# 面向大规模机群的可扩展 OLAP 查询技术

王会举[1),2)] 覃雄派[1),2)] 王 珊[1),2)] 张延松[3)] 李芙蓉[1),2)]

[1)](数据工程与知识工程教育部重点实验室(中国人民大学) 北京 100872)

[2)](中国人民大学信息学院 北京 100872)

[3)](中国人民大学中国调查与数据中心 北京 100872)

**摘 要** 大数据时代,由中低端硬件组成的大规模机群逐渐成为海量数据处理的主流平台之一.然而传统基于高端硬件平台设计的并行 OLAP 查询算法并不适应这种由不可靠计算单元组成的大规模并行计算的环境.为改善其在新计算环境下的的扩展性和容错性,该文对传统数据仓库的数据组织模式及处理模式进行改造,提出了全新的无连接雪花模型和 TRM 执行模型.无连接雪花模型基于层次编码技术,将维表层次等关键信息压缩进事实表,使得事实表可以独立处理数据,从数据模型层保证了数据计算的独立性;TRM 执行模型将 OLAP 查询的处理抽象为 Transform、Reduce、Merge 3 个操作,使得 OLAP 查询可被划分为众多可并行执行的独立子任务,从执行层保证了系统的高度可扩展特性.在性能优化方面,该文提出了 Scan-index 扫描和跳跃式扫描算法,以尽可能地减少 I/O 访问操作;设计了并行谓词判断、批量谓词判断等优化算法,以加速本地计算速度.实验表明:LaScOLAP 原型可以获得较好的扩展性和容错性,其性能比 HadoopDB 高出一个数量级.

**关键词** 大规模可扩展;OLAP 查询;无连接雪花模型;TRM 执行模型;跳跃式扫描;Scan-index;大数据

**中图法分类号** TP311 **DOI 号** 10.3724/SP.J.1016.2015.00045

## Scalable OLAP Queries Processing Towards Large Cluster

WANG Hui-Ju[1),2)] QIN Xiong-Pai[1),2)] WANG Shan[1),2)] ZHANG Yan-Song[3)] LI Fu-Rong[1),2)]

[1)](*Key Laboratory of Data Engineering and Knowledge Engineering* (*Renmin University of China*), *MOE*, *Beijing* 100872)

[2)](*School of Information*, *Renmin University of China*, *Beijing* 100872)

[3)](*National Survey Research Center at Renmin University of China*, *Beijing* 100872)

**Abstract** In big data era, a large computer cluster composed of low-end servers has become one of the most popular platforms for massive data analysis. While traditional OLAP query processing algorithms that are implemented on high-end servers don't adapt to the large and less reliable parallel computation environment. To improve its scalability and fault tolerance, we modified the conventional data schema and processing style of data warehouse, and proposed join-free snowflake (JFSS) and TRM execution model. Based on hierarchy encoding, JFSS schema compresses dimension hierarchies' information into fact table, which gives a fact table the ability of processing data independently without dimension tables, and ensures the computation independence of data from schema level. TRM execution model abstracts all data warehouse processing into three operations: Transform, Reduce and Merge (TRM execution model), and divides an OLAP query into massive independent tasks, which ensures the large scalability and fault tolerance of the system from execution level. To optimize the performance, we proposed Scan-index algorithm

收稿日期:2011-06-20;最终修改稿收到日期:2014-09-24.本课题得到国家"九七三"重点基础研究发展规划项目基金(2014CB340403)、国家重大科技专项基金(核高基项目 2010ZX01042-001-002)、国家自然科学基金(61170013、61272138)、中国人民大学科学研究基金(中央高校基本科研业务费专项资金(10XNI018))资助.**王会举**,男,1979 年生,博士,主要研究方向为大规模集群数据库、内存数据库.E-mail:wanghuiju@ruc.edu.cn.**覃雄派**,男,1971 年生,博士,讲师,中国计算机学会(CCF)会员,主要研究方向为数据库查询优化、内存数据库、并行数据库.**王 珊**,女,1944 年生,教授,博士生导师,中国计算机学会(CCF)会士,主要研究领域为高性能数据库、知识工程、数据仓库.**张延松**,男,1973 年生,博士,副教授,主要研究方向为高性能数据库、内存数据库.**李芙蓉**,女,1989 年生,硕士,研究方向为云计算.

and jump scan algorithm to reduce I/Os as much as possible, and designed batch evaluation of predicates algorithm, parallel evaluation of predicates algorithm etc., to accelerate the processing speed of each data node. Experimental results show that, the prototype system, LaScOLAP yields high scalability and fault tolerance, and its performance is an order of magnitude higher than HadoopDB.

**Keywords** large scalability; OLAP query; join-free snowflake schema; TRM model; jump scan; Scan-index; big data

## 1 引 言

数据仓库规模的爆炸式增长①,迫使越来越多的企业将应用从高端服务器移植到了由中低端硬件构成的大规模计算机机群.

然而,传统的并行数据库难以适应此新的硬件环境.大多数并行数据库是为有限规模高端服务器组成的机群而设计[1].在此种环境下,节点失效属稀有事件,复杂查询大都可以在数小时内完成,其处理失败查询的基本措施是重做整个查询.然而,在将其部署于由中低端硬件构成的大规模机群计算环境时,节点失效概率将迅速增加,一个节点的失效可能导致整个查询的重新执行.极端情况下,并行数据库将可能出现不停重做查询的局面.

传统数据仓库往往按照星型模型或雪花模型对数据进行组织,并依赖于连接操作来处理查询,当连接的数据不能保证在并行处理节点上的局部性时,OLAP 查询处理算法同样难以适应大规模机群的计算模式:(1)如果将事实表和维表按照传统的并行数据库方式均匀分布于各个数据节点,连接操作将引入大量数据迁移,降低了 OLAP 的整体性能;(2)如果采用将维表在各个数据节点复制、事实表水平均匀分割的方式,又会引入较高的存储空间和维表更新维护代价.以 10 GB 的维表和 1 TB 的事实表为例②,维表虽然只占事实表的 1%,但若在 100 个节点复制维表,维表所占空间将变为 $100\times10$ GB$\approx$1 TB——几乎等同于事实表的容量.节点数越多,维表数据所占空间比例将越大.

为了获得高度可扩展的 OLAP 查询处理模式,本文对数据仓库的传统数据组织模式进行了改造,提出了基于扫描的无连接雪花模型.基于新的数据模型,我们设计了高度可扩展的、完全不同于传统查询处理方式的 TRM 执行模型.

无连接雪花模型 JFSS(Join-Free Snowflake Schema)采用局部层次编码技术,将维表层次信息压缩进事实表,使得事实表可以独立执行维表上的谓词判断、聚集等操作,从而使连接的数据在大规模机群上实现局部性,消除了连接操作.

TRM(Transform-Reduce-Merge)执行模型,将数据仓库的操作抽象为 Transform、Reduce、Merge 3 个操作:(1) Transform.主节点对查询进行预处理,将查询中作用于维表的操作(如谓词判断,group-by 聚集操作等)转换为事实表上的操作;(2) Reduce.每个数据节点并行地扫描、聚集本地数据,然后将处理结果返回给主节点;(3) Merge.主节点对各个数据节点返回的结果进行合并,并基于合并数据执行后续的过滤、排序等操作.TRM 执行模型可将 OLAP 查询分解为众多可独立执行和恢复的子任务,并将网络传输量限制于聚集结果的合并上,从而实现了高效可靠的执行模式.

为了加快数据节点的处理速度,我们采取了如下优化算法及策略:

(1) Scan-index 算法用动态索引的方式有效地避免了多列查询时的列间连接操作,有效减少了列存储模型的元组重构代价;

(2) 基于压缩数据设计的批量谓词判断可以使多个具有相同属性值的元组只执行一次谓词判断,从而加快谓词判断的速度;

(3) 跳跃式扫描算法可以尽可能地跳过与查询不相关的数据块,有效减少 I/O 访问操作.

最后要强调的是,本研究所依赖的关键技术——层次编码技术,在某些领域已有应用,如 XML 数据管理等.我们的最大贡献是结合 OLAP 查询的特点,对其进行优化改造(提出局部层次编码方式),将其用于解决当前 OLAP 领域面临的最具

① WinterCorp: 2005 TopTen Program Summary. http://www.wintercorp.com/WhitePapers/WC_TopTenWP.pdf

② 在 SSB 标准测试集中,维表所占事实表比例基本都在 1%左右,一般不会超过 3%,数据规模越大,维表所占比例越低.

挑战性的问题之一——大数据的处理分析，从而实现了高性能的、可扩展的 OLAP 查询处理方式.

基于 SSB 标准基准测试的实验表明，原型系统 LaScOLAP(Large Scale OLAP)具备高度的可扩展性和容错性，同时其性能也远超面向大规模机群的混合式数据库 HadoopDB[1].

本文第 2 节列出相关工作；第 3 节论述无连接雪花模型；第 4 节讨论无连接雪花模型在机群环境下的存储策略；第 5 节讨论 TRM 执行模型及各种优化算法的实现；第 6 节进行实验分析.

## 2　相关工作

MapReduce[2]将数据的访问操作抽象为一系列 Map 操作和 Reduce 操作，把一个任务划分为多个可以独立执行扫描操作的子任务，子任务间通过物化的〈*key*，*value*〉形式连接起来，从而获得较好的扩展性和容错性. 本文的一些思想受启发于 MapReduce 技术，通过查询转换(Transform)，将所有查询抽象为 Reduce 和 Merge 操作，把查询任务分为众多独立扫描的子任务，在机群上并行执行，从而获得可扩展的 OLAP 查询执行方式. 但 LaScOLAP 是对 OLAP 查询的高度抽象，而 MapReduce 是对通用数据处理操作的抽象. 此外，LaScOLAP 没有采用 MapReduce 步步物化的方式，而是把扫描和聚集操作用流水线方式执行，因此对数据仓库查询执行效率更高.

文献[3-4]提出用维度的层次编码代替星型(雪花)模型中的主外键，并基于层次编码对事实表进行聚簇存储，从而将星型(雪花)连接转换为基于 UB-Tree[5]的空间范围查询. 本文也采用了层次编码技术，但与之前研究相比，本文侧重于研究利用层次编码对星型(雪花)模型进行改造，以设计扩展性更好的数据模型. 同时查询的处理也与之前研究有所不同，本文提出的是一种通用的查询处理框架，不仅仅是针对 ad-hoc 查询. 此外，从编码规则上来看，文献[4-5]采用的是基于层次全局域的方式——层次中所有成员被统一编码，本文中编码方式是基于层次局部域的方式——对属于同一个父类的成员统一编码. 从空间上来看，本文的编码方式更加节省空间.

文中提出的无连接雪花模型借鉴了泛关系模式(Universal Relation)[6]的思想，但与其不同的是，我们的模型并没有将维表的所有属性都放入事实表，而只是将维表的层次信息通过编码的方式压缩进事实表，从而更加节省空间.

星型(雪花)模型：传统的星型模型或者雪花模型都是依赖于连接操作来处理查询，本文中的无连接雪花模型是基于扫描操作来处理查询.

IBM 的 Blink[7]系统将整个星型(雪花)模型预连接为一张大表，然后将连接后的大表根据属性值出现频率划分为多个单元，再将每个单元压缩编码为二进制形式. 在执行查询时从每个单元的头部开始解压并处理每个元组. 不同于 Blink 系统，LaScOLAP 只是将关键的维度层次信息压缩进事实表，而非全部. 两者压缩的技术也不同，Blink 采用基于字典的压缩技术，而 LaScOLAP 采用基于层次编码的压缩技术. 同时 LaScOLAP 可以实现多种表扫描的方式，比如跳跃式表扫描等，而 Blink 系统只能提供一种全表扫描的访问方法. 但 LaScOLAP 借鉴了 Blink 系统中通过物化手段来简化复杂星型连接操作的思想和并行谓词判断算法.

## 3　JFSS 逻辑模型

### 3.1　基本概念

OLAP 数据主要分为度量和维度两类. 度量(或事实)，主要是数值型，随时间变化，对应某特定时间点跟某事物相关的值(比如银行某天的存款额，某地区的特定年份的人口数等)；维度被用来描述度量，通常，维度几乎是静态的，较少随时间变化的(如时间、地区、产品等). 维度数据通常较小，事实数据占据绝大部分空间. 事实表数据往往是以批量方式追加进数据仓库的.

在 ROLAP 中，每个维度通常被存储在一张关系表中(称为维表)，表中属性往往代表不同的层次或者某些描述信息，比如时间维度，可以包含年、月、日等不同层次，同时也包含某些描述信息，比如是否节假日等. 不同层次往往存在包含和被包含关系，如年包含月，月包含日. 某些情况下，一个层次也可以被多个层次包含，比如层次日可以被层次月包含，也可以被层次星期包含.

度量被存储在事实表中，事实表通过描述度量的维属性同维表建立参照完整性约束关系.

### 3.2　JFSS 无连接雪花模型

正如前言部分提到的，星型(雪花)模型因为复杂连接操作的存在而不适合大规模机群操作. 泛关系模型将模型中所有属性集中于单个关系，虽可避免连接操作但却付出了昂贵的存储空间代价.

数据仓库查询涉及的列一般为维表中的层次属性和事实表度量属性(非层次属性上的操作可以转换为层次属性上的操作,我们将在 5.1.3 节讨论),基于数据立方体上的查询即是例证.基于此,JFSS(Join-Free Snowflake Schema)模型采取了如下策略:借鉴泛关系模式(universal relation)的思想,将维表信息物化到事实表.但与泛关系模型不同的是,JFSS 模型没有将所有属性都存入事实表,而是将维表层次信息通过层次编码压缩进事实表.JFSS 模式可被看作"准泛关系"模型.JFSS 模型的详细定义如下.

**定义 1**(层次局部域). 设 $L$ 是维度 $D$ 的一个层次,其所有成员的集合为层次 $L$ 的全局域.设层次 $L$ 的父类层次为 $PL$(最高层次的父类记为 $ALL$),$L$ 的一个局部域是指同属一个父类(记为 $p$)的 $L$ 层成员集合,记为 $localDomain_{PL=p}(L)$.如图 1 所示,$localDomain_{PL='China'}(City)=\{Beijing, Hong\ Kong, Shanghai\}$.层次全局域是多个层次局部域的合集.

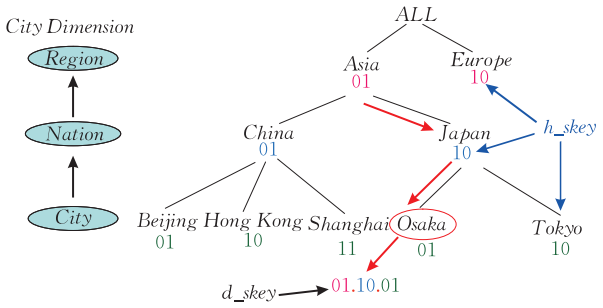


图 1 维度层次树及编码示例

**定义 2**(局部层次码). 设某一层次的局部域成员基数为 $m$.我们为此局部域定义一个一对一的映射函数 $S: localDomain_{PL=p}(L) \rightarrow [0, m]$,以使对于每一个 $u$,$u' \in localDomain_{PL=p}(L)$,且 $u<u'$,都有 $S(u)<S(u')$,$S(u)$ 称为 $u$ 的局部层次码,简记为 $h\_skey_{D,L}(u)$.本研究中,用二进制串来表示一个局部层次码,对一个成员基数为 $m$ 的层次局部域来说,共需要 $[\log_2 m]$ 个二进制位.如图 1 所示,*China* 节点有 3 个孩子节点 *Beijing*,*Hong Kong* 和 *Shanghai*,我们可以对其依次编码为 01,10 和 11.

**定义 3**(全局层次码). 每一个维度都对应一棵层次树.该层次树上每一个节点都存在一个按照先序遍历从根(最高层节点)到其自身的路径.我们把路径上每个节点的局部层次码连接后形成的码称为该节点的全局层次码,维度 $D$ 的 $L$ 层成员 $M$ 的全局层次码记为 $d\_skey_{D,L}(M)$.如图 1 所示,*Osaka* 的局部层次码 01,其在 *City* 维度中的全局层次码为 01.10.01.

若某层次成员对应多个父类节点(比如中国按地域讲属于亚洲,但从经济发展水平讲又属于发展中国家),到达该节点的路径就对应多条.对其编码时,我们先将到达该节点的所有路径抽取出来,形成一棵子树,然后按照广度优先顺序将路径上每个节点的局部层次码进行连接,即形成该节点的全局层次码.广度优先遍历可以保证较高的层次在编码串中处于较前的位置,从而获得较好的聚簇效果(详见 4.2.1 节).特别需要指出的是,本文中的层次含义更为广泛——维表中的属性都可以被看做层次.比如如果将日期维度(*Year*, *month*, *isHolidayFlag*, *dateKey*)中的 *isHolidayFlag* 属性看做一个层次,则其对应层次树的层次为 *dateKey*→*isHolidayFlag*.

**定义 4**(复合层次码). 以层次为粒度、按照层次由高到低的顺序对多个维度的全局层次码交错组合后的编码称为复合层次码(记为 *md_skey*,下文中简记为复合码).

本文根据经验设定的默认编码规则为:时间维度为第 1 优先维度,其他维度的优先级根据其成员数确定,成员数越少优先级越高.对如图 2 所示维度 *Time*、*City*、*Product* 的成员数分别为 500、200、50,3 个维度的优先级由高到低为 *Time*→*Product*→*City*,其对应的编码规则为 *Year*.*Product_Type*.*Region*.*YearMonth*.*Product*.*Nation*.*Date*.*City*,如图 2 中虚线箭头走向所示.

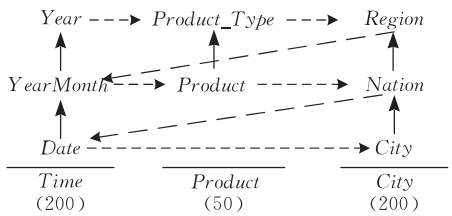


图 2 复合层次码编码示例

JFSS 模型同样包括维表和事实表两种表.每一层次对应一维表,由如下列组成:

(1)全局层次码(*d_skey*).维度层次树上由最高点到该节点的路径上的编码组合;

(2)原维表中函数依赖于该层次的其他属性.

事实表包含的列信息如下:

(1)该事实表参照的各个维度组合而成的复合层次码 *md_skey*;

(2)原事实表中其他属性,主要是度量信息.

事实表中采用复合层次码可以:(1)减少表示

二进制字段时的二进制位数的浪费(1 个二进制位也需要用一个字节存储),从而更加节省空间;(2)进行更加有效的谓词判断算法,如批量谓词判断、并行谓词判断等;(3)提供更加有效的 I/O 访问(排序后,数据自然就形成聚簇. 详见 4.2.1 节). 出于对存储空间考虑,事实表中的原外键字段可以删除.

图 3 举例对比了星型模型和无连接雪花模型之间的对应关系. 从中可以看出,后者事实表已经包含了所有维度的层次信息,从而使基于事实表扫描的处理方式成为可能.

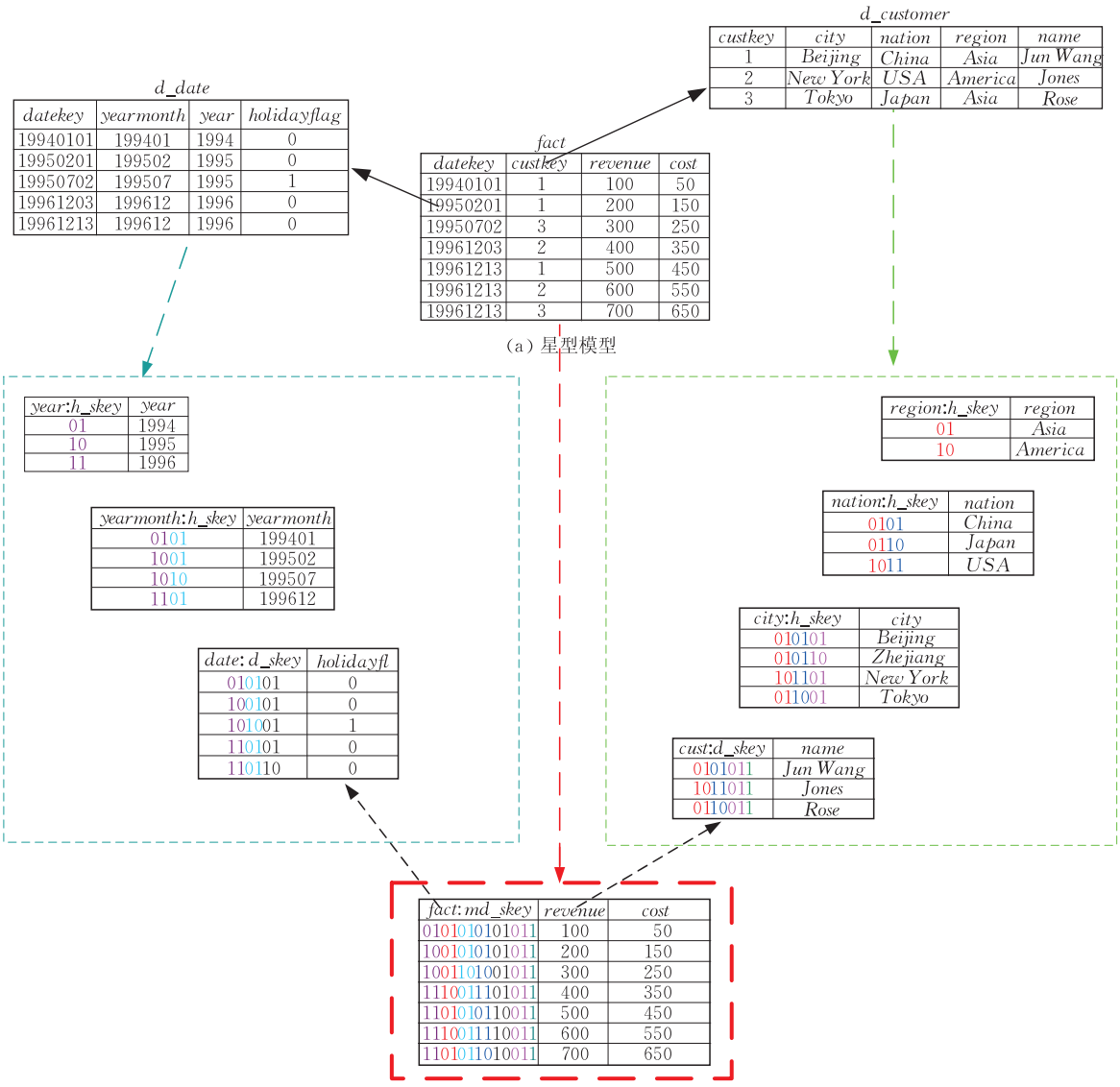


图 3　无连接雪花模型同经典星型模型的对比

此外,维表虽然较少变化,但一旦变化,即可能影响到事实表数据. 因此,维表更新后的高效数据同步算法是我们未来的重点研究内容之一.

## 4　物理存储模型

通常情况下,维表所占存储空间较少(以 Star Schema Benchmark(SSB)标准测试集 30 GB 数据为例,维表只占 1.1%). 在 JFSS 模型中,事实表已含维表主要信息,可以独立处理数据. 基于此,我们采用了主-从架构来存储事实表和维表. 维表集中存储于主节点,事实表均匀分布于各个从节点(文中也称为数据节点).

### 4.1　维表存储策略

在设计维表的存储策略时,我们主要考虑了如下两点:(1)维表数据量通常较小(一般至多属于 GB 级),且访问频度高(几乎所有查询均会涉及维表访问);(2)当前服务器内存至少都是 GB 级,某些

已达 TB 级.因此,维表基本可以全部放入主节点的内存中,从而避免 I/O 操作.

少数内存不足以存放所有维表的情况下,此时可借助于外存,同时可利用索引来提高维表访问速度.

### 4.2 事实表存储策略

事实表的访问性能决定了整个查询的性能,因此针对事实表存储我们采取了如下一系列优化措施.

#### 4.2.1 分布式存储策略

在预处理阶段我们根据事实表的复合码对事实表进行排序,排序后记录自然形成的聚簇,而且层次越高聚簇效果越明显.比如复合码按照年.省份.月.地区.日期的编码规则排序后,同年份的相关查询会得到最好的聚簇效果,其后层次聚簇效果依次减弱,最底层的日期的聚簇效果最差.

如在 3.1 节描述,事实表数据往往是随时间变化、以批量方式追加进系统的.因此,在 LaScOLAP 的实现中,我们将时间相关维度作为第 1 优先维度,以实现增量式的排序效果:新加入的数据经排序后自然排在所有数据的后面.因此我们每次只需对新加入的数据单独排序,然后追加进系统即可,从而大大降低排序的代价.对于极少数情况下,时间顺序和数据加载顺序不完全一致时,往往需要在加载后执行一次全局排序操作.

基于排序聚簇后的事实表,我们将其划分为多个 *chunk*,每个 *chunk* 包含若干块,然后以 round-robin 方式,以 *chunk* 为单位在节点间循环布局,如图 4 所示.数据的备份也以 *chunk* 为单位采用类似 MapReduce 的备份策略.

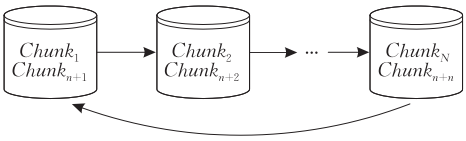


图 4 事实表分布式存储示例

#### 4.2.2 物理块存储优化

〈*key*,*value*〉存储.绝大多数星型(雪花)查询中出现的谓词操作都是基于维度层次的,聚集操作也大多如此(如 MOLAP 中的多维查询).在分析型领域,列存储往往优于行存储.基于以上分析,LaScOLAP 采用了类似〈*key*,*value*〉形式的列式存储模型:*key* 为复合码 *md_skey*,*value* 为事实表中的除复合码外的某一属性值.事实表中 *value* 大多为定长类型(数值型),出于空间考虑,我们只在每一个块的头部存储 *key* 的二进制位数、*value* 类型及长度等,从而省下大量的记录指针等信息的存储空间.同时,数据类型信息等存储在块内,可以为数据的存储分布带来极大的灵活性.对于占少数的不定长数据类型,仍采用传统的 Page-slot 存储方式.

前缀压缩存储.压缩可以有效改善 I/O 性能,节省磁盘空间.事实表按照 *key* 值排序后,相邻元组的 *key* 值往往含有较多的相同二进制位,意味着我们可采用基于块的前缀压缩算法:将每一块的第 1 条记录存储完整的 *key* 值,紧邻的后续元组只存储同前一条元组不同的位串(示例见图 5).


图 5 相邻元组压缩后存储示例($v_1$,$v_2$ 为度量值)

## 5 TRM 执行模型

LaScOLAP 按照统一的执行模型 TRM(Transform-Reduce-Merge)处理所有的查询.其框架如图 6 所示.当一个 SQL 查询到达时,主节点上的 Transform 模块将所有维度上的操作都转换为事实表复合码上的操作.然后将转换后的查询下发至各个数据节点,各个数据节点独立并行地执行本地数据的 Reduce 操作——扫描、聚集、排序,并将聚集排序后的数据返回到主节点,最后由主节点进行数

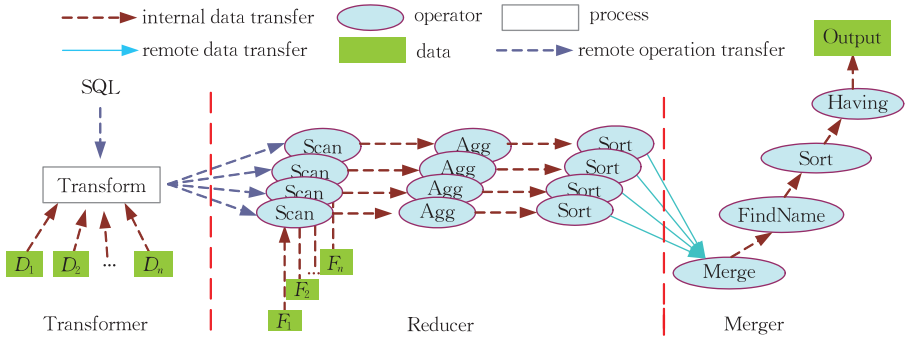


图 6 TRM 执行模型

据的合并、代码翻译及排序等后续操作. 数据仓库的聚集查询往往具有高输入低输出的特点(以 SSB 基准测试中查询为例，在 30 GB 的数据量下，所有查询的输出记录都在 1000 条以下，所占空间不超过 100 KB)，因此网络传输量也被降至最低.

### 5.1 Transform

析取范式可以表示为多个合取范式的并集，如下我们的讨论集中于合取范式.

#### 5.1.1 等值谓词判断转换

所有谓词项都是等式判断的谓词称为等值谓词. 我们将等值谓词判断转换为如下二进制操作：

$mask\_e \ \& \ t.md\_skey = constant\_e$

掩码 $mask\_e$ 用于抽取谓词中涉及到的层次. $constant\_e$ 是由谓词中涉及到的层次的期望值组成的二进制串.

以图 3 中 JFSS 模型为例. 等值谓词判断 $d.year = 1996$ and $c.region = 'Asia'$，其对应的 $mask\_e = '11.11.00.00.00.00.0'$，$constant\_e = '11.01.00.00.00.00.0'$. 其转换后的谓词判断为 $md\_skey \ \& \ '11.11.00.00.00.00.0' = '11.01.00.00.00.00.0'$.

#### 5.1.2 范围谓词判断转换

含有不等式谓词项的谓词称为范围谓词. 值得指出的是，范围谓词也可能含有等式谓词项. 因此，范围谓词被转换为一个含有两个谓词判断的新合取范式：一个是等值谓词项，用于判断该合取范式中的所有等值谓词；另一个是范围谓词项，用于判断此合取范式中的所有不等式谓词项. 在此，我们详细讨论后者.

一个合取范式中的所有不等式谓词项被转换为两个二进制串，$constant\_max$ 和 $constant\_min$. $constant\_max$ 由谓词中涉及到的层次的上限值按照复合编码的编码规则组合而成，代表满足谓词判断的元组的最大值；对应地，$constant\_min$ 由谓词涉及到的层次的下限值组合而成，代表满足谓词判断的元组的最小值. 没有出现在合取范式中的层次相应位用全 0 代替. 同样也产生一个掩码 $mask\_r$ 来抽取该谓词判断中涉及到的层次. 对于每一个元组 $t$ 执行如下谓词判断：$(mask\_e \ \& \ t.md\_skey = constant\_e)$ and $((mask\_r \ \& \ t.md\_skey)$ between $constant\_min$ and $constant\_max)$. 仍以图 3 中 JFSS 模型为例. 范围谓词判断 $d.year \geq 1994$ and $d.year \leq 1996$ and $c.region = 'Asia'$，其对应的 $mask\_e$ 为 '00.11.00.00.00.00.0'，$constant\_e$ 为 '00.01.00.00.00.00.0'，$mask\_r$ 为 '11.00.00.00.00.00.0'，$constant\_min$ 为 '01.00.00.00.00.00.0'，$constant\_max$ 为 '11.00.00.00.00.00.0'. 转换后的谓词判断为 $md\_skey \ \& \ '00.11.00.00.00.00.0' = '00.01.00.00.00.00.0'$ and $md\_skey \ \& \ '11.00.00.00.00.00.0'$ between '01.00.00.00.00.00.0' and '11.00.00.00.00.00.0'.

#### 5.1.3 列表谓词判断转换

对于 Like、In 等列表谓词判断，其转换只是简单的代码替换，然后将替换后的代码下发至各个数据节点，由各个数据节点在扫描时进行谓词判断.

对于基于维表上其他描述信息的谓词判断我们也采取类似于 In 列表的转换手段，将符合条件的维度成员的全局层次码下发到各个数据节点.

#### 5.1.4 Group by 转换

Group by 语句的转换比较简单，只需要产生一个用于抽取 group by 语句中涉及到的层次的掩码即可(记为 $mask\_group$)，对每一个元组 $t$ 执行

group by $t.md\_skey \ \& \ mask\_group$.

#### 5.1.5 一个完整的转换例子

如下基于图 3 中 JFSS 模型的星型查询：

```
select d.year, c.nation, sum(revenue)
from fact, d_date, d_customer
where d.year≥1994 and d.year≤1996
      and c.region='Asia'
group by d.year, c_nation
```

转换后的 Reducer 可用 SQL 伪表达为：

```
select md_skey & '11.11.00.11.00.00.0',
  sum(revenue)
from fact
where (md_skey &'00.11.00.00.00.00.0'
  ='00.01.00.00.00.00.0') and
  (md_skey & '11.00.00.00.00.00.0'
  between '01.00.00.00.00.00.0'
  and '11.00.00.00.00.00.0')
group by md_skey & '11.11.00.11.00.00.0'
```

### 5.2 Reduce

转换后的查询被下发到各个数据节点. 每个数据节点按照如下流水线方式执行本地数据的处理：扫描→聚集→排序. 谓词的判断和聚集操作都是基于复合码进行的. 其中扫描是最耗时间的，我们采用了多种技术手段及算法来提高扫描性能.

#### 5.2.1 并行谓词判断

并行谓词判断是系统默认的谓词判断方式. 同

传统的 one-by-one 谓词判断方式相比，并行谓词判断能同时执行多个谓词项的判断. 相比于 Blink 中的谓词判断算法[7]，我们的算法不用区分偶数列和奇数列. 对于等值谓词判断只需要执行 5.1.1 节中转换后的谓词判断即可，所有的等值谓词都是通过一次二进制位操作完成的；对于范围谓词判断，按照 5.1.2 节转换后的谓词执行，所有的谓词判断至多需要一次等值谓词判断和一次范围谓词判断.

5.2.2 批量谓词判断算法

批量谓词判断用于压缩数据. 当采用压缩格式存储数据时，相邻元组的高位是相同的，作用在相邻元组的共同位上的谓词，其执行结果也是相同的，此时只需要执行一次谓词判断，后续元组上的谓词判断便可以跳过不再执行，也不用解压操作.

以图 5 中两个元组为例，第 2 个元组的前 25 位和第一个元组前 25 位相同，因此当谓词判断作用于前 25 位时，如果第 1 个元组符合条件，第 2 个元组也符合，可以不再执行谓词判断. 同理，如果第 1 个元组不符合条件，第 2 个元组也不符合，同样也可以不再执行谓词判断. 排序后的事实表，越高层编码重复片段越长(只有最底层的没有重复片段)，因此，批量谓词判断可以显著加速复杂谓词判断的执行.

5.2.3 跳跃式扫描

数据仓库中的查询选择率往往较低，在用基于扫描的处理方式时，大部分 I/O 都可能是浪费的. 为了加速数据的访问，可以考虑在〈*key*，*value*〉上建立索引. 度量属性上的索引可直接利用通常的索引技术，本文不再深入讨论；但若在复合码上建立通常的一维索引(B+树等)却是不可行的：复合码形式上是一个普通字段，但本质上是一个多维信息的组合；而且数据仓库中的查询往往只涉及部分层次，从而导致索引建立顺序和查找顺序极有可能是不一致的. 比如，我们为以图 2 的编码规则形成的复合码建立了 B+树索引后，当我们查找 *Region*='*Asia*'的记录时，由于只比较该复合码中的 *Region* 层次对应的子编码，该 B+树也就无法处理.

为此，本文提出了跳跃式扫描算法. 跳跃式扫描是基于有序数据、相对于顺序扫描提出的，其核心思想是尽量多地跳过与查询无关的数据块. 为达到此目的，我们将每一块的第 1 条记录抽取出来形成一张跳跃表，查询时根据跳跃表项和谓词来判断哪些块可能存在符合条件的记录，同时排除不可能存在符合条件记录的数据块，从而减少 I/O 操作.

跳跃式扫描算法主要包括两个步骤：(1) 产生可能包含地址块的地址候选集；(2) 基于候选集进行数据的访问.

**定义 5**(最小候选码). 大于等于当前复合码并且符合谓词条件的最小复合码，简记为最小候选码，简记为 *MCCode*.

最小候选码举例. 假设一个复合码含四个层次，每一层次的局部编码的值范围分别为[1，4]，[1，7]，[1，10]，[1，15]. 要在该复合码中查找等于 *.2 *.3 的记录(* 代表任意值)，那么复合码 1.3.2.3 的最小候选码为 2.2.1.3，表示大于等于 1.3.2.3 并且各层等于 *.2. *.3 谓词的最小复合码. 若谓词判断为范围谓词判断，为查找第 2 个层次介于 2 和 4 之间，第 4 个层次等于 2 的记录，那么复合码 2.5.3.2 的最小候选码为 3.2.1.2. 在复合码 2.5.3.2 中，第 2 个层次 5 已不在要求的 2 和 4 范围之内，所以求其 *MCCode* 时，需向高层次进一，同时置其他值为最小值.

最小候选码的求解算法见算法 1. 算法 1 首先根据谓词判断，提取出每一层次的上限值，按照复合码编码顺序组合成 *uprKey*；同理，提取出所有下限值组成 *lowKey* (行 1～2)，然后从最高层到最低层[①]依次设置 *MCCode* 每一层的 *h_skey*：如果参数 *key* 中的当前层次的 *h_skey* 小于对应的 *lowKey* 中的 *h_skey* 值，将所有低于或等于当前层次的层次置为 *lowKey* 中对应的 *h_skey*(行 4～6)；如果其 *h_skey* 大于 *uprKey* 中对应的 *h_skey*，将继续往高层查找直至找到某一层次 *h_skey* 增加 1 后仍在其对应范围内(行 8～10)，然后将该层及低于该层的所有 *h_skey* 置为最小值(行 11～12). 如果到达最高层次后，仍无层次增 1 后在其范围内，说明该 *key* 值大于最大的 *MCCode*，也即该 *key* 不存在对应的 *MCCode*.

**算法 1.** *CalMCCode*.

输入：*key*，*predicates*

输出：*MCCode*

1. Hierarchy *lowKey*[] := *genLowBnd*(*predicates*)
2. Hierarchy *uprKey*[] := *genUprBnd*(*predicates*)
3. *pos* := *getFirstHierarchyNotInRange*(*key*，*predicates*)
4. IF *pos*>0 //*h_skey*<its low bound THEN
5. FOREACH *low_h* IN *pos*…*lowest_hrchy* DO
6. *key*[*low_h*] := *lowKey*[*h*]
7. ELSE IF *pos*<0 THEN //*h_skey*≤its upper bound

① 维度层次中的高低依据层次树中的层次定. 比如时间维度中，年层次是高层次，日层次是低层次.

```
8.  FOREACH h IN [pos]+1…highest_hrchy DO
9.   IF lowKey[h]≤key[h]+1≤uprKey[h] THEN
10.   key[h]=key[h]+1
11.   FOREACH low_h IN h-1…lowest_hrchy DO
12.    key[low_h]=lowKey[h]
13.   ENDIF
14.   IF h>highest_hrchy
15.    RETURN NULL //No MCCode
16.  END//FOREACH
17. ENDIF
18. RETURN key
```

**定理 1.**　一排序数据，如果某数据块包含符合条件的记录，该块的最大值必大于等于其前一数据块中最大复合码的最小候选码. 对于第 1 个数据块取其最小复合码作为其前一数据块的最大复合码值.

证明.　设 $P$ 为一谓词，$block_i$ 为有序数据块列中的某一数据块，$minKey_i$ 为 $block_i$ 中的最小 $md\_skey$ 值，$maxKey_i$ 为最大值；$c\_key$ 是 $block_i$ 中满足谓词 $P$ 的某记录的 $md\_skey$ 值，$minMCCode_i$ 为 $minKey_i$ 的 $MCCode$，$maxMCCode_i$ 为 $maxKey_i$ 的 $MCCode$. 我们可以做如下推理：

(1) 由于所有的 $md\_skey$ 值是有序的，可推出：$maxKey_{i-1}\leq minKey_i$；

(2) 由 $maxKey_{i-1}\leq minKey_i$ 可推出：$maxMCCode_{i-1}\leq minMCCode_i$；

(3) 已知 $minMCCode_i=\min(\{key\mid key\geq minKey_i$ 并且 $key$ 满足谓词 $P\})$，$c\_key$ 是 $block_i$ 中一个通过谓词判断的记录的 $md\_skey$ 值，因此

$c\_key\geq minMCCode_i\geq maxMCCode_{i-1}$；

(4) 因为 $maxMCCode_{i-1}\leq c\_key\leq maxKey_i$，那么 $maxMCCode_{i-1}\leq maxKey_i$.

为便于计算，我们取下一数据块的第 1 个复合码值(该块的最小复合码)作为当前块的最大复合码. 产生待扫描数据块地址列表的算法如算法 2 所示.

**算法 2.**　*GetCandidateBlockAddressList*.

```
输入：JumpTable，lowKey，uprKey
输出：addressList
1. minKey=lowKey
2. MCCode=CalMCCode(minKey,[lowKey,uprKey])
3. FOREACH h_key IN JumpTable DO
4.  IF key≥MCCode THEN
5.   add corresponding address to addressList
6.   MCCode=CalMCCode(key,[lowKey,uprKey])
7.  ENDIF
8. ENDFOR
9. RETURN addressList
```

算法 2 首先根据谓词判断 $predicates$，设置初始 $key$ 为所有层次的局部层次码 $h\_skey$ 的最小值(行 1)并计算其 $MCCode$ 作为当前 $MCCode$(行 2)，接着将跳跃表中每一项同当前的 $MCCode$ 作对比(行 3～4)，如果当前的 $key$ 满足定理 1，我们将其加入结果列表中并计算新的 $MCCode$(行 5～6). 重复以上步骤，直至跳跃表中所有项都得到处理.

数据文件中的一个块对应跳跃表中一个表项，其所占空间是(数据文件大小/块大小)×复合码长度. 以 SSB 30 GB 数据量为例，按照非压缩格式存储的某一度量对应的跳跃表所占空间为(1.67 GB/8 KB)×10 Bytes，只占数据文件的 1.2‰. 跳跃表是针对特定数据文件建立的. 在大规模机群下，数据已经被均分于每个数据节点，因此，每个节点上的数据文件所占空间一般属于 GB 级，因此跳跃表的空间一般属于 MB 级. 为了加速跳跃表的检索速度，也可以在跳跃表之上再建立二级甚至更多级跳跃表.

### 5.2.4　Scan-index 扫描

Scan-index 扫描算法主要是针对基于列存储模型的多列查询提出的优化算法. 事实表按列存储，可以有效去除不必要字段的 I/O 代价，但也带来另外一个问题：当查询涉及多个度量值时，往往需要将这些度量值连接起来进行某些操作. 这也是列存数据库涉及多列查询时性能较低的主要原因之一.

Scan-index 是在程序运行过程中创建的内存索引，它可以在每一个度量的访问过程中动态更新，每一个度量的访问都可以基于前一个度量更新的 Scan-index 进行数据的访问(第 1 个度量的访问只能基于全列扫描或者跳跃式扫描的方式实现). 在此，我们仍就合取范式谓词判断进行讨论. 其基本思想来源于如下：对于同属一个元组的多个列来说，如果在某一次谓词判断时，某列不符合条件，该元组将不符合整个合取范式，也就没有必要再访问后续列. 本文用一个位图来实现 Scan-index 索引，其中每一位标识其对应偏移量的元组是否符合条件，true 表示符合，false 表示不符合.

由于所有数据聚簇存储，而且数据仓库查询的选择率较低，因此该位图中会出现大量的连续 true 或者 false. 为节省空间，对于非压缩数据我们采用如下三元组[二进制标识，开始位置，长度]来压缩存储该位图. 后续度量的块地址可以通过元组偏移量除以块大小计算出，块内偏移量可以通过元组偏移量对块内记录数求余得出.

对于压缩数据，由于不同数据类型的度量对应

的数据块内存储的元组数往往不同，因此较难根据元组偏移量直接计算出元组所在数据块及元组在数据块内偏移量.此种情况下，Scan-index 算法只适用于作用在相同数据类型的度量上的查询.由于度量基本为数值型且数据类型较少，因此，此限制并不会对 Scan-index 算法在压缩数据上的应用产生较大影响.我们用如下结构体数组以块为单位存储基于压缩数据产生的位图：[含满足谓词条件的元组的块在数据文件中的偏移量，该块内元组对应的位图].以 SSB 30 GB 数据集为例，选择率最高的 Q4.1 查询中，该位图所占空间仅为 93 KB.

Scan-index 扫描以两步执行：(1) 依据全列扫描或者跳跃式扫描产生初始的 Scan-index 位图；(2) 循环执行剩余谓词直至所有谓词都被执行完.基于前一列产生的 Scan-index 位图访问当前列，执行同该列相关的谓词并更新 Scan-index 索引位图中的相应位.其中，(2) 中操作主要完成事实表中同度量相关的谓词判断.

OLAP 查询访问的列大多为维度的层次和事实表的度量(MOLAP 中基于数据立方体上的查询即是例证)；在 JFSS 模型中，事实表的复合码已经包含所有的维度层次信息.因此对于大多数查询来讲，第 1 次扫描后即可确定哪些元组符合查询，后续列的访问都可以基于初始的 scan-index 位图进行数据访问.以如下投影操作为例：

$$\pi_{m1,m2}(\sigma_{md\_skey\&\text{“11100000111”}=\text{“10001000101”}}(factTable)).$$

基于 scan-index 的执行示意如图 7 所示.第 1 个度量的访问需要执行全表扫描操作，而第 2 个度量的访问就可以基于第 1 次扫描产生的 Scan-index 索引直接进行访问.

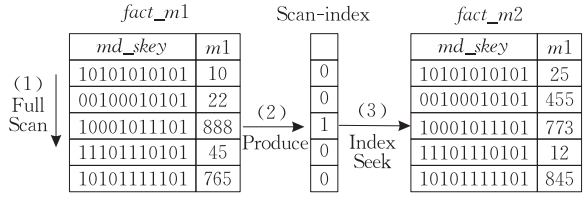


图 7 Scan-index 举例

一个元组对应 Scan-index 中一二进制位，因此，Scan-Index 所占总空间可元组数计算得出.以 SSB 30 GB 数据集为例，大多数查询产生的 Scan-index 索引所占空间都在 KB 级.如果采用压缩技术，该空间可以进一步缩小.

### 5.3 Merge

每一个数据节点执行完自己的 Reduce 操作后，将本地最终的聚集数据上传至主节点进行合并，并将 *md_skey* 复合码中的代码按照用户要求完成名称替换等工作.如果查询中存在 having 或者排序操作，也在此阶段执行.

各个数据节点只是将聚集后的数据上传至主节点，因此网络传输量和合并操作代价也是较低的(详细分析见 6.1 节).

## 6 实验分析

基于以上论述，我们基于 C++和 MPI(版本为 Open MPI 1.4.3)实现了 LaScOLAP 原型系统.实验采用的测试集为 SSB① 基准测试.实验机群包含 14 个节点，每个数据节点运行 Ubuntu 10.10 操作系统，配备一 Intel 酷睿 2 双核处理器、2 GB 内存和 140 GB 磁盘容量.网络带宽为 1 Gbps.

我们选择大数据分析的主流平台——MapReduce 类似系统作为对比对象，重点关注于 LaScOLAP 的扩展性、容错性及性能的验证分析.鉴于扩展性难以通过实际规模验证，我们采用了理论分析的方式(见 6.1 节).在性能的对比实验中，基于如下考虑，我们选择了 HadoopDB[2] 作为对比对象：(1) HadoopDB 是基于 MapReduce 和关系数据库的混合式数据库，在恰当的数据分布策略下，可同时具备 MapReduce 的扩展性和关系数据库的性能；(2) HadoopDB 代表 MapReduce 类似系统的最佳性能.大多数情况下，HadoopDB 的性能优于 Hadoop② 及 Hadoop++[8].在本实验中，我们又通过优化的数据分布策略，将所有查询都推入数据库层执行，从而使 HadoopDB 的性能达到最优；(3) 所有查询都在数据库层执行，也侧面对比了 LaScOLAP 同关系数据库的性能.

### 6.1 扩展性分析

根据 TRM 执行模型，可以将 LaScOLAP 的总处理代价(*TotalCost*)分为 3 部分：查询转换代价(*TC*)、数据节点 Reduce 代价(*RC*)、聚集结果网络传输及合并代价(*MC*)，即 $TotalCost=TC+RC+MC$.查询转换代价主要用于谓词中层次代码的查找操作：从编码后的维表中查找出查询谓词中涉及层次的局部层次码；各个数据节点操作代价主要是对本地数据的 I/O 访问；聚集结果的网络传输及合

① O'Neil P, O'Neil E, Chen X. The Star Schema Benchmark (SSB). http://www.cs.umb.edu/poneil/StarSchemaB.PDF

② Hadoop. http://hadoop.apache.org/

并代价取决于结果集大小.

由于按照 JFSS 模型组织的维表占用空间一般属于 GB 级或以下，我们可以按照 4.1 节的论述，采用全内存操作或基于索引的外存查找，以快速地完成查询的转换. 数据仓库查询具有高输入低输出的特点. 以 SSB 为例，任何一个查询的输出记录数都在 1000 条以下. 因此，即便每个节点按输出 1000 条〈*key*，*value*〉对计算，其传输的数据量仅为 1000×14 Bytes=13 KB. 这在 1 Gbps 的普通带宽上也只需 0.1 ms 即可完成传输. 同时，较小的结果集(KB 级)使得结果集合并操作可迅速完成. 因此，网络传输和合并操作代价 *MC* 只会占总处理时间的较小比例.

在整个星型模型中，事实表占据绝大部分空间，因此各个数据节点的处理时间应该占整个查询处理时间的较大比例. 每个节点的处理时间主要用于本地事实表扫描操作，因此 *RC* 应该正比于事实表数据量(如图 8 所示). 假设事实表的大小为 $F$，节点数为 $N$，则 $RC=f(F/N)$. 图 8 显示了 LaScOLAP 在单节点上执行 Reduce 操作时(输出聚集结果集为 800 条记录的情况下)，其执行时间随数据规模增长的变化情况. 从图中我们可以看出，*RC* 同数据规模成近似正比. 因此可推出，节点数的增加(对应数据规模的下降)，Reduce 操作执行时间会线性减少.

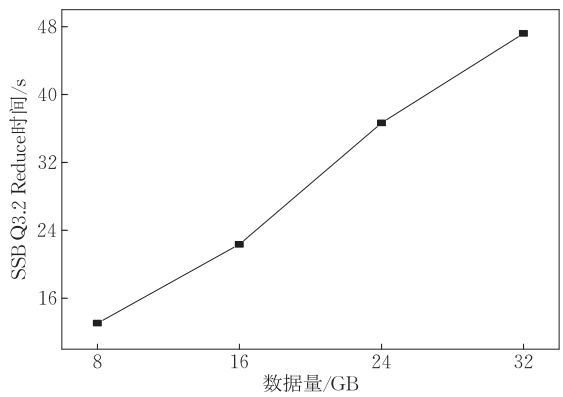


图 8　Reduce 执行性能同数据量的关系

基于以上分析，我们对查询转换时间(*TC*)和合并时间(*MC*)占总处理时间(*TotalCost*)的比例进行了测试分析. 实验采用 30 GB 数据集，基于两个节点进行(一个为主节点，一个为从节点). 我们选择 Q3.2 作为测试查询. 该查询输出结果集较大(较大的 *TC*+*MC*)，同时只涉及一个度量的访问(较小的 *TC*)，因此依据该查询测出的比例代表较大的 (*TC*+*MC*)/*TotalCost* 值. 测试结果表明(*TC*+*MC*)/*TotalCost* 值都在 0.12%以下. 因此 $TotalCost \approx RC = f(F/N)$，说明系统的性能同每个数据节点的数据量基本上是成正比的，也即意味着，LaScOLAP 的扩展性几乎是线性的.

### 6.2　性能分析

同 HadoopDB 对比时，我们采取如下数据分布策略将所有查询下推至数据层执行：(1) 按照最大维表 customer 表的主键 *c_custKey* 进行事实表和 customer 表的划分，以保证相同 *c_custKey* 的事实表记录和 customer 表的记录位于同一数据节点内；(2) 在每个数据库节点复制其他维表；(3) 采用手工编码方式实现查询，保证所有查询都在 PostgreSQL 数据库中执行. 此种方式对于 HadoopDB 的商业化版本 Hadapt① 也是最优的. 同时，我们将每个 PostgreSQL 的 *work_mem* 参数配置为 200 MB，*share_buffers* 参数为 100 MB，以使其发挥较高性能. 本实验采用 500 GB SSB 测试数据，HadoopDB 的数据库采用 PostgreSQL 9.0.2 版本，Hadoop 为 0.20.2 版本. 为排除负载均衡等对性能的影响，我们将数据冗余设为 1.

从图 9 可以看出 LaScOLAP 基于非压缩数据的平均执行性能是 HadoopDB 的 13 倍. LaScOLAP 性能优于 HadoopDB，主要得益于〈*key*，*value*〉存储方式和基于扫描的数据处理方式. 前者有效地减少了不必要的 I/O 操作，后者降低了查询处理的代价. HadoopDB 在执行 Q3.1 查询时，耗费较长时间，主要原因在于该查询选择率较高，引入较高的物化代价. 关于 LaScOLAP 第 1 组查询执行时间较长的问题我们在下一节讨论.

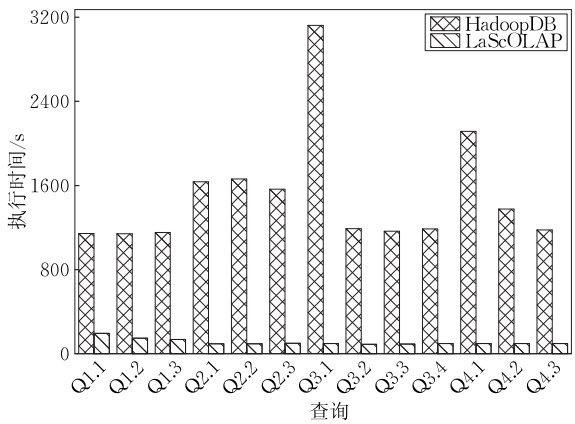


图 9　LaScOLAP 和 HadoopDB 性能对比(无索引)

### 6.3　LaScOLAP 各种优化算法性能分析

机群规模较大时执行情况较复杂，不便于跟踪分析，因此在分析各种优化算法时，我们选择了两个节

① Hadapt Inc. http://www.hadapt.com

点组成的机群进行分析:一节点作为主节点,另一节点作为数据节点,选用的数据集为 30 GB SSB 数据.我们对比了 LaScOLAP 基于非压缩数据及压缩数据的顺序扫描性能和跳跃式扫描性能(如图 10 所示).

从图 10 中可以看出,LaScOLAP 执行顺序扫描时,压缩数据平均处理性能是非压缩格式的 1.67 倍.然而,基于压缩数据上跳跃式扫描性能和非压缩数据上的跳跃式扫描性能差别不大.主要原因在于两者的 I/O 量接近.

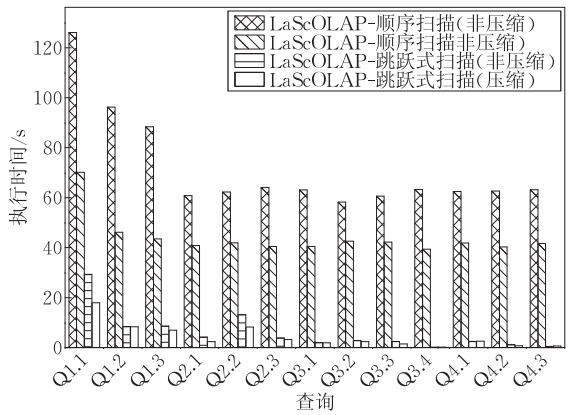


图 10 LaScOLAP 跳跃式扫描与顺序扫描性能对比

跳跃式扫描可以显著提高扫描的性能.跳跃式扫描平均性能是压缩数据的顺序扫描性能的 8 倍,是非压缩数据性能的 13 倍多.第 4 组查询涉及两个度量,但其总的执行时间同只涉及一个度量的第 2 组和第 3 组查询较为接近,说明 Scan-index 算法可以有效减少第 2 个度量扫描的 I/O 量.

我们也观察到第 1 组查询在执行各种优化算法时,时间都是最长的.原因在于第 1 组查询涉及 3 个度量,而且只有一个谓词作用于维度层次上,其他谓词均作用于度量上,从而导致初始 Scan-index 产生了较多符合谓词的元组,引起较多 I/O 操作.

查询 Q2.2 执行跳跃式扫描的时间相对较长,原因在于该查询中谓词判断作用于较低层次,排序聚簇后同该查询相关的记录分散于较多数据块,导致 I/O 量相对较大.对此,我们也进行了进一步的验证.基于非压缩数据,我们对跳跃式扫描处理的数据块占关系总空间的比例及各查询的选择率进行了统计对比,如图 11 所示.从图中可以看出,跳跃式扫描需扫描的数据块数比例大体上同选择率是一致的:选择率高,扫描的数据块就多;选择率低,扫描的数据块就少.平均只需要扫描 5%(最多需要扫描 25%)的数据块即可完成查询.这也侧面说明排序后的数据聚簇有效减少了 I/O 访问.然而,从图 11 中我们也可以看到两个异常点:Q2.2 查询跳跃式扫描的数据块占总数据块的比例远高于其查询选择率,而 Q3.1 远低于其查询选择率.我们分析这主要在于两个查询操作的维度的优先级及层次不同:Q3.1 查询是作用于最高优先级的 3 个维度(*date*→*supplier*→*customer*)的最高层次,因此排序后的数据可以获得较好的聚簇效果,从而减少 I/O 较多;而 Q2.2 作用于最低优先级的 part 维度的较低层次,因此不能获得像 Q3.1 查询一样的聚簇效果.这说明查询的层次的高低会较大地影响跳跃式扫描的性能.

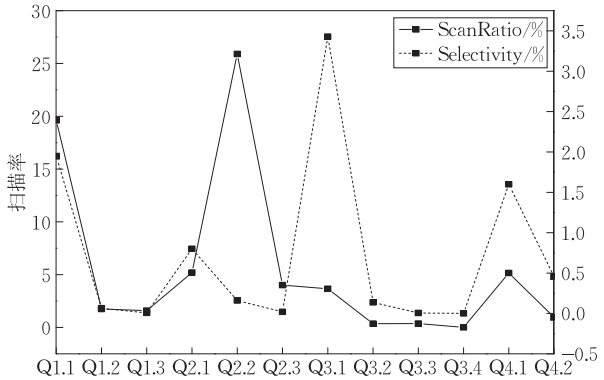


图 11 压缩数据跳跃式扫描访问的数据块占总数据块比例与查询选择率

### 6.4 容错性分析

LaScOLAP 的容错实现目前较为简单,只是通过重做失败节点的任务来获得容错能力.如果中心节点在一预设的时间内没有收到数据节点的心跳信息,它就会在其备份节点上重新执行其任务.为了测试 LaScOLAP 的容错能力,我们选择 Hadoop 为对比对象,Q3.2 作为测试查询.对于 Hadoop,将每个节点的 Map 任务个数和 Reduce 任务个数设为 1,采用最高效的广播方式实现连接①;对于 LaScOLAP,每个节点启动一个 Reduce 任务.二者均在查询执行到一半时随机断掉一个节点.结果显示,LaScOLAP 和 Hadoop 的执行时间分别发生了 16% 和 12% 的增长(绝对时间的增长分别为 35 s 和 142 s).从时间增长的比例上看,LaScOLAP 容错性弱于 Hadoop,其主要原因在于 Hadoop 实现了有效的预测执行、分阶段(Map 阶段和 Reduce 阶段)容错等优化机制,而 LaScOLAP 当前采用的容错机制较为简单.

未来,我们计划借助检查点机制来改善 LaScOLAP 的容错能力:由于 OLAP 查询的结果集较小,在 LaScOLAP 执行过程中,可以周期地将每个任务的执行状态信息(如已扫描记录的位置信息)、

① 每个 Map 任务从 HDFS 上直接读取维表数据并在内存中建立散列表,然后基于散列连接来实现星型连接.

中间聚集结果等增量地上传至可靠节点；一旦某结点任务执行失败，LaScOLAP 可以基于此检查点信息从失败点继续执行，从而避免整个任务的重新执行.

### 6.5　批量谓词判断分析

我们分别选择具备简单谓词判断 Q2.1（仅含一个由两个等值谓词判断组成的合取范式）和复杂谓词判断的 Q3.3（含 1 个由 4 个合取范式组成的析取范式）为代表批量谓词判断进行分析. 测试结果见图 12.

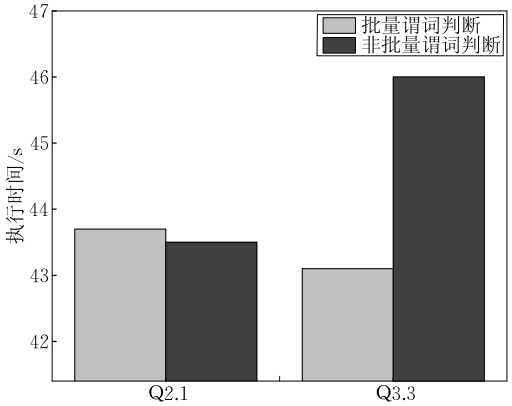


图 12　Q2.1 和 Q3.3 批量谓词判断性能分析

从图中可以看出，在执行简单谓词判断时，批量谓词判断的性能略低于通常方式（对应图中非批量谓词判断方式）. 但面对复杂谓词判断（Q3.3），具备明显优势.

## 7　结论及未来工作

为解决大数据上的星型（雪花）查询性能问题，本文从如下两个方面入手：（1）高度可扩展的架构. 利用层次编码技术对传统的星型（雪花）模型进行改造，消除了事实表与维表之间的复杂连接操作，提出了基于扫描的无连接雪花模型，将数据仓库查询的处理抽象为 Transform、Reduce、Merge 3 个操作（TRM 执行模型）. 新的数据模型和新的执行模型使得 OLAP 查询经转换后可被划分为众多独立子任务，在大规模机群上并行执行，从而保证了数据计算的独立性和系统的高度可扩展特性；（2）快速的本地计算算法. 在 I/O 仍是本地计算瓶颈的前提下，提出尽可能跳过不相关 I/O 的新优化途径. 提出了跳跃式扫描、Scan-index 等新型扫描算法. 为了加速本地的计算速度，本文也提出了针对数据仓库应用的〈*key*，*value*〉数据存储格式及前缀压缩算法、批量谓词判断算法等. 实验分析表明，原型系统 LaScOLAP 可以获得近乎线性的扩展能力，在执行 OLAP 查询时，其性能也比 HadoopDB 高出一个数量级.

未来仍有许多有挑战性的工作.（1）TRM 基于扫描的执行方式同 MapReduce 的执行方式是类似的，使得 LaScOLAP 有可能移植到 Hadoop 框架上，从而同时实现关系数据库的高性能特性和 MapReduce 的大规模可扩展特性；（2）维表更新的处理. 维表并非一成不变，偶尔也会发生成员的增加、删除、更新，甚至层次的变化等，因此如何高效地实现维表和事实表中层次编码的同步更新是 LaScOLAP 必须解决的问题；（3）查询共享技术. 在多个查询间共享磁盘 I/O 等操作，分摊 I/O 代价；（4）优化器设计. LaScOLAP 依赖扫描操作来处理查询，可以设计多种独特的扫描方式，如本文中的跳跃式扫描、传统的基于索引的扫描等，如何选择一条最优的访问路径是系统性能优化的关键问题之一.

## 参　考　文　献

[1] Abouzeid A, Bajda-Pawlikowski K, Abadi D, et al. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads//Proceedings of the 35th International Conference on Very Large Data Bases. Lyon, France, 2009: 733-743

[2] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters//Proceeding of the 6th Symposium on Operating System Design and Implementation. San Francisco, USA, 2004: 137-150

[3] Markl V, Ramsak F, Bayer R. Improving OLAP performance by multidimensional hierarchical clustering//Proceedings of the 1999 International Database Engineering and Applications Symposium. Montreal, Canada, 1999: 165-177

[4] Karayannidis N, Tsois A, Sellis T K, et al. Processing star queries on hierarchically-clustered fact tables//Proceedings of the 28th International Conference on Very Large Data Bases. Hong Kong, China, 2002: 730-741

[5] Bayer R. The universal B-Tree for multi-dimensional indexing: General concepts//Proceeding of the 1st International Conference on Worldwide Computing and Its Applications. Tsukuba, Japan, 1997: 198-209

[6] Korth H F, Kuper G M, Feigenbaum J, et al. SYSTEM/U: A database system based on the universal relation assumption. ACM Transactions on Database System, 1984, 9(3): 331-347

[7] Raman V, Swart Gt, Qiao L, et al. Constant-time query processing//Proceedings of the 24th International Conference on Data Engineering. Cancun, Mexico, 2008: 60-69

[8] Dittrich J, Ruiz J-A Q, Jindal A, et al. Hadoop++: Making a yellow elephant run like a cheetah (Without It Even Noticing) //Proceedings of the 36th International Conference on Very Large Data Bases. Singapore, 2010, 3(1): 518-529

**WANG Hui-Ju**, born in 1979, Ph. D. His research interests include large cluster database, main-memory database.

**QIN Xiong-Pai**, born in 1973, Ph. D., lecturer. His research interests include query optimization, main-memory database, parallel database.

**WANG Shan**, born in 1944, professor, Ph. D. supervisor. Her research interests include high performance database, data warehouse and knowledge engineering

**ZHANG Yan-Song**, born in 1973, Ph. D., associate professor. His current research interests include main memory database and high performance databases.

**LI Fu-Rong**, born in 1989, M. S. Her research interest is cloud computing.

**Background**

The data in enterprise information systems continue exploding, which brings a challenge to today's data warehouse technologies. To deal with the ever-growing data volumes, enterprises can choose to deploy a data warehouse in one of the following two architectures—A traditional parallel DBMS or a cloud platform.

Parallel DBMS (PDBMS) is usually deployed on hundreds of high-end servers, where failures are relatively rare in such an environment. Once a failure happens, a parallel database can simply re-execute the query. When deployed on a cloud platform consisting of thousands of cheap machines, PDBMS meets the scalability wall. As the machines in a cloud platform are less reliable and more numerous, system failures become more common and less tolerable. The performance of PDBMS also reaches its peak quickly as the system scales out to more than several hundreds of machines.

While MapReduce-based system is superior to PDBMS in scalability and fault tolerance, it is inferior to PDBMS in efficiency. In particular, MapReduce-based systems are very inefficient at join operations, let alone star (snowflake) joins that involve a significant number of tables.

In recent years, a number of PDBMS have added MapReduce front-ends to their query processing engines. The examples include Teradata, Vertica, Greenplum and Aster Data. However, as such simple extensions do not touch the internal execution engines, they do not take the full advantages of either PDBMS or MapReduce. HadoopDB (commercialized version Hadapt), has started to tightly integrate PDBMS and MapReduce. Some improvements have been made on both scalability and efficiency. However, the results are still not satisfactory for data warehouse applications. If a join operation involves multiple join attributes, such as a star join, HadoopDB would incur excessive I/O and data transmission cost and loses its performance advantage.

In this research, we explore the feasibility of building a data warehouse that yields the scalability and fault tolerance of MapReduce and the performance of RDBMS based on relational theory and techniques. Our design considers two levels: the schema level and the execution level. On the schema level, we improve the star (snowflake) schema and propose the decomposed snowflake schema, which eliminates the star (snowflake) join in query processing. On the execution level, we propose TRM execution model, which handles all data warehouse queries by three operations: transform, reduce, and merge. Based on these techniques, we divide a data warehouse-style query into many independent tasks, which can be distributed and executed independently across a cluster.

This work is partly supported by the Important National Science & Technology Specific Projects of China ("HGJ" Projects, Grant No. 2010ZX01042-001-002), the National Natural Science Foundation of China (Grant Nos. 61070054, 61170013), the Fundamental Research Funds for the Central Universities (the Research Funds of Renmin University of China, Grant No. 10XNI018), and the Graduate Science Foundation of Renmin University of China (Grant Nos. 10XNH096 and 11XNH120).