0.摘要

✓1.介绍

✓2.背景工作与动机

✓2.1.Parallel-Database相关

✓2.2.OLAP相关

✓2.3.动机

✓3.并行化的数据聚合立方体生成

✓3.0.一些数量的假定

✓3.1.内存模型

✓3.1.1.并行编程模型与多线程模型的相异点

✓3.1.2.基于读写互不冲突原则的内存分配

✓3.2.memory最优方案

✓3.3.速度最优方案

✓3.4.混合方案

4.从元数据聚合立方体到目标数据聚合立方体的优化

✓4.1.目标与评估标准

✓4.2.聚合路线的贪心选择算法

✓4.2.1.算法的描述

✓4.2.2.算法的证明

✓4.3.从给定的预处理cuboid中的最优起始选择

✓4.3.1.选择标准

✓4.3.2.选择算法的实现

✓4.4.最优化预处理cuboid生成

✓4.4.1.古典方法（Stanford， 1997）

✓4.4.2.对花费函数的修正

✓4.5.聚合的并行化实现

5.评估

✓5.1.实验环境

？5.1.1.由实验环境进行的参数选择测试

5.2.实验结果

？5.2.1.对于底层cuboid生成算法的评估

？5.2.2.对于由cuboid到cuboid的聚合算法的评估

？5.2.3.对于聚合路线选择算法的评估

？5.2.4.花费函数在不同实验平台上的实际计算

✓5.3.实验分析与讨论

✓5.3.1.何时使用生成并行化的数据立方体生成的混合方案

✓5.3.2.对于修正后的花费函数的评估

✓6.结论

✓7.引用

1.

在现代计算机系统及其应用中，数据量的增大已经成为了一个必然的趋势。伴随其而来的计算量的增加，使得越来越多高性能相关的新兴硬件／软件技术的出现成为势在必行。

在高性能计算领域之中，并行计算已经成为了一个很重要的工具。各种新兴硬件的出现，以及伴随它们而诞生的各种并行编程模型，都为高性能计算领域的向前发展提供了十足的力量。

而随着数据规模的增大，对于大量数据的管理、分析也在逐渐成为一项重要的需求，不仅仅是因为本身，更是因为这个领域所做的工作将是所有未来计算机领域的基石。其中作为代表的数据库领域，近些年来不断致力于发展新的数据分析技术，改进数据关联的基本操作与算法，以及更多地尝试新的硬件平台，层次结构，在这些方面都取得了长足的进展。

近些年来人们也开始在各种方面开始尝试将这两者结合在一起，但是由于现行并行计算编程（譬如大计算量，无相关性，弱控制）与传统数据库领域（譬如ACID特性）在某些理念上的冲突，使得在数据库的并行优化方面，还有很多可以开展的工作没有得到人们的充分认识。

本文尝试从在数据库领域比较重要的一个领域：OLAP出发，探寻利用并行计算硬件与编程思想，来尝试从源数据生成数据立方体，以及从数据立方体出发如何生成别的数据立方体这两方面进行优化

本文的贡献可以从以下几个角度来阐述：首先，本文给出了一套基本的时间空间复杂度估计函数，并在这个基础上提出了生成最底层cuboid的时间／空间最优算法，以及结合它们的优点，提出了平衡各自缺点的混合型方法；其次，本文给出了一套如何评估cuboid生成的时间的算法，（并且基于并行计算的基础改进了现有的生成预处理cuboid的方法）

本文接下来将按照如下方式组织：第2章介绍已有的工作成果与本研究开展的动机，第3章介绍用并行计算从源数据生成底层cuboid的数种方式并分析其优劣，第4章给出一套从源cuboid到目标cuboid的生成方案及其优化，第5章对这个工作进行了评估，第6章将总结本文。

2.1.

并行计算与数据库

并行计算是一个发展历史悠久的领域，其最早的发源要追寻到上世纪五十年代后期。上世纪六十年代至七十年代，共享内存的多处理器体系结构出现了。在上世纪八十年代，大规模并行处理器的出现开始占领市场，然而旋即又被集群所取代。发展至今天，并行计算硬件的主流已经成为了多核处理器[1].

在这个主流下，越来越多的硬件被发掘出来参与到并行计算中来。图形处理器（GPU）最初是被开发来实现以电子显示为主的各方面的图形演算与生成的，然而由于其架构满足流水线深，可利用核心多等特点，很快人们就开始研究GPU的通用用途（GPGPU），其中一个很重要的领域就是GPU参与的并行计算。2006年，NVIDIA首先提出了CUDA，并在2007年首次正式将其发布，这也成为了世界上首个在GPU上进行通用并行编程的平台[2]。在不久之后的2009年，苹果公司也由Khronos Group进行开发，提出了自己的一套通用并行计算平台OpenCL[4]。这一套开发平台不仅仅局限于对GPU的并行潜能的挖掘，而是在此基础上支持了更多的新兴硬件，例如FPGA，DSP等。并行计算在这些优秀平台的支持下开始在各个领域蓬勃发展。

鉴于并行计算其尤其适合对于大量数据的高密度计算的特征，在数据库相关领域，近些年来也有研究者开始了试图将数据库与并行计算硬件（主要是GPU）结合在一起的研究。B. He等人所在的研究组作为最早涉足该领域的研究组之一，在2008-2009年之间出版了相关论文对于关系型数据库在GPU上的可能性进行了探讨。[5]中提到了在GPU上进行的对于关系型数据库join操作的并行优化，而[6]最早地提出了一套系统的GPU上评估查询执行效率的模型，还有一套完整的在GPU上进行的query plan的生成方案。这两项研究成果至今仍被相关研究领域的人员频繁引用。另外，[7]中也对传统关系型数据库中的事务处理进行了适合并行架构上的优化。

在接下来的几年，有许多的研究成果基于前述的成果，在并行计算和数据库的结合方面做出了很多的新的尝试。这其中有很多的工作集中在了对于query的高效处理的方面。[8]中首次给出了一种对于并发处理query的解决方案，改变了从前至多只能并行处理一个委托的不同部分，从而使得总体硬件资源利用效率低下的状况，使得query之间可以有机会被并行处理，极大地提高了硬件的利用率，提高了query的吞吐量。[9]中则尝试从另一个角度挖掘并行硬件的架构特点，通过开发一套对于多个query的流水线处理系统以及与之相协调的并发内核控制和数据交换方法，来使得并行硬件在单位时间内处理多个query的性能得到了大幅度的提升。另外，本文也给出了一套query处理流水线化的分析评估模型，为后来者的工作打下了基础。

总的来说，近些年来在数据库领域，并行计算作为一个新生并且强有力的事物，正在越来越多的方面发挥着其强大的性能，为数据库系统的方方面面保驾护航

2.2.

OLAP，数据仓库及其他

在数据库领域，一个历史悠久的课题就是，对于大规模数据仓库，如何进行有效的管理，分析等一系列数据相关的操作。在这之中不得不提到的就是OLAP（On-Line Analytical Processing，线上分析处理）。这项技术将大量的原始数据聚合到特定的数据立方体中，在数据立方体中进行各种数据的分析，统计等等操作将比在原始数据集合中来得更加方便，快捷。[10]

对于OLAP的研究，同样开始于很早以前。[11]很早地从各个方面对于OLAP技术进行了一个全面的调研，并且指出了OLAP技术将来具体发展的形式。[12]则算是最早的具体研究如何高效快捷的生成OLAP的核心——数据立方体的论文之一。这篇论文首次提到“数据立方体中的cuboid并非每次都需要从原始数据生成”，并且基于这个观点给出了一套行之有效的预处理cuboid生成的评估方案，以及在此评估方案下的一套次最优的贪心算法。本文的其中一部分工作将基于他们的工作进行扩展与延伸。

近些年来，随着各种更新的系统架构，诸如分布式集群等的出现，OLAP从最开始的单机处理单机分析开始，渐渐打开了新的发展方向。[13]则具有远见卓识地基于in-memory database的视点，对OLAP，以及OLTP，在它们运行的基本存储架构上进行了一系列的优化和创新，这也使得因此而诞生的SAP HANA[14]很快成为了包括in-memory database在内，目前世界上做的最优秀的in-memory platform之一。也有很多的文章着眼于并行计算的视点来考虑如何进行数据仓库相关一些操作的优化。[15]作为他们之中最早也是最具体的研究成果之一，系统地给出了一套评估用并行硬件优化数据仓库相关操作的性能的标准。同一时期也有一系列文章则从不同的方面考虑了对于并行架构中的OLAP相关操作的优化。其中[16]通过适当的控制在cache中的query的调度从而使得CPU-GPU架构之间的通信与运行效率提高了，从而使与之相关的OLAP相关操作性能也得到了提升。[17]则通过调整query候选，以及query中间结果在各个并行硬件单元上的分配，与[6]提出的评估模型不同，放弃了评估，转而采取即时地解决这些问题，从而使得其在OLAP的一些相关问题的处理上有了性能上的提升。

综上所述，OLAP是一个由来已久的问题，早些年的研究多是基于OLAP算法本身，而近些年来的研究更多专注于如何将其与现代的新兴系统架构，新兴硬件等等结合在一起，来让OLAP在现代技术的支持下展现出全新的活力。

2.3.

上述的研究背景中，我们可以发现：更多的研究要么专注于如何运用并行计算平台对并发／并行query执行进行优化，要么专注于为在新的系统架构上部署高效的OLAP算法而设计具有针对性的软硬件结构的优化，似乎并没有一项研究来针对如何具体的运用并行架构体系来优化OLAP中数据立方体的生成算法本身。因此本研究就尝试着从这个领域着手，对这一问题提出自己的见解。

3.0.

在接下来的讨论中，本文会涉及到一些计算，其中有一些时间，以及空间的量需要在此列出，以方便计算与讨论。这些量包括：

在系统中并行执行的并行kernel数量：K

数据具有n个维度属性，其中每个维度的取值数量由（d1, d2, …, dn）决定

在数据聚合立方体中一个cell的大小：s

一个kernel负责生成的cell个数：g

数据集的大小：d

一个kernel完整扫描一个数据集所需要的时间：t\_scan

一个cell在内存中拷贝所需要的时间：t\_copy

将一个已经生成的数据聚合立方聚合进另一个数据聚合立方中所需要的时间：t\_agg

数据集被划分成的部分数：p

3.1.1.

目前在并行编程模型中常使用的库中，大多拥有“kernel”这一概念，即并行算法执行的单元。所有的并行算法以kernel为基础，通过并行硬件对于多个kernel的同时计算从而达到并行计算的目的

kernel比起通常的多线程模型，有如下的相异点：

1、kernel属于元程序，即kernel的执行是不可分割的

这使得kernel在执行之中属于不可控状态：kernel直到执行完成为止都无法被中止

2、现有的编程模型中，并没有kernel对于临界区数据的锁机制

从而使得kernel之间对同一内存区域的协同访问／修改变得困难

3、在任意一个时刻，并行硬件上的一个计算单元中最多存在一个运行中的kernel

从而使得对于kernel的任务调度变得更为死板，需要事先完成

4、然而也是由于以上原因，kernel比通常的多线程模型更加易于控制，于是现代并行编程模型中主流即采用kernel模型而非一般意义上的多线程模型，因为高性能硬件（例如GPU）通常来说都被设计成缺少足够的控制处理器从而提高计算处理器的数量，使得最大计算潜能得到提升

3.1.2.

由前所述，在以kernel为基础的并行模型中，对共享数据的内存管理是一件不容易的事情，特别当面对读写冲突（例如RAW）时会变得难以控制。因此，直观的一种解决思路（同时也是现在比较常用的解决思路）便是事先划分每一个kernel所使用的内存区域，这样就可以在不造成读写冲突的情况下最大化并行kernel之间的并行度

在下文中，将提到三种不同的由源数据生成底层cuboid的方案。这三种方案采用的内存模型都基于上述的内存互不相关原则，并且将会在方案被具体展示时被具体提出

3.2.

kernel以cell为划分

（具体文本之后添加）

·所有kernel共用一个shared memory数据聚合立方体

·每个kernel仅操作被调度方式分配到的那些cells

算法运行的步骤：

0.根据调度方案生成kernel

这一步需要消耗的时间：C（一个系统相关的值） + t\_copy \* Πdi（将空白数据聚合立方体空间拷贝给kernel）

这一步中存在的最大空间占用：d（数据集的大小） + Πdi \* s \* 2（数据聚合立方体的空间）

1.如果仍存在待处理的kernel，转2；否则转4

2.对于每一个正在并行处理的kernel，需要完成以下操作

2.1.完成一次对于数据集的完整scan

2.2.以扫描的结果生成对应的cells（通过直接修改shared memory中的对应cells的内存实现）

对于每个kernel，其运行时间为t\_scan

较上一步没有额外的空间占用

3.转1

4.将并行模型中的计算结果（生成完毕的数据聚合立方体）由并行模型中拷贝到主存

这一步需要消耗的时间：t\_copy \* Πdi（将生成的数据聚合立方体空间拷贝给主存）

这一步存在的最大空间占用：d（数据集的大小） + Πdi \* s \* 2（数据聚合立方体的空间）

5.（由于只存在一个part的数据聚合立方体，因此多part的数据聚合立方体进一步聚合成为一个立方体的操作免去）

考察：总共有Πdi / g个kernel需要被执行以保证所有的cells被生成，而同时并行执行的kernel数量为K，步骤2的总时间开销应该为：t\_scan \* (Πdi / K / g)，并且注意到K不可能大于Πdi / g，即有约束条件Πdi >= g \* K

综上：

总的运行时间： t\_scan \* (Πdi / K / g) + C + 2 \* t\_copy \* Πdi (+ 0 \* t\_agg)

过程中最大的空间占用：d + 2 \* Πdi \* s

3.3

kernel以dataset part为划分

（具体文本之后添加）

·所有kernel独自使用一个独立的shared memory区域（类同于local memory）

·每个kernel负责生成一个part的dataset的完整数据聚合立方体

·在最后将所有part的数据聚合立方体合成为一个对全dataset的数据聚合立方体

·在这个算法前提下，可以决定这个dataset需要划分为多少个kernel来执行，假定为k0 >= K

0.根据调度方案生成kernel

这一步需要消耗的时间：C（一个系统相关的值） + t\_copy \* Πdi \* k0（将空白数据聚合立方体空间拷贝给kernel）

这一步中存在的最大空间占用：d（数据集的大小） + Πdi \* s \* (k0 + 1)（数据聚合立方体的空间）

1.如果仍存在待处理的kernel，转2；否则转4

2.对于每一个正在并行处理的kernel，需要完成以下操作

2.1.完成一个partition scan

2.2.以扫描的结果生成对应的cells（通过直接修改shared memory中的对应cells的内存实现）

对于每个kernel，运行时间为t\_scan / k0

每个kernel占用了一个大小为Πdi \* s的shared memory区域，以及共用的dataset，以及主存中等待写回的k0个Πdi \* s内存区域

3.转1

4.将并行模型中的计算结果（生成完毕的数据聚合立方体）由并行模型中拷贝到主存

这一步需要消耗的时间：t\_copy \* Πdi \* k0（将生成的数据聚合立方体空间拷贝给主存）

这一步存在的最大空间占用：d（数据集的大小） + Πdi \* s \* (k0 + 1)（数据聚合立方体的空间）

5.聚合part立方体：第一个part将直接被写入结果数据聚合立方体中，接下来的k0 – 1个数据立方体将被聚合进这个结果数据聚合立方体中

这一步需要消耗的时间：(k0 - 1) \* t\_agg

这一步中存在的最大空间占用：d（数据集的大小） + Πdi \* s \* (k0 + 1)（数据聚合立方体的空间）

考察：k0个kernel需要执行k0 / K轮才能执行完毕，则步骤2的总运行时间为t\_scan / k0 \* k0 / K = t\_scan / K，而在其余计算式中亦能看到k0取值越小越有利于降低各方面的开销，因此可以简单地令k0 = K

综上：

总的运行时间：C + t\_scan / K + 2 \* t\_copy \* Πdi \* K + (K - 1) \* t\_agg

过程中最大的空间占用：d + Πdi \* s \* (K + 1)

3.4.

kernel同时以part和cell划分

（具体文本之后添加）

·一个kernel处理对应dataset part的某些cells的生成

·内存的占用方式依据kernel的具体执行顺序来决定：

若优先执行具有相同cells划分的那些kernels，则同时执行K个kernel，它们都需求g个cell的local memory空间，则此处需要额外分配给kernel的空间为K \* g \* s

若优先执行具有相同part的kernel，则无所谓执行一次执行多少kernel，在所有同一类的kernel执行完毕之前，它们共享一个partition data cube，它们需要额外分配空间为Πdi \* s

·如果能够控制先执行哪些kernel，此处选择简单的方式二，这个时候主要要根据K \* g和Πdi的大小关系来决定需要几个partition data cube的空间

简单来说，就是对K \* g / Πdi 的上取整再+1，可以证明在这个前提下每轮执行都能满足所有kernel的独立读写需求

·如果不能够控制kernel的先后执行顺序，则只能如同方式2一样使用最大的内存分配

·在这两种前提下，无论哪种，最终都要经历p – 1次全data cube上的aggregation，以及p次全data cube的拷贝（入与出），因此可以认为运行时间是相同的

综上：

总的运行时间：C + t\_scan \* Πdi / g / K + Πdi \* p \* t\_copy \* 2 + t\_agg \* (p – 1) \* 2

方式1：g \* K <= Πdi & p = 1

方式2：p = K & g = Πdi

可以看到与之前的分析契合

最大内存分配：d + Πdi \* s + Πdi \* (ceil(K \* g / Πdi) + 1)

为什么在内存分配上会出现与方式一与方式二的错位？

因为在此方案中，并不能保证某n批次的kernel处理完毕时正好落在某个part的边界，因此需要额外的内存空间来作为其他的partition data cube的空间以保证聚合的正确

4.1

评估标准：

假定n个维度，分别具有维度值为(d1, d2, …, dn)，给定一个聚合顺序为(i1, i2, …, in)，则

第一步聚合的维度为i1，聚合所需要扫描的聚合cell自然是Πdi

而该步聚合后，新的聚合立方体具有cell的个数为Πdi (i != i1)

如果我们认为扫描一个cell的时间都是相等的（如同最开始的假设），则容易得出总的扫描数->聚合时间为：…

目标：找出一个目标方案，使得从源cuboid到目标cuboid的聚合时间最小

4.2

4.2.1

算法的描述：我们总选择当前维度值最大的那一维来进行聚合

4.2.2

现在假定一个聚合过程中的某两步，选择聚合第i维和第j维

考察这两步聚合发生前的聚合立方体与之后的数据立方体：与这两步聚合的具体顺序无关，结果来看都为后者比前者多聚合了i，j两个维度,因此，对于这两步之前的扫描次数总和，与这两步之后的扫描次数总和无关系

假定i，j两步聚合后的聚合立方体还有cell的个数为D，以及i先被聚合

这两步的总扫描次数：D \* di \* dj + D \* dj

类似的，j先被聚合时得到的是： D \* di \* dj + D \* di

此时可以看出，先聚合di、dj中比较大的那一个对应的维度时会得到更小的总扫描次数

因此，给定任意一个序列，除非其已经是严格按照如下的标准来完成聚合，否则总能找到一次调整，使得调整后的聚合序列拥有更少的总扫描次数

序列对应的维度值是严格单调非增的

而满足最少总扫描次数的聚合序列，可以由一个简单的贪心算法得到（前述）

证毕

4.3.

事实上，对于一个系统而言，存在忙时就势必会存在闲时，意即存在这样一些时刻，系统计算资源得到空闲。而由4.2的分析可知，从源cuboid到目标cuboid是需要计算的，这个计算量与聚合操作的次数，路径上的中间cuboid都有关。

作为4.2.中提到的算法的前提条件，“源cuboid”与“目标cuboid”是明确的，即从源cuboid到目标cuboid的聚合次数是一定的，而4.2.算法在这个前提下给出了最小代价的聚合方案。然而在实际操作中，用户只需求目标cuboid，意即其并不关心这个目标cuboid是从哪个地方来的（可能是直接从源数据生成，也可能是由其他一些已经生成好的cuboid生成）。于是，一个直观的想法就是我们或许可以不需要每次都从最底层的cuboid来现场计算出目标cuboid，而是从某些预先计算好的cuboid来计算目标cuboid，从而减少聚合次数，进而减少总的计算代价。这些预处理的cuboid可以在系统空闲时间被计算出来，从而不占用任何的实际运行时间。

4.3.1.

如前所述，假定我们现在已经有了一个预处理过后的cuboid集合。可知这个集合中势必包含一个最底层的cuboid

任意给定一个目标cuboid，一个预处理的cuboid能生成这个目标cuboid的充要条件是：预处理cuboid中不存在这样的已聚合维度，使得目标cuboid中该维度未聚合。（\*）

沿用4.2.的计算代价，我们可以简单地得出我们问题的定义：从满足（\*）的cuboid中寻找一个使得由4.2.计算式能给出最小值的cuboid，此cuboid即为所求。

4.3.2.

这个问题可以由一个并行算法实现，每个kernel简单地使用一个位于共享内存中的内存单元以存放最后的计算结果。每个kernel需要完成的计算如下：

1、并行计算设备读入目标cuboid和一个预处理集合中的cuboid。这里只需要读入它们对于维度的描述

2、先进行判断：该预处理集合中的cuboid是否能生成目标cuboid

3、如果能，由4.2.式计算出其计算代价，并将其存回共享内存

4、所有kernel执行完毕后，共享内存中的计算代价结果拷回CPU，并由CPU对这个数组进行评估来选择最适合的预处理cuboid

事实上，由于预处理的cuboid未必很多（每个cuboid本身是会占用不少存储空间的），而其中维度数据占的比例又非常小，因此这个算法即使全部交由CPU来执行也是可行的

4.4.

如前所述，在考虑cuboid的生成问题时，没有必要每次都从最底层的cuboid来进行生成，而是可以从预先预处理好的cuboid中寻找一个较为合适的，在被选择的cuboid的基础上更快更好地生成所需的目标cuboid。这一小节从经典的cuboid预处理选择生成算法出发，在新的硬件模型上对此进行了一定的拓展。

4.4.1.

在。。。中，一种以硬盘I/O时间作为主要衡量因素的评估标准，以及由此而生的一种生成次最优解的贪心算法被提出。具体的算法描述如下：

假定每个cuboid（下文称为v）在进行处理时拥有一个花费函数C(v)，同时假定在除了最底层的cuboid之外，预先处理好的cuboid的个数为k。集合S作为当前已选择的cuboid的集合。

定义一个收益函数B(v, S)，具体定义如下：

对于每一个能从v聚合得来的w，定义Bw为

1. 令u是在S中的能聚合生成w的且拥有最小的花费函数值的cuboid

2. 如果C(u) > C(v)，则Bw = C(u) – C(v), 否则Bw = 0

B(v, S) = ΣBw

由此我们可以看出，收益函数的定义实质是“选择当前cuboid比起选择由它而生成的所有cuboid相比能多得到的收益”

基于这个收益函数，有一个对于所需生成的cuboid进行选择的贪心算法如下：

1. 初始状态：S = {最底层cuboid}

2. 如下的操作进行k次：

2.1. 选择通过计算B(v, S)能得到最大值的，并且不在S中的v

2.2. S = S union {v}

3. 返回的S集合即为所求的S集合

4.4.2.

4.4.1.中提到的贪心算法，可以看到其最终执行的效果与C(v)的选择息息相关。在前述论文发表的时代背景中，可以认为“从硬盘到内存中的对于预处理cuboid的存取”占用了绝大多数的时间，意即存储之间的I/O成为了整个系统的绝对瓶颈；另一方面，I/O时间在系统硬件配置恒定的情况下基本只与存取的数据的大小有关，具体到每个数据聚合cuboid上，就是和cuboid作为一张table所具有的行数（cell的数目）有关。因此，在原论文中，这个花费函数被简单的设定成为了该cuboid的行数。所有的计算都通过“I/O数据量最优”来间接地指向“时间最优”。

然而，上述的假定并不周全。我们再来详细考察从一个事先预处理好的cuboid到目标cuboid的过程：

1. 从存储介质（例如硬盘）中拷贝相应的cuboid到内存中以备调度

2. 向并行计算设备发出计算请求时，对应的cuboid将被拷贝到并行设备上

3. 并行设备计算下一步的cuboid

4. 计算完毕的结果将被拷贝到主存中，如果仍然没有得到目标cuboid，重复2-4步

因此，对于一个完整的过程，我们能得到每一步的计算时间表达式为：（假定目标cuboid有D个cell，从预处理的cuboid到目标cuboid一共聚合了维度q个，它们的维度值分别为d1, d2, …, dq，一个cell的大小为s）

1. C\_copy\_hd\_to\_m \* D \* Πdi \* s

2-4三步将如下考虑：每一步执行的cuboid都比上一步少了一个维度，具体少的维度可以由4.2.的方法给出，这里将k次的时间总和写在一起：

2. 初始化k次 + 总共拷贝的cell量为Σ((D \* Πdi \* s) / (d1 \* … \* dj))（对j求和），这里j代表第j次初始化，取值范围为从0到q - 1

-> T2 = C\_init\_time \* k + C\_copy\_m\_to\_d \* Σ((D \* Πdi \* s) / (d1 \* … \* dj)) \* s

3. k次计算cuboid的时间总和为：Σ((D \* Πdi) / (d1 \* … \* dj)) \* t\_scan\_per\_cell / K，其中K如3.0.所述。具体到第j+1次计算为何所需时间是(D \* Πdi) / (d1 \* … \* dj) \* t\_scan\_per\_cell / K，在后文的4.5.将进行分析

4. k次拷出，总共拷贝的cell量为Σ((D \* Πdi \* s) / (d1 \* … \* dj))（对j求和）。因此这一步的花费时间为：C\_copy\_d\_to\_m \* Σ((D \* Πdi \* s) / (d1 \* … \* dj)) \* s，这里j的取值范围从1到q

由此我们能得到一个总的时间计算表达式作为C(v)。其中我们可以发现，出现了一些新的常量，而这些常量与4.5.中提到的聚合方法，以及系统的一些具体硬件参数是有关的：

C\_copy\_hd\_to\_m：指从硬盘向内存中拷贝单位大小的数据所需要的时间

C\_init\_time：初始化由4.5.指定的cuboid聚合函数所需要的时间

C\_copy\_m\_to\_d：从内存向并行设备拷贝单位大小的数据所需要的时间

t\_scan\_per\_cell：4.5.中决定的聚合方法中，单个kernel对于每一个聚合前的cell进行扫描以及聚合进聚合后的cuboid的平均时间

C\_copy\_d\_to\_m：从并行设备向内存拷贝单位大小的数据所需要的时间

关于这一小节留下来的这些常量，它们如何取值很大程度上决定了我们该如何在前人论文的基础上修正我们对于cuboid的选择方式。它们的具体取值将会在第五章通过实验来推定，从而在此基础上再来回顾修正前后的花费函数对于我们最终生成预处理cuboid的决策的影响

4.5.

前述的段落中，我们阐明了如何选择聚合的各个因素，从而使得聚合的平均性能尽可能达到最优。本小节将讨论如何具体实现从源cuboid到目标cuboid的聚合

由4.4.，我们计算出了一个使得平均聚合代价最优的预处理集合；由4.3.，我们找到了一个能最优化对于给定目标cuboid的生成的预处理cuboid，并且在此同时由4.2.得到了一条从选择出的预处理cuboid到目标cuboid的生成路线。因此，这一步要做的就是按照这个路线，逐个逐个地由预处理cuboid生成到目标cuboid。可以发现这个问题每个步骤其实是相似的：聚合当前cuboid的某个给定的维度。因此下文的讨论中仅讨论一步的聚合。

进行如下的假定：对于当前cuboid，未聚合维度的维度值集合为(dd1, dd2, …, ddj)。假定我们需要聚合维度di，考察聚合后的cuboid，我们可以发现，所有满足如下条件的聚合前的cuboid的cell将会被聚合成聚合后cuboid的一个cell：

(ddk1, ddk2, …, ddk(i-1), di, ddk(i+1), …, ddkj)，其中ddkm（m从1到j，m代表维度）是明确的值，di是一个变量。它们将会被聚合到聚合后的cuboid中的(ddk1, ddk2, …, ddk(i-1), ddk(i+1), …, ddkj)这个cell中

因此，我们可以令每个kernel生成一个聚合后的cell。注意到所有的kernel合起来将完成一次对于聚合前cuboid的扫描，因此计算代价上是最优的，无需像第3章中一般特地使用多个cuboid的存储空间，而仅需要一个共享的cuboid空间即可。同时这个定义如同3.2.提到的那样，可以直接避免读写冲突，因此这是一个可行的方法。

总结起来，对于每个kernel，其需要执行的事情如下：

扫描其对应的cell，将它们聚合到一个cell中

由此可以得到时间开销与空间开销为：

时间：ddk1 \* ddk2 \* … \* ddkj \* t\_scan\_per\_cell / K

空间：ddk1 \* ddk2 \* … \* ddkj \* s + ddk1 \* ddk2 \* … \* ddk(i-1) \* ddk(i+1) \* … \* ddkj \* s

注意到在该聚合方法中，对于每个聚合前cuboid的每个cell，我们简单地取出内部的数据，并将其累计进入对应的聚合后的cell，那么基于前文所假定的cell结构相似性，我们可以认为t\_scan\_per\_cell在任何阶段都是一定的。

由得到的时间开销表达式，也可以回答在4.4.2.中留下的关于毎步计算时间开销表达式的问题。

5.

在本章中，由于OpenCL的多平台性，我们将在两个不同的平台上进行：一个是CPU上的并行计算，另一个是GPU上的并行计算。通过同样的一个OpenCL程序在两个不同的硬件架构上的运行来确定在不同的情况下各个在4.4.2中提到的常数，并用这些常数具体评估在不同硬件平台上的运行性能，并对这些结果进行对比。

5.1.

本次实验采取了两种并行硬件架构来完成实验：

1. 纯CPU架构运行在如下的硬件上：Intel Core i5 Dual Core Processor@2.9GHz，8GB 2133MHz LPDDR3 memory，512GB SSD。

2. CPU-GPU混合架构运行在如下的硬件上：ATI Radeon HD 8570， Intel Core i7-6700@3.4GHz（8 cores），8GB 2133MHz DDR3 memory， 128GB SSD

5.1.1.

在进行实验之前，首先需要对4.4.2.中提到的诸常数进行考察，从而明确在算法运行的全过程中哪方面将成为性能瓶颈。

对于C\_copy\_hd\_to\_m

对于C\_copy\_hd\_to\_m的考察可以通过读取的文件的大小来确定。我们可以采用一个“纯粹的读取程序”——即只将数据读入内存，在读取的过程之中不进行任何多余的操作——来完成求出这个常量的实验。

因此，我们直接简单地利用现有程序中的“从硬盘中读入数据集”这一步来进行系数的计算。对不同大小的数据集进行读入所需要花费的时间为：（均为10次的平均数值）

在纯粹CPU并行架构上：

|  |  |  |
| --- | --- | --- |
| 数据条目数 | 总的数据集文件大小 | 平均读取用时（单位：s） |
| 262144 | 3,479,769 bytes | 0.3520096 |
| 2097152 | 27,841,394 bytes | 2.741692 |
| 16777216 | 222,741,273 bytes | 21.11312 |
| 134217728 | 1,781,907,696 bytes | 166.6692 |

取这四个数据点进行拟合，能得到直线的斜率，即我们所需的C\_copy\_hd\_to\_m的估计值为9.34613E-8 s / byte

在CPU-GPU并行架构上：

对于C\_init\_time

在纯CPU架构上：通过对不同大小的dataset，以及分配的kernel数不同等等变量的对照实验，我们可以发现这个时间基本都落在0.002s到0.0025s内，因此我们可以取定C\_init\_time = 0.00225s作为我们的实验用常数

在CPU-GPU架构上：

对于C\_copy\_m\_to\_d与C\_copy\_d\_to\_m

这两个变量其实我们可以统一成同一个量，因为决定设备之间拷贝速度的最大因素是设备之间的总线传输速率，而这个速率并不因数据的流向而改变。

对这部分的测量，我们采用对于数据拷出这个部分的时间来作为标准。在已经实现的代码中，拷出的数据仅为一个完成聚合的最底层cuboid，而如前文所述，每个cell的结构是相同的，以至于他们的大小也是相同的，于是拷出的数据量完全由拷出的cell个数决定。

我们通过修改数据集中每个维度数据的可能的取值的数目，来控制最初生成的cuboid中cell的个数（因为这个cuboid中对于每个维度的每个取值的所有组合，都有且仅有一个cell与之对应，即此时cell数=维度值的总乘积）

cell的大小：136 bytes

测试时为了简便都使用的三维聚合

单个kernel中分配的内存，除了cell的空间以外，还有一些对于kernel本身信息的标记，所以看到的大小跟cell个数 \* cell大小相差一个常数

在纯CPU架构上：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| X | Y | Z | cell总数 | 单kernel | Kernel数 | 总拷贝量 | 时间(平均) |
| 31 | 79 | 11 | 26939 | 3663708 | 16 | 58619328 | 0.02738936 |
| 7 | 11 | 13 | 1001 | 136140 | 16 | 2178240 | 0.00081016 |

由此可以作差估算出C\_copy\_d\_to\_m与C\_copy\_m\_to\_d为：4.70919E-10s / byte

在CPU-GPU架构上

对于t\_scan\_per\_cell

这个系数将在5.4.中被计算

5.2.

在本小节中，本文提到的数个算法的运行性能将在两种并行平台上得到评估。

5.2.1.

对于从源数据生成最底层cuboid算法的性能评估：

为了检验在不同的数据集上并行与否对性能的影响程度，在实验中我们将采用数据条数为262144，2097152，16777216，134217728这四个规模的数据集来进行。根据前文的算法设计，数据聚合立方体中cell的个数并不会影响到算法的时间复杂度，因此我们在每个测试之中都将数据维度值控制在（19, 101, 13）（即cell总数为24947）。

在纯CPU架构上：

对于空间最优算法的评估：

对于时间最优算法的评估：为了检验不同的kernel数对于实际运行速度的影响，我们在134217728条目的数据集上分别进行了不同的kernel数（指生成的并行kernel数，并不是指）运行的测试。这里所有的运行时间都是五次运行的平均。结果如下：

|  |  |
| --- | --- |
| Kernel数 | 运行时间（平均）（单位：s） |
| 1 | 2.810336 |
| 2 | 1.798154 |
| 4 | 1.701672 |
| 8 | 1.718508 |
| 16 | 1.62508 |

另外，对于不同规模的数据集的测试有结果如下（基于kernel数=16）

|  |  |
| --- | --- |
| 数据集条目数 | 运行时间（平均）（单位：s） |
| 262144 | 0.0260788 |
| 2097152 | 0.06733256 |
| 16777216 | 0.3991288 |
| 134217728 | 1.62508 |

在不同的kernel运行时，从系统管理中能够查询到的内存空间占用如下表：

|  |  |  |
| --- | --- | --- |
| 数据集条目数 | Kernel数 | 总内存占用 |
| 262144 | 1 | 18.5MB |
| 262144 | 2 | 25.0MB |
| 262144 | 4 | 37.9MB |
| 262144 | 8 | 63.8MB |
| 262144 | 16 | 115.6MB |
| 2097152 | 16 | 171.6MB |
| 16777216 | 16 | 619.6MB |

由上面这些测试数据和图表，可以得到如下结论：

1、由于该实验平台的CPU是2 core的，因此在多核实验中，生成的计算核心在超过2之后运行速度并没有实质性的变化（因为系统同时只能执行两个计算核心）。

2、另一方面，通过数据集大小和生成底层cuboid的时间的图表，可以看到这个算法运行的时间确实和原始数据集大小基本呈线性关系，与最初的算法分析一致。

在CPU-GPU架构上：

5.2.2.

对于由cuboid到cuboid的聚合算法的评估

这一小节中将进行有无采用并行方式对于聚合性能的评估

在纯CPU架构上：

在本部分的测试中，我们将固定在cuboid聚合中使用聚合路线选择算法。采用的原始cuboid是基于262144个元素的数据集生成的cell总数为24947(13, 101, 19)的cuboid，目标cuboid是三步聚合后的顶层cuboid。测试结果如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Kernel数 | Step1 | Step2 | Step3 | Total |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 4 |  |  |  |  |
| 8 |  |  |  |  |
| 16 |  |  |  |  |

在CPU-GPU架构上：

5.2.3.

对于聚合路线选择算法的评估

在本小节中，我们将测试不同的聚合路线选择对于从一个较原始的cuboid到另一个上游cuboid的聚合性能的影响。

在纯CPU架构上：

下文将列出参加测试的原始cuboid的各项参数，以及实际程序运行的结果。为了控制变量的原则，这里固定OpenCL生成的计算kernel数为16。这些原始cuboid都是在5.2.1.的测试中生成的。为了简单起见，我们假定所有的聚合的最终目标都是一个“全维度聚合”cuboid，即一个顶层cuboid，从它将无法再进行任何聚合。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 原数据集条目数 | cuboid的cell个数 | 路径优化 | 聚合步骤所需时间（第一次） | 聚合步骤所需时间（第二次） | 聚合步骤所需时间（第三次） |
| 134217728 | 24947 | T | 0.000851837 | 0.000028404 | 0.000006149 |
| 134217728 | 24947 | F | 0.000828847 | 0.000103967 | 0.000018604 |
| 262144 | 24947 | T | 0.000859935 | 0.000026190 | 0.000006404 |
| 262144 | 24947 | F | 0.000833913 | 0.000168310 | 0.000017815 |
| 262144 | 3333 | T | 0.000277024 | 0.000007584 | 0.000005732 |
| 262144 | 3333 | F | 0.000260116 | 0.000046094 | 0.000021583 |

根据测试所得数据，可以做出柱形图来表现出几种不同的测试数据集之间的性能差异。

可以从数据汇总表格和柱形图中看到如下规律：

1、由于这个算法是从cuboid出发的计算算法，因此与原始数据集的大小无关

2、如同分析的那样，我们的路径优化算法在第一步聚合时是不会起到任何作用的，因此第一步的执行时间基本相同，可能由于系统架构或者是代码书写的问题，使用路径优化的方案甚至略慢于不使用路径优化的方案。而从第二步开始，经过了聚合路径优化的算法开始凌驾于未经过优化的算法。但是由于第一步聚合占用的时间占了整个聚合步骤的最主要部分，因此总体时间上来看优化的比例大概在10%-15%之间。

在CPU-GPU架构上：

5.2.4.

根据前文测试所得结果，下面我们分别在不同的平台上计算出一个花费函数，并且用这个花费函数与Stanford的方案下的花费函数来计算在不同情形下的预处理集合，以及评估他们之间的性能差距。

这个测试中，由于本身维度空间比较小（3），所以可能的cuboid总数也并不是太多（8），因此在我们的测试中固定预处理的cuboid数量为2。评估最后的性能时，我们将以“除去底层cuboid以外的所有cuboid（7）生成的总时间”作为评估标准。

在纯CPU架构上：

根据5.1.中估计的结果，我们已经有了花费函数估计式中的三个系数如下：

C\_copy\_hd\_to\_m = 9.34613E-8 s / byte

C\_init\_time = 0.00225s

C\_copy\_m\_to\_d = C\_copy\_d\_to\_m = 4.70919E-10s / byte

考察t\_scan\_per\_cell，其指的是在从cuboid到cuboid聚合时“单个kernel”对于“聚合前的单个cell”的处理时间。因此，这个地方要进行计算时，首先要选择的数据集应该也是“单kernel前提下”进行聚合的时间测量结果。因此我们将采用5.2.2.中的测量数据进行计算。

5.3.1.

在本小节中，我们将考察何时必须采用3.4.中提出的方案。

显然，若是条件允许，3.3.的方案易于内存管理，易于程序流程控制并且拥有最高的运行速度，应该是我们的首选。而也如前所述，3.3.中的最大的问题在于潜在的占用内存空间过多问题。因此考察3.4.中提出的方案何时必须被使用，其实另一方面来说也是考察3.3.中方案使用的边界条件。

在讨论内存空间问题的大前提下，我们可以发现，在整个算法运行之中，占用内存最多的主要是两块数据：直接读入的原始数据集，以及在运行中产生的临时subcuboid。下面就从这两个方面来分析其对于方案选择的影响。

在5.2.1.的实验数据中，可以计算得出，在本实验所给定的cell的结构基础上，一个拥有24947个cell的cuboid所占用的空间约为6.5MB。因此仅从设备空间因素考虑，假定我们有2GB内存空间能够分配给这些临时subcuboid的前提下，能够同时运行的最大kernel数和一个cuboid的总cell数的关系大概如下：

|  |  |  |
| --- | --- | --- |
| 一个cuboid的总cell数（维度值相关） | 最大可并行kernel数 |  |
| 1000 | 7692 | GPU |
| 5000 | 1538 | GPU |
| 10000 | 769 | GPU |
| 25000 | 307 | GPU |
| 50000 | 153 | GPU |
| 100000 | 76 | GPU？ |
| 250000 | 30 | ？ |
| 1000000 | 15 | CPU |
| 2500000 | 6 | CPU |
| 5000000 | 3 | CPU？ |
| 7500000～ | 1 | x |

基本可以得到如下结论：在维度总乘积小于100000的时候，基本上可以不做任何调整将其部署在GPU上进行并行计算，正如本论文中实验所涉及到的数据集那样。在100000～2500000的区间，如果为了保证在GPU上的并行度，则应该转而采取混合方案来进行计算；而在CPU上仍可以采用直接计算的方法而不影响到并行度。在更大的区间上，则由于存储空间已经不足以支持数个，甚至是一个完整的cuboid，则无论CPU抑或是GPU都应该采取混合方案来进行计算。

然后，我们再来考察一下数据集大小对于是否采用混合方案的影响。然而实际上，由于本来就存在“原始数据集过大以至于无法一次全部读取”的问题，所以在内存的分配上，我们可以主要以cuboid占用的总空间作为衡量标准，然后对原dataset进行一定程度的划分，使得每个part的内存占用与临时subcuboid占用的空间总和不超过允许的最大限度。在此基础上，每次都在其中的一个part上运行我们的算法（然后我们的算法在运行过程中把这个part再分成一些更小的part），并将这个part的聚合结果暂时存储在一些其他存储介质中，最后再收集起来形成一个完整的数据集的底层cuboid。

综合以上两部分的分析，我们可以得知，采取何种方案其实直接相关于问题的规模，即我们需要得到的cuboid具体的cell总数的多少。然后在这个基础上，我们可以适当的在编程上对一次读入的数据集的大小进行调整，使得程序能尽可能更快地运行。

5.3.2.

在本小节中，我们主要评估在不同的实验平台，也就是在不同的系数修正下得到的我们的估计花费函数和Stanford方案的差异点。

可以发现，实际上在硬盘性能不足，抑或是并行执行的算法计算量本身并非太大的话，对应项的系数修正之后，我们的方案的花费函数的最终表达式基本上和Stanford的式子是统一的：修正系数能让我们的花费函数的读写时间项的影响和别的项有超过1个数量级的差距，甚至更大。

而当我们在更宽的主板数据总线／更快的硬盘，抑或是运行更复杂的，计算量更大的并行算法的时候，系数的修正有时就会使得修正后的花费函数开始偏向于花费函数中的计算项。此时两种花费函数所选择出来的预处理cuboid将会有所区别。而因为我们的花费函数直接以时间作为考量因素，因此在以时间作为衡量标准的最终运行效率上自然就比以空间作为运行效率的间接考量因素的Stanford方案要更优

但无论怎样，我们的方案指出了一个事实：在不同的硬件平台，不同的算法构成的前提下，花费函数都是需要重新计算的。

6.

本文提供了基于OpenCL的利用并行计算平台来对OLAP中数据聚合立方体的生成／计算进行优化的基本动机，设计方案，具体实现和评估结果。事实证明，采用了并行计算的方案在各方面都强于非并行方案，并且基于并行计算平台的特点而进行的一些优化也体现出了一定的效果。

未来该项研究的扩展方向将会往数个方向进行：在并行计算cuboid的过程中，加入online aggregation技术；重新审视花费函数，寻找更为简单易懂的，但是性能仍然能保持的方案；以及将以上这些技术整合成为一个更为完整的数据库系统。

7.参考文献

[1] The Manycore Shift: Microsoft Parallel Computing Initiative Ushers Computing into the Next Era. Nov 2007

[2] http://www.nvidia.com/object/cuda\_home\_new.html

[3] https://en.wikipedia.org/wiki/CUDA

[4] https://en.wikipedia.org/wiki/OpenCL

[5] B. He, K. Yang, R. Fang, M. Liu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In SIGMOD, pages 511–524, 2008

[6] B. He, M. Liu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander. Relational query coprocessing on graphics processors. ACM Transactions on Database Systems, 34(4), December 2009.

[7] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. Proc. VLDB Endow., 4(5):314–325, 2011.

[8] K.Wang, K.Zhang, Y.Yuan, S.Ma, R.Lee, X.N.Ding, X.Zhang. Concurrent Analytical Query Processing with GPUs. Proc. VLDB Endow., 7(11):1011-1022, 2014

[9] P.Johns, J.He, B.He. GPL: A GPU-based Pipelined Query Processing Engine. In SIGMOD, 2016

[10] https://en.wikipedia.org/wiki/Online\_analytical\_processing

[11] S.Chaudhuri, U.Dayal. An Overview of Data Warehousing and OLAP Technology. In SIGMOD, 1997

[12] V.Harinarayan, A.Rajaraman, J.D.Ullman. Implementing Data Cube Efficiently. In SIGMOD 1996

[13] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In SIGMOD, 2009.

[14] https://www.sap.com/product/technology-platform/hana.html

[15] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. Proc. VLDB Endow., 6(10):817– 828, Aug. 2013.

[16] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled CPU-GPU architectures. Proc. VLDB Endow., 8(4):329–340, Dec. 2014.

[17] T.Karnagel, D.Habich, W.Lehner. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. Proceedings of the VLDB Endowment, Vol. 10, No. 7, 2017