜利者为 f，它的收益是 70。注意到 f 并不是第一轮的最佳选择。

我们的第三选择是总结图 8 中的最后一列，第三轮选择的胜利者是 d，从它本身和 g 中得到总收益是 60。

因此我们可以得到贪心算法的选择为 b, d, f。当然，还有 a，我们从 800 个总视图的估价减少到了 420，这个代价已经是最优的了。

实例 4.2：现在让我们来检测图 9 的格，我们将会看到，这个格和 k = 2 的点阵一样糟糕。贪心算法从顶视图 a 开始，首先选择 c，其收益是 4141，这意味着当我们用 c 代替 a 时，c 和 c 之下共 40 个视图都从 200 个估价提升到了 99。

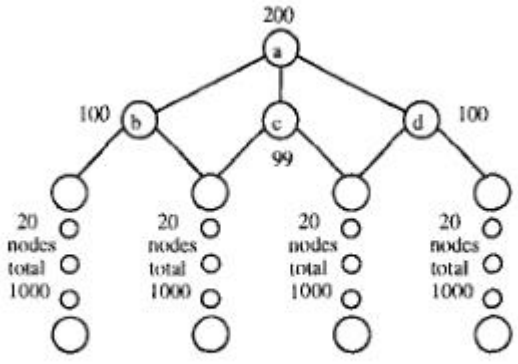

Figure 9: A lattice where the greedy does poorly

对于我们的第二轮选择，我们选择 b 或者 d，他们的收益都是 2100。特别

的，我们考虑 b，它为它自己和它之下 20 个节点每个提高了 100 个估价，则，对于 k = 2，贪心算法产生了收益为 6241 的结果。

然而，最佳选择是选择 b 和 d。这两个一起能将每个节点都提高 100 个估价，算上他们自己和其下 80 个节点，共提高了 8200 个估价，贪心算法和最佳算法的比率为 6241/8000，大概为 3/4,。实际上，令 c 的估价接近 100，令这 4 条链接有任意大量的视图，对于 K= 2 我们可以找到接近 3/4 比率的例子，但是不会更差。

4.2 贪心算法的实验

我们可以在 Fig.4 中的格上通过使用 TPC-D 标准的数据库进行贪心算法的运行。Figure 10 从第一张视图到第十二张还有最后一张视图证明了视图的结果顺序。每个单元的好处，总时间和总的空间消耗都是行数。平均查询时间是总时间除以视图的总数。(示例中是 12)

	Selection	*Benefit*	*Time*	*Space*
1.	*cp*	infinite	72M	6M
2.	*ns*	24M	48M	6M
3.	*nt*	12M	36M	6M
4.	*c*	5.9M	30.1M	6.1M
5.	*p*	5.8M	24.3M	6.3M
6.	*cs*	1M	23.3M	11.3M
7.	*np*	1M	22.3M	16.3M
8.	*ct*	0.01M	22.3M	22.3M
9.	*t*	small	22.3M	22.3M
10	*n*	small	22.3M	22.3M
11.	*s*	small	22.3M	22.3M
12.	none	small	22.3M	22.3M

Figure 10: Greedy order of view selection for TPC-D-based example

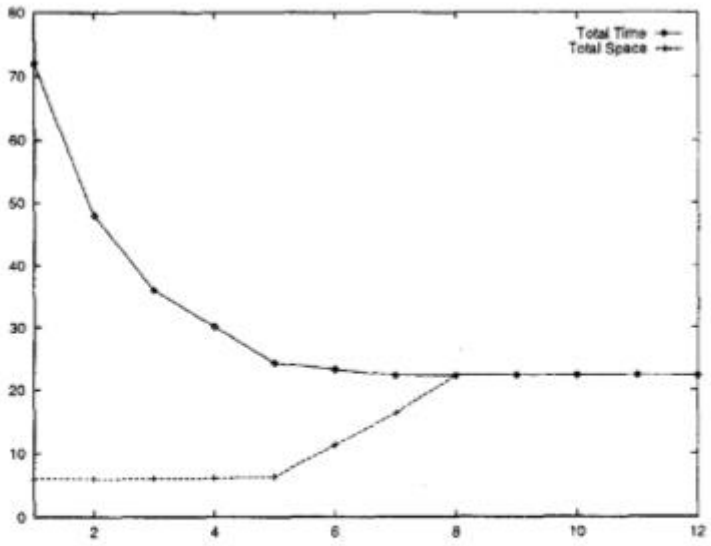

Figure 11: Time and Space versus number of views selected by the greedy algorithm

这个例子证明了物化一些视图是十分重要的，同时也说明了为什么物化所有视图不是一个好选择。Figure 11 有总时间消耗和在 Y 轴上的空间消耗，X 轴上的视图数量。这很直观地显示出选择尽可能少的视图和空间，查询时间会持续下降。当我们选择了 5 个视图时，我们不能通过使用大量数据空间来继续优化查询时间。在这个例子下，我们知道了什么时候应该停止选择视图。如果我们选择了

从头数起的前五张视图，cp, ns, c，和 p 那么我们就会得到几乎最小的可能执行总时间，然而总空间消耗和只选择第一张视图相比并没有大很多。

4.3 保证贪心算法的运行效果

我们可以证明无论给出的是什么格，贪心算法永远都不会执行得很差。具体地说，贪心算法比起最优的算法至少有 63%的结果相似。预测是分数是(e - 1) / e，e 是自然算法的基。

从 $T_\emptyset$ 开始我们的探索，我们需要加强某些叙述。让 m 为格中视图数。假设我们选择了最上面的视图。每个查询的响应时间就是最上面这 $T_\emptyset$ 视图的行数。令时间为。假。设除了最上面的视图，我们还选择了一些视图集合记为 V。令单个查询的平均响应时间为 T_V。V 的优点就是减少平均响应时间，$-T_V$。因为单个查询的评价最小响应时间就等价于这个集合优化时间的最大值。

假设 $v_1, v_2, \cdots, v_k$ 是通过贪心算法选择的 k 个有序视图。让 a_i 为 v_i 的最优解，集合包括最上面的视图和 $v_1, v_2, \cdots, v_{i-1}$ 。令 $V = \{v_1, v_2, \cdots, v_k\}$。

假设 $W = \{w_1, w_2, \cdots, w_k\}$是 k 个视图的最优集合，如获得最大的好处。这些视图的顺序看上去像是抽象的可我们需要构成一个顺序。已知 w 的顺序是 $w_1, w_2, \cdots, w_k$，定义 b_i 是 w_i 在集合中的好处，集合包括了第一张视图和 $w_1, w_2, \cdots, w_{i-1}$。定义 $A = \sum_{i=1}^{k} a_i$ 和 $B = \sum_{i=1}^{k} b_i$。

我们很容易证明通过贪心算法得到的集合 V 的好处 B_{greedy} 是 $T_\emptyset - T_v = A / m$，最佳选择 W 是 $B_{opt} = T_\emptyset - T_w = B / m$ 。在[HRU95]的完整版中，我们知道：

$$B_{greedy}/B_{opt} = A/B \geq 1 - (\frac{k-1}{k})^k$$

例如，在 k = 2 时我们有$A/B \geq 3/4$，就是说贪心算法达到了最优解的 3/4 的性能。在例子 4.2 中对于 k = 0 存在具体的格使贪心算法结果与最优解之比为 3/4。在[HRU95]中我们知道对任意的 k 我们可以构造一个格使$A/B \geq 1 - (\frac{k-1}{k})^k$。

当 k 趋于无穷时，$(\frac{k-1}{k})^k$接近 1/e，所以$A/B \geq 1 - \frac{1}{e} = (e-1)/e = 0.63$。这就是说，没有一个格使贪心算法结果与最优解结果之比小于 0.63。相反地，我们可以构造坏例子的序列使其不能证明比 0.63 高。我们可以在如下的定理上总结我们的结果：

定理 4.1 对于任意的格，让 B_{greedy} 为通过贪心算法选出的 k 个视图的好处，B_{opt} 为 k 个视图的最优解。然后$B_{greedy}/B_{opt} > 1 - \frac{1}{e}$。而且，这个的边界是宽松的，这意味着存在一个格使$B_{greedy}/B_{opt}$ 近似于 1 – 1/e。

一个有趣的地方在于贪心算法是我们希望找到的任意多项式时间算法中最好的。Chekuri [Che96]已经证明了，使用近期发表的论文 Feige [Fei96]，无论 P= NP，没有一个多项式时间算法可以保证一个比贪心算法更好的边界。

4.4 贪心算法是最优解的例子

4.3 节的分析让我们发现贪心算法是最优解或近似最优解的一些实际存在的例子。这里有两种情况，这些情况下我们从未探究过比贪心算法更有效的解法：

1. 如果 a_1 比其他 a’s 都大，那么贪心算法就是近似最优解。

2. 如果所有的 a’s 都一样大，那么贪心算法就是最优解。

这些声明的判定都是基于[HRU95]定理 4.1 的证明的。

4.5 基本模型的拓展

我们的模型在实际情况下至少存在两种情况使其失效。

1. 一个格的视图在一个查询中基本不可能得到相同的概率去响应。而且，我们之所以可以将每个视图的概率相加其实是需要得到其频率的。

2. 我们会分配一些视图空间(在必须被物化的第一张图上我们不考虑这种情况)而不是考虑分配视图的一些固定的数字。

第一点需要其他的一些额外想法。当计算好处时，我们会计算每个视图的概率。贪心算法将在其表现上有相同的最大值 63%。

第二点表示了另外一个问题。如果我们没有现在选择的视图的数量，但是限制其空间，那么我们需要考虑每个视图在用于物化的单位空间上得到的好处。贪心算法看上去还是适合的，但是一个额外的纠纷就是我们可能会有一些小视图却在单位物化空间上有很多的好处，而大视图的单位物化空间好处则很小。此时只能选择小视图而不是大视图，因为当我们选择完小视图时我们已经没有足够的空间去选择大视图了。然而，我们在[HRU95]中证明了下列定理，定理指出如果我们忽视了边界情况，那么贪心算法的运行情况实际上和普通的例子是差不多的。定理可以允许我们假设上述情况下贪心算法的单位物化空间好处是差不多的。

定理 4.2 用 B_{greedy} 表示好处而 S 表示一些用贪心算法选择后视图组合占用的空间。B_{opt} 表示不占用像 S 这样大空间的最优解视图组合。$B_{greedy}/B_{opt} > 1 - \frac{1}{e}$ 是成立的，而且边界是宽松的。

6 结论和未来工作

在这篇论文中我们已经观察讨论了为了将查询响应时间最小化选择哪个数

据立方集合的问题。对决策问题及其应用而言，视图的物化是一个必要的查询优化策略。在论文中，我们严谨地讨论了成功的策略与正确的选择之间的关系。我们使用了 TCP-D 数据库做完一个样例数据库去展示为什么选择物化某部分数据立方而不是全部数据立方是十分重要的。

我们的第二个成就就是建立了对多维分析十分有效的格框架。我们的贪心算法可以在这个格下正常工作而且有约束地选择正确的视图去物化。我们给出的贪心算法在许多约束下对一个重要的持续的因素为找到最优解决方案进行深入的分析。而且，[Che96]已经证明了没有多项式时间的算法比贪心算法更有用。最后，我们找到了超立方体格中最常见的例子而且具体分析了时间和空间上的消耗。在某种意义上，视图形成了在不同达到时间的内存继承。在一般情况下的内存继承中，数据往往会因为根据不同运行时间的需要被不同类型的内存载体保存(如缓存，主存)。我们目前在观察探究数据立方的同样的动态物化。

鸣谢

我们十分感谢 IBM 的 Bala Iyer 和 Piyush Goel 对实验的支持，同时感谢 Chandra Chekuri 和 Rajeev Motwani 对论文的评论。