门 1 6r000007070rei1015T03N1010gr005F0 0

1D1010301011010100m 0or0101r0104C701019100 D⁰

810tol0fn5010段10 3 的D 61010rCf10 00

1010700069-0T010110Q0

**第** **3** **章** **OLAP 与结构化数据分析**

**3.1** **联机分析处理(OLAP)与结构化数据分析**

联机分析处理 (Online Analytic Processing,OLAP, 也称为在线分析处理)是在以 星型模型(或者雪花模型)建模的数据仓库上进行多维分析。多维分析指的是从各个角度 对我们感兴趣的一些数量进行汇总分析，比如我们从地区、客户、时间等维度，对销售明 细数据进行汇总分析。

结构化数据分析是一个(包含联机分析处理)更为宽泛的概念。它运行在结构化数据 上(一般以关系数据库表进行建模),分析可以表达成一个SQL 聚集查询。数据库的模式 没有必要按照星型模型进行严格建模，查询可以是一个简单的汇总查询，没有必要从多个 角度进行复杂汇总。比如，我们需要从销售明细表上分析出最近一段时间内销售量最大的 前5种产品。我们直接按照产品ID 进行销售明细数据分组，汇总销售额，然后按照汇总 的销售额进行排序，提取前5种产品即可。

本章介绍传统的联机分析处理技术，以及近年来兴起的SQL on Hadoop技术。

**3.1.1** **从操作型的业务数据库向数据仓库抽取、转换和装载数据**

大量的业务系统采用关系数据库来进行数据管理，随着业务的不断发展，各个企事业 单位和政府部门积攒了大量的业务数据。通过分析这些数据，可以展示数据中隐藏的规 律，指导业务决策。

业务系统的后台数据库需要支持日常业务的持续运行，数据处理任务的响应时间要求 一般比较短，这类数据处理任务称为联机事务处理 (OLTP) 。 比如在银行里，存款、取 款、查账、转账的业务，要求的响应时间一般是几秒钟。

数据分析任务一般在大量数据上(甚至在整个数据库的数据上)运行，以便获得宏观 的汇总信息和分析结果。这类任务消耗大量资源，人们对其响应时间要求没有那么紧迫， 一般以分钟或小时计。为了避免这类数据处理任务占用过多资源，对日常业务的顺利运行



产生干扰，一般在业务数据库之外建立数据仓库系统。它从业务数据库抽取、转换和装载 (Extract Transform and Load,ETL) 数据，支持人们在上面执行各种分析任务。

**3.1.2** **数据仓库与星型模型**

Bill Inmon认为，数据仓库是面向主题的 (Subject Oriented)、集成的 (Integrated)、

非易失的 (Non Volatile) 和时变的 (Time Variant) 数据集合，用以支持管理决策。面 向主题的是指每个数据仓库对应于企事业单位决策所包含的所有分析对象(数据)。集成 的是指数据仓库按照决策主题选择数据，把分布在各个部门中的多个异构数据源的数据集 成起来，并且以新的数据模型来存储。非易失的是指数据仓库的数据装载以后， 一般不会 删除。时变的是指随着业务的发展，新的业务数据不断地被抽取和装载到数据仓库中，以 便进行分析。数据仓库一般不对应到某个厂商的具体产品，而是指一种面向分析的数据存 储方案。

业务数据库上主要执行操作型处理任务，数据仓库上主要执行分析型处理任务。业务 数据库的新数据不断通过抽取、转换、装载过程，装载到数据仓库中，这就是企事业单位 采用的数据处理基础设施架构。虽然某些数据库系统，比如 SAP HANA内存数据库等， 宣称支持OLTP 和 OLAP 混合负载，但是它主要应用于分析型应用场合，并支持数据的 快速更新，一般不会把它当作运行操作型业务的基础数据库平台，同时在上面运行大量的 分析负载。表3-1展示了操作型数据处理和分析型数据处理的主要区别。

**表3-** **1** **操作型数据处理和分析型数据处理的差别**

|  |  |  |
| --- | --- | --- |
| 比较项目 | 操作型数据处理 | 分析型数据处理 |
| 数据模型 | 实体-关系模型(ER模型) | 星型模型和雪花模型 |
| 操作的记录数量 | 少量记录 | 大量记录 |
| 数据是否可以更新 | 数据可以更新、删除 | 一般只对数据进行追加、不删除、极少更新 |
| 响应时间要求 | 秒级 | 分钟级、小时级 |
| 目 的 | 支持业务运行 | 支持决策需求 |

数据仓库主要采用星型模型进行数据建模。在这个模型中，包含事实表和维表。事实 表记录具体的业务交易，比如记录客户的购物信息。维表记录分类信息，比如时间信息、 地理区域信息、产品分类信息等。

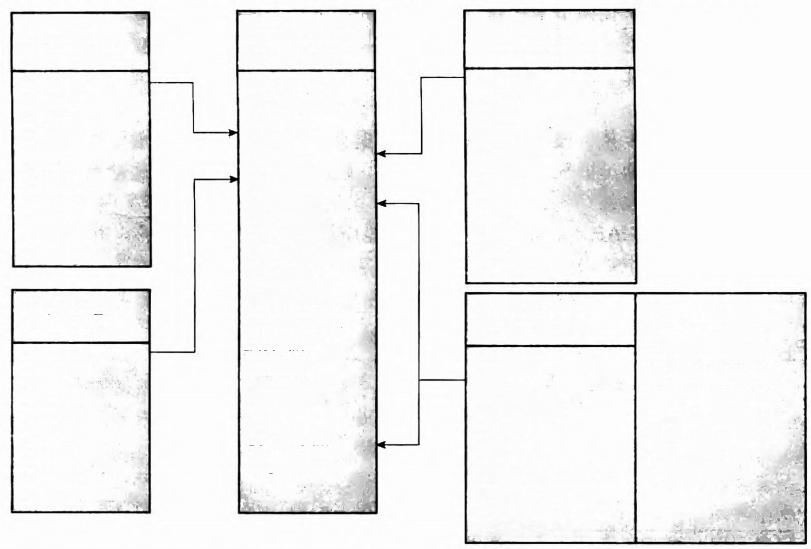
每个维表代表人们观察数据的一个角度。一般维表具有层次结构 (Hierarchy), 即人 们观察事物的不同细节，比如时间维度包括年、季度、月份、日期等不同细节的层次 (Level)。维表的具体的一个取值称为维的成员 (Member)。 比如，某年某月某日是时间 维(时间维的最低层次是日期)的一个成员。事实表通过外键 (Foreign Key) 和维表关 联起来，它除了和维表建立联系之外，更重要的是它记录了业务的一些度量 (Measure) 信息。比如，购物信息里的商品单价、商品数量、价格小计等，这些数值型属性称为度量 值。下面我们介绍SSB 测试基准的数据模型，该模型为一个典型的星型模型。

SSB(Star Schema Benchmark) 是麻省州立大学波士顿校区的研究人员研发的基于现 实商业应用的数据仓库测试基准。该测试基准被学术界和工业界广泛接受，用来测试决策 支持类应用中的数据库系统性能。

**这个测试基准包含数据模型、工作负载以及性能指标三个方面。图3-1显示了该测**

试基准的数据模型，包含 一 个事实表和四个维表。事实表LineOrder 记录订单信息，维表 Customer 记录客户信息，维表Part 记录配件信息，维表Date 记录时间信息(具有年/月/ 日概念层次关系),维表Supplier 记录供应商信息。图3-1中，每个数据库表的名称后面 的括号里是字段名的前缀，比如 LineOrder 表的字段名的前缀为LO\_, 那么OrderDate 字段的全名应该为LO\_OrderDate。

事实表通过外键和维表建立主外键关系，包括通过外键lo\_custkey 和 Customer 的主 键c\_custkey 建立主外键关系，通过外键lo\_partkey和 Part表的主键 p\_partkey 建立主 外键关系，通过外键lo\_suppkey 和Supplier表的主键s\_suppkey 建立主外键关系，以及 通过外键lo\_orderdate 和时间维表DwDate的主键 d\_datekey 建立主外键关系等。



Part(P\_)

200000\*[1+log₂SF] PartKey

Name

Mfgr

Category Brandl

Color Type

Size

Container

Date(D\_)

7Years of Days DateKey

Date

DayOfWeek Month

Year

YearMonthNum YearMonth

DayNumInWeek

LineOrder(LO\_) SF\*6000000

OrderKey

LineNumber

CustKey

PartKey

Supp Key

OrderDate

OrdPriority

ShipPriority

Quantity

ExtendedPrice OrdTotalPrice Discount

Revenue

SuplyCost

Tax

CommiDate ShipMode

Customer(C\_)

SF\*30000

CustKey

Name

Address

City

Nation

Region

Phone

MktSegment

Supplier(S\_) SF\*2000

SuppKey Name

Address City

Nation Region Phone

DayNumlnMonth DayNumlnYear WeekNumlnYear

MonthNumlnYear SellingSeason

LastDaylnMonthFL HolidayFL

WeekDayFL

**图** **3** **-** **1** **Star Schema Benchmark(SSB) 数据模型**

**3.1.3** **联机分析处理** **(OLAP)**

数据仓库模型建好并且装载数据以后，我们就可以对数据进行分析。数据仓库上的分 析任务包括简单分析和复杂分析。简单分析指的是利用数据生成报表以及进行多维分析。 复杂分析指的是在数据上运行复杂的统计方法、机器学习和数据挖掘算法，从而发现不是 那么显然的规律，发现新的知识。本章主要讲述业务数据上的简单分析。

在这里，简单分析主要是指联机分析处理，包括固定报表和多维分析，其表现形式是 在数据仓库数据上执行查询获得汇总信息。由于这些分析无须在数据上进行多次扫描和执 行复杂的处理逻辑，所以其响应时间得到很好的控制，在高性能的数据仓库系统上，比如 SAP HANA以 及 MonetDB, 甚 至 可 以 获 得 秒 级 的 响 应 时 间 。 图 3 - 2 展 示 了 操 作 型 数 据 库、数据仓库及其上面的工作负载。



|  |  |  |
| --- | --- | --- |
| 操作型数据处理 | 分析型数据处理 |  |
| 数据服务与 联机事务处理 | 简单分析  (联机分析处理) | 复杂分析(统计分析、 数据挖掘、机器学习) |

抽取、转换、装载 业务数据库

数据仓库

**图3-2** **操作型数据库与数据仓库**

联机分析处理((LAP) 是数据仓库的主要负载和应用，通过分析操作，为高层管理 人员提供决策支持。 OLAP 系统根据分析人员的要求，快速灵活地进行大量数据的复杂查 询处理(虽然是复杂查询处理.但是相对于数据挖掘来讲，其处理逻辑仍然是简单的)。 OLAP 系统的前端软件把查询结果以直观的图表方式提供给决策人员，方便他们掌握业务 状况，做出正确的决策。

联机分析处理的主要操作包括下钻 (Drill Down)、上 卷 (Drill Up)、切片 (Slice) 、 切块 (Dice) 、 旋 转 (Pivot) 等。下钻和上卷是改变维的层次，变换分析的粒度。上卷是 在某个分析维度上，将低层次的细节数据概括到高层次的汇总数据。下钻是在某个分析维 度上，从高层次的汇总数据深入到细节数据进行观察。比如当我们把商品销售情况表 (SSB 数据模型中的 Line(rder 表)的一部分记录在时间维度上按照月度进行分组汇总 (比如销售额)以后，上卷操作则把分析的粒度变成年度，即按照年度进行分组汇总，下 钻操作则把分析的粒度变成日期(即以天作为汇总的粒度)。切片和切块是选定一部分维 度值，然后查看度量数据在剩余维度上的分布情况。如果剩余的维度只有两个则称为切 片，如果剩余维度有三个或者以上称为切块。比如在SSB 模型中，我们选定某个(些)顾 客和供应商后，对销售数据在剩余的两个维度即配件维度、时间维度上进行汇总，就是切 片操作。如果我们只选定某个(些)顾客，对销售数据在剩余的三个维度，即供应商维 度、配件维度、时间维度上进行汇总，就是切块操作。

旋转操作是变换维的方向，也就是在汇总表格中(在Excel 中称为数据透视表),重 新安排维的位置，即对行/列进行互换。比如在表3-2中，我们对销售数据进行了时间、 供应商、配件三个维度的汇总。我们选择配件维度作为行方向的维度.时间维度和供应商 维度作为列方向的维度。假设我们仅2010年的4个季度进行销售额汇总.选择了两个供 应商和三个配件。为了简化表格设计，我们把合计数和行方向、列方向的维度的成员放在 一起。通过这张表(见表3-2).我们方便地观察到2010年不同季度、不同供应商、不同 配件的销售额，而且得到各个配件的合计数、各个季度的合计数，以及各个季度之下的各 个供应商的合计数。

**表3-2** **商品销售额汇总表a**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2010年第1季度 合计300 | | 2010年第2季度 合计333 | | 2010年第3季度 合计306 | | 2010年第4季度 合计224 | |
| 供应商1 合计105 | 供应商2 合计195 | 供应商1 合计108 | 供应商2 合计225 | 供应商1  合计111 | 供应商2 合计195 | 供应商1 合计99 | 供应商2 合计225 |
| 配件1 合计42 | 35 | 65 | 36 | 75 | 37 | 65 | 33 | 75 |

续前表

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2010年第1季度 合计300 | | 2010年第2季度 合计333 | | 2010年第3季度 合计306 | | 2010年第4季度 合计224 | |
| 供应商1 合计105 | 供应商2 合计195 | 供应商1 合计108 | 供应商2 合计225 | 供应商1  合计111 | 供应商2 合计195 | 供应商1  合计99 | 供应商2 合计225 |
| 配件2 合计421 | 35 | 65 | 36 | 75 | 37 | 65 | 33 | 75 |
| 配件3 合计421 | 35 | 65 | 36 | 75 | 37 | 65 | 33 | 75 |

注：表中的销售额数据仅供参考，非实际数据。

接着，我们把时间维度做行/列互换，把它作为行方向的维度，重新观察数据，新的 汇总表如表3-3所示。通过行/列互换以后，我们仍然观察到行列互换之前的各个Cell 的 合计数，还得到两个供应商的合计数、各个季度的合计数，以及各个季度下各个配件的合 计数等。

**表3-3** **商品销售额汇总表** **b**

|  |  |  |  |
| --- | --- | --- | --- |
|  | | 供应商1 合计423 | 供应商2 合计840 |
| 2010年第1季度 合计300 | 配件1,合计100 | 35 | 65 |
| 配件2,合计100 | 35 | 65 |
| 配件3,合计100 | 35 | 65 |
| 2010年第2季度 合计333 | 配件1,合计111 | 36 | 75 |
| 配件2,合计111 | 36 | 75 |
| 配件3,合计111 | 36 | 75 |
| 2010年第3季度 合计306 | 配件1,合计102 | 37 | 65 |
| 配件2,合计102 | 37 | 65 |
| 配件3,合计102 | 37 | 65 |
| 2010年第4季度 合计324 | 配件1,合计108 | 33 | 75 |
| 配件2,合计108 | 33 | 75 |
| 配件3.合计108 | 33 | 75 |

需要指出的是，我们不仅能够对汇总表进行行列互换，而且可以对行方向的维度进行 顺序调整，比如对表3-3的时间和配件的顺序进行调整，我们将得到各个配件的销售额 的合计数，各个配件在各个季度的销售额的合计数等。

**3.1.4 三种类型的OLAP 系统**

OLAP 系统以数据仓库作为基础，从数据仓库中提取详细数据的一个子集，经过聚集 操作，生成汇总结果(也可以事先计算汇总结果),供前端分析工具读取和展现。实现该 功能的软件称为OLAP 服务器。

按照数据存储格式分类，OLAP 系统可以分为多维 OLAP(Multi Dimensional OLAP,MOLAP) 、 关 系OLAP(Relational OLAP,ROLAP) 以及混合OLAP(Hybrid

OLAP,HOLAP) 三类。

MOLAP 将 OLAP 分析所用到的多维数据，在物理上存储为多维数组的形式，形成 “立方体”(Cube) 的结构。星型模型的各个维的属性值，被映射成多维数组的下标值或 下标的范围，聚集数据(即汇总数据)作为多维数组的值存储在数组的单元中。由于 MOLAP 采用了新的存储结构，从物理层实现(LAP, 因此又称为物理(OLAP(Physical OLAP) 。MOLAP 对数据进行预先汇总，由于实际应用中的星型模型存在若干维度，比如 销售数据仓库的星型模型包含地区、时间、产品等维度，此外，有些维度还存在层次结 构.比如时间维度存在“年-季度-月-日”的层次结构，在哪些维度、在什么维度层次上 进行数据的预先汇总，这样的组合是非常多的，每种组合生成的汇总数据，称为一个Cu- boid 。如果我们在各种组合上都事先计算汇总数据，其好处是进行多维分析时分析速度 快，但消耗的存储空间非常大。

ROLAP 将多维数据存储在关系数据库中，通过把OLAP 操作表达成 SQL 查询的形 式，在关系数据库上执行，获得汇总结果。对于一些使用频率比较高、计算工作量比较大 的查询，可以把它定义为物化视图，以提高查询的性能。物化视图 (Materialized View) 是把查询的结果存储到关系数据库表中，以备用户再次发起该查询时，无须从明细数据进 行扫描和汇总，直接从物化视图进行查询即可。ROLAP 通过一些软件工具或中间软件实 现，物理层仍采用关系数据库的存储结构，因此称为虚拟 (LAP(Virtual(LAP)。

M(OLAP 的优点是性能高.缺点是占用大量的空间保存汇总数据。 ROLAP 的优点是 采用关系数据库保存原始明细数据，占用空间有限。由于每个(LAP 操作都被转换成一个 SQL 查询重新执行，性能受到影响。MOLAP 和 ROLAP 有着各自的优点和缺点，结构迥 异，给开发人员设计(LAP 系统造成了困难。为此，人们提出一个新的(LAP 结构—— 混合OLAP(HOLAP), 它把MOLAP 和 ROLAP 两种技术结合起来，是这两种技术优点 的有机结合，能满足用户各种复杂的(LAP 分析请求。 HOLAP 基于混合数据组织实现 OLAP, 比如低层数据是关系型的，高层数据是多维数组型的，也就是将细节数据保留在 关系型数据库的事实表中，但是聚集后的数据保存在“立方体”(Cube) 中，这种方式具 有更大的灵活性。HOLAP 的查询效率比ROLAP 高，但低于MOLAP。

**3.2** **高性能OLAP系统的关键技术**

**3.2.1** **列存储技术**

在关系数据库的表格里，每个记录包含若干属性列的值。如果数据库表按照行存储格 式保存 (Row-wise Storage), 那么在每个数据块(或者页面)里面，每行记录的各个属性 列的值依次存放。如果数据块的大小，小于一个记录的大小，那么存储一个记录可能需要 超过一个数据块的空间。如果一个数据块的大小，大于一个记录的大小，那么一个数据块 就可能存放多个记录。

在联机事务处理((LTP) 类应用中，大部分的事务存取少量的记录，但是需要读取 和写入这些记录的所有(或者大部分)属性列。行存储对于(LTP 类应用是一种优化的存 储结构。

需要指出的是，在面向OLTP 应用的数据库系统中，数据以页面为单位存储在磁盘里 面，页面的大小一般是8KB,16KB 等。在面向数据分析的大数据管理系统中，比如 Ha- doop, 数据以数据块为基本单位保存在磁盘中，数据块的大小一般为64MB,128MB,

256MB 等。

图3-3展示了数据库表中的记录是怎么按照行存储格式进行存储的。这个数据库表 记录了人们的社会安全号、姓名、年龄、所在的州、城市和具体的街道地址等信息，我们 假设一个数据块正好存放两行记录。在实际应用中， 一般一个数据块(或者页面)可以存 放若干记录。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SSN** | Name | Age | Address | City | **State** |
| 1012599898 | John | 44 | 798 Main ST | Binghamton | NY |
| 892375863 | Smith | 37 | 1636 Riverside ST | Syracuse | NY |
| 318370709 | Wang | 23 | 455 June ST | Chicago | IL |
| 101259875 | Sun | 25 | 385 Clinton ST | Los Angles | CA |
| …… | | | | | |

|  |  |
| --- | --- |
| 1012599898John|44 798 Main ST|BinghamtonNY| 892375863|Smith|3711636 Riverside STISyracuse|NYI | 3183707091Wang|23 455 June STIChicago IL  101259875|Sun|25385 Clinton STILos Angles|CA |

Block 1 Block 2

**图3-3** **行存储示意图**

在面向分析型应用的数据库系统中， 一般采用列存储的方式保存数据。我们先了解列 存储的基本原理，然后分析它带来的好处。采用列存储，若干记录的某个属性列的值连续 存储在一系列数据块中，由此完成某个属性列的存储，对其他属性列，依此办法处理。

对于上述数据库表，因为它包含六列，所以需要六个数据块保存这些属性列，如果一 个数据块不能容纳某个属性列的所有值，则需要更多的数据块来保存，如图3-4所示。

Block1

Block2

Block3

Block4

Block5

Block6

|  |
| --- |
| 1012599898 892375863 318370709 101259875… |
| John|Smith|Wang|Sun… |
| 44|37|23|25|… |
| 798 Main ST|1636 Riverside ST|455 June ST|385 Clinton ST… |
| Binghamton|Syracuse|Chicago|Los Anglesi… |
| NYINYIILICA|… |

**图3-4** **列存储示意图**

使用列存储，带来的一个好处是，由于数据块包含相同数据类型 (Data Type) 的 值 ， 我们可以使用数据压缩技术，减少磁盘空间占用和存取这些属性列数据的I/O 操作开销， 从而加快数据处理过程。

其中最重要的一种压缩技术称为字典编码 (Dictionary Based Compression)。一 般来 讲，在一列数据中总是有些重复值，比如图3-3中用户所属的州，其取值为美国的50个 州之一，大量的用户在这个属性的取值会出现很多重复。除此之外，产品名称、产品编 码、城市名称等属性列，也有很多的重复值。我们可以建立一个字典，对这些取值分配一 个编码，建立编码到取值的对应关系。 一般来讲，编码长度(即它占用几个字节)一般比

这些属性原来的属性值小得多，于是我们达到了数据压缩的效果。比如有一个属性列，不 同的取值有上百万个，每个取值的大小从64字节到128字节不等，数据库表总共有500 万行，没有压缩之前，其占用的空间为5M×128=640MB 。 当我们采用字典编码方法， 100万个不同值 (Distinct Value), 需要四个字节作为编码的长度对其进行唯一编码即可， 数据被压缩到5M×4=20MB 。 选择操作以及连接操作可以通过查询改写，直接在编码以 后的字段值上进行操作，无须把编码恢复回原来的数据值。比如“CA” 州名，经过编码 以后，获得0001的编号，那么 “state='CA'” 查询条件，就可以改写成 “state=0001”。

分析型应用对数据的存取表现出这样的特点，它一般涉及数据库表大量的记录，但是 不会对这些记录的所有属性列进行操作，只需存取少部分数据库表的属性列，对它们进行 分组和汇总。比如一个数据库表有100个属性列，某个查询只需读取5个属性列，那么它 只需存取5%的数据量(相对于全表扫描)就可以完成查询。采用行存储，则95个不需要 的属性列也被读进来，查询处理做了很多无用功。采用列存储，相对于行存储，在处理上 述查询时只需提取必要的5个属性列数据就可以，需要付出的开销要少得多。

如下简单的聚集查询，目的是把2011年2月2日的所有销售明细提取出来，把每个 明细记录的单价和数量相乘，然后累加，获得该日的销售额。

|  |
| --- |
| **SelectSum** (price\*quantity) ◆ **From** sales  **Where** sales.date='2011-02-02' |

当采用行存储.查询的处理模式是，如果数据库表有索引，而且查询选择率 (Selec- tivity) 比较低，那么首先扫描索引，取得所有符合条件的记录的ID, 然后提取相应的记 录，在记录上进行投影，即选取单价 (price) 以及数量 (quantity) 列，然后进行汇总， 如图3-5所示。由于无关的数据列也被提取出来，当无关数据列的数量还比较多时，查 询的效率是比较低的。如果没有合适的索引可以使用，那么需要进行全表扫描，代价就更 大了，查询的效率就更低了。



Projection on price,quantity

**All Rows**

**Predicate Evaluation**

**Full Table Scan**

Rows

Rows

**图3-5** **行存储与查询处理(全表扫描)**

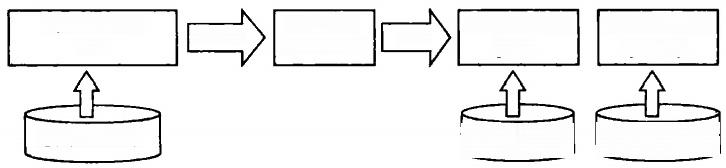
当采用列存储，查询处理器只需提取日期 (date) 、 单价 (price) 、 数量 (quantity) 等数据列，而且在提取日期列时，可以应用选择谓词，即使用查询条件 “sales.date=

2011-02-02'”对数据进行过滤，只提取该数据列的一部分，由此获得符合条件的记录 (所在的行)的偏移量。根据这些偏移量，继续提取相对应的单价 (price) 和数量 (quan- tity) 两个数据列，进行乘法计算，并进行汇总即可，如图3-6所示。

采用列存储，我们不仅可以节省I/O) 开销，而且可以节约内存空间，因为与查询无关 的属性列都没有被装载到内存中。虽然列存储具有便于压缩、加快查询等优势，但是它也 带来一些成本。当我们需要重新把一行记录组装出来时，代价是比较大的。由于隶属于同 一行的各个属性列的值保存在不同的数据块中，在查询涉及的数据列比较多的情况下，每



个属性列都需要单独的磁盘I/ (去提取数据。列存储一般适用于读取少量属性列进行分析 的应用场合，而不是经常对某些记录进行整体操作的场合。

Scan column date

Column date

Scan Column price

Predicate Evaluation

Scan Column quantity

values

IDs

Column price Column quantity

**图3-6** **列存储与查询处理**

在某些应用场合，完全把各个属性列切割开单独存放不一定是最好的方案。对于一些 经常一起访问的属性列，比如上述查询中的单价 (price) 列和数量 (quantity) 列，把它 们组合构成列分组 (Column Family) 一起存放，可以加快查询处理。

**3.2.2** **位图索引技术**

位图 (Bitmap) 索引是针对数据仓库的索引形式。对于低基数 (Cardinality, 即不同 值的数量)的字段可以建立 Bitmap 索引。下面以一个用户表为例，展示如何建立性别字 段上的 Bitmap 索引。性别字段只有“男”/“女”两个不同取值，因此它是一个低基数的 字段。假设用户表的实际数据如表3-4所示(某些无关字段值未显示)。

**表3-4** **Bitmap索引实例**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | 用户ID | 用户名 | 性别 | 出生年月 | 其他字段 | “男”Bitmap序列 | “女”Bitmap序列 |
| 1 | 5 | Tom | 男 | 1980.01 | … | 1 | 0 |
| 2 | 6 | Mary | 女 | 1980.02 | … | 0 | 1 |
| 3 | 7 | June | 女 | 1979.12 | … | 0 | 1 |
| 4 | 14 | Mark | 男 | 1979.11 | … | 1 | 0 |
| 5 | 15 | Kate | 女 | 1985.03 | … | 0 | 1 |
| 6 | 16 | John | 男 | 1982.02 | … | 工 | 0 |
| … | … | : | … | : | : | … | … |

表3-4的最后两列，分别是针对性别为“男”和性别为“女”的Bitmap 索引的具体 取值。我们看到，在针对性别取值为“男”的 Bitmap 索引中，当该记录的性别字段的取 值为“男”时 .其取值为1,否则为0,每个记录只需一个比特 (Bit) 保存索引信息。由 于 Bitmap 索引出现大量的连续的0和1,再经过压缩.其空间占用一般很小。

上述Bitmap 索引，支持查询条件为 “User.sex=‘ 男’”这样的查询。通过扫描针 对性别字段取值为“男”的Bitmap 索引，可以快速定位符合条件的记录为1号、4号和6 号记录。

从这个实例可以看出，当我们为某个字段建立 Bitmap 索引时， Bitmap 序列的个数是 字段的不同取值的个数 (Number of Distinct Value),未压缩情况下每个字段值对应的Bit

列表的Bit 数量为记录的个数。假设记录数为n, 字 段A,的基数为C,, 为字段A, 建立的

Bitmap 索引的大小为n×C;/8 字节。当 C,很大，即字段的基数较大时， Bitmap 索引的大 小急剧膨胀，建立 Bitmap 索引的代价相对于其所获得的收益变得不合适了。此外，标准

的 Bitmap 索引并不支持范围查询。

对于某些范围查询字段，其基数(即不同的取值)是巨大的，直接使用上述 Bitmap 索引方法，不仅空间开销大，而且并未支持范围查询，于是人们提出了分容器的 Bitmap 索引的思想。其实现机制是：把字段的值域 [Vlow,Vhigh] 划分成一个个的片段，每个片 段内包含一定的记录数量，称为容器 (Bin), 各个容器形成首尾相连的关系。

建立 Bitmap 索引时，不是为字段的每个不同值建立索引，而是针对每个容器建立索 引，也就是对整个值域的一个子集范围进行索引。当执行范围查询 “V₁<=A;<=V₂”

时，对于完全处在 [V₁,V₂] 之间的片段，直接提取其 Bitmap 索引，即可获得对应的记 录。对于与 [V₁.V₂] 部分重叠的片段，根据其 Bitmap 索引提取记录后，还需具体判断 字段值是否真正落在 [V₁,V₂] 范围内。

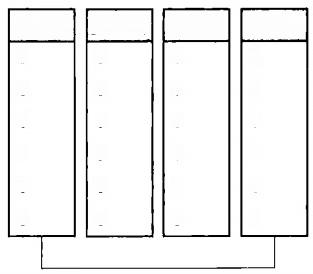
比如，对于资产总额字段，我们建立了(0,10),(10,20), …,(90,100)等片段 (假设最大资产数量是100万美元)。当查询“25<=总资产<=56”的用户时，[30, 40),(40,50)两个片段对应的记录肯定符合条件，把两者的Bitmap 索引进行或 (Bit- wise(R) 操作，就可以寻访到符合条件的记录。(20,30),(50,60)两个片段包含的记 录则有可能符合条件，也可能不符合条件，把两者的 Bitmap 索引进行或操作，寻访到相 应记录后，还需确认该记录是否符合“25<=总资产<=56”的条件，如图3-7所示。

表格数据

|  |
| --- |
| **用户ID姓名性别**出生年月 总资产其他字段总资产区间 |
| 5 T o m 男 1980.0122万美元 (20,30) |
| 6 Mary 女 1980.0257万美元 … (50,60) |
| 7 June 女 1979.1233万美元 … [30.40] |
| 14 Mark 男 1979.1125万美元 … [20,30] |
| 15 Kate 女 1985.0342万美元 … (40,50) |
| 16 John 男 1982.0253万美元 … (50,60) |

总资产字段的Bitmap索引





|  |  |  |
| --- | --- | --- |
| [20,30]  1  0  0  1  0  0 | (30,40)  0  0  1  0  0  0 | (40,50) [50,60]  0 0  0 1  0 0  0 0  1 0  0 1 |

查询条件“25<=总资产<=56”对应的范围

**图3-7** **分容器** **Bitmap 索引实例**

如果对字段的值域分割的粒度过大，那么在查询条件和片段部分覆盖情况下，需要提 取和判断的记录数量就越多。如果值域分割的粒度过细，那么需要为更多的片段建立 Bit- map 索引，造成更大的索引空间开销，需要折中考虑。

**3.2.3** **内存数据库技术**

随着内存价格的下降，我们可以为服务器安装更大的内存。对于中小规模的数据库来 讲，我们可以把数据库的所有数据都装载到内存。内存数据库 (Main Memory Database, MMDB) 和磁盘数据库 (Disk Resident Database,DRDB) 的主要区别在于其数据的主副 本在内存中，磁盘等外存仅仅作为数据备份的设备。

内存数据库分为面向OLTP 应用的内存数据库和面向(LAP 应用的内存数据库两类。 面向OLTP 应用的内存数据库，有来自IBM 的 SolidDB 数据库、来自Oracle 的 Timesten 数据库，以及 VoltDB 等。面向(LTP 应用的内存数据库支持高速的事务处理，应用于电

信业务、股票期货交易撮合、工业控制等需要实时 (Real-Time) 响应的应用场合。面向 OLAP 应用的内存数据库，有 SAP HANA 数据库、Vectorwise 数据库、MemSQL 数据 库等。面向(OLAP 应用的内存数据库，支持快速的数据查询和分析。下面我们从存储、索 引、查询优化、并发控制、恢复技术等方面.介绍内存数据库的相关技术。

**1.** **存储技术**

传统数据库系统，围绕磁盘来进行设计。数据的主副本保存在磁盘中，在需要存 取时装载到内存缓冲区 (Memory Buffer)。在内存数据库中，数据的主副本保存在内 存中，暂时不用的数据保存到磁盘，以便为将要用到的数据腾出空间。围绕内存重新 对数据库系统进行设计的策略称为 Anti-Caching 策略。 一旦数据完全保存在内存中， 如何快速有效地在内存和CPU Cache之间进行数据交换，成为性能优化的关键点。优 化的目的就是提高CPU 的指令缓存和数据缓存的命中率。降低Cache Miss。这和磁盘 数据库的优化重点是不一样的，在磁盘数据库里，我们重点考虑如何在磁盘和内存缓 冲区之间快速有效地交换数据。在内存数据库中，缓冲区管理器 (Buffer Manager) 不复存在。

对于一个数据库表 (Relational Table,关系表),数据可以按照三种格式来进行存储， 分别是行存储、列存储和 PAX。 行存储在一个数据页 (Data Page) 中，一行一行地存储 数据，每一行的数据是紧密排列在一起的。面向(LTP 应用的数据库一般采用行存储的存 储格式，因为OLTP 事务一般仅仅存取少量的数据行。列存储格式把关系表划分成各个 列，每个列单独连续存放。在分析型应用中.列存储格式比行存储格式获得超过一个数量 级的性能表现。在这类应用中，查询需要扫描大量的数据，但是仅仅存取少量的数据列。 当数据采用列存储格式，我们可以采用一些压缩技术来对各列数据进行高效的压缩，减少 空间占用。

PAX(Partition Attributes Across) 存储格式，是行存储和列存储的混合格式。首 先.数据库表被横向划分成若干页面，每个页面内部各个属性(列)的值被连续存放。 PAX 格式极大地提高了CPU Cache的命中率，在进行数据Cache 存取时，最高可以降低 75%的等待时间，因为进行查询处理时，只需扫描相关的数据列。相对于行存储， TPC-H 查询的执行时间可以提高11%～48%。此外，由于数据是首先进行横向划分的，PAX 支 持高度的扩展能力。

由此可见，行存储格式有利于执行(LTP 负载，列存储格式有利于执行分析型 (OLAP) 负 载 ，PAX 格式则兼顾了(OLTP 和OLAP 负载的性能。此外，为了支持混合负 载，人们把面向读操作优化 (Read-Optimized, 比如列存储)的存储结构和面向写操作优 化 (Write-Optimized, 比如行存储)的存储结构结合起来。已有的数据按照列存储格式保 存，支持快速存取和操作，称为主副本。新增或者修改的数据用行存储格式保存在缓冲区 (Write-Optimized Buffer) 中，称为Delta 数据。Delta 数据定期合并到列存储格式的主副 本中。

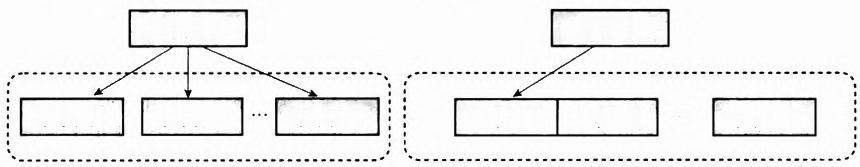
**2.** **索引技术**

当我们从数据库中查询数据时，如果有索引的帮助对数据进行定位，我们就可以避免 全表扫描 (Full Table Scan)。面向内存数据库的索引技术可以分成如下几类.分别是 Cache 敏感的索引 (Cache Conscious Index)、Cache不敏感的索引 (Cache Oblivious In- dex) 以及特殊索引，比如面向非易失性内存 (Non-Volatile Memory) 的索引等。



Cache敏感的索引根据内存层次 (Memory Hierarchy①) 的一些参数，比如内存级别 的数量、每个内存层次的块大小、各个内存层次之间的相对读写速度 (R/W Relative Speed) 等，设计紧凑的数据结构，尽量减少Cache Miss 。Cache不敏感的索引，则独立 于这些内存层次的参数进行设计，无论在什么样的硬件平台上运行，都获得可以接受 性能。

比如 Rao 和Ross 设计的Cache 敏感的 B+-tree 索引 “Cache Sensitive B+-tree”, 在 索引树中，每个节点的子节点连续存放.仅仅保存第一个子节点的开始地址(见图3-8)。 其他子节点可以通过偏移量进行定位。由于每个节点的所有子节点只需保存一个指针， Cache 敏感 B+-tree 索引提高了CPU Cache的利用率。



父节点

子节点2

子节点在内存中紧密存放(有利于减少CPU Cache Miss)

需要一个指针(减少内存占用)

子节点在内存中分散存放 需要若干指针

子节点n 子节点1

子节点1

子节点2

子节点n

父节点

.

**图3** **-** **8** **Cache敏感的B+ 树索引(示意图)**

**3.** **查询优化**

对于内存数据库来讲，查询优化的重点是如何利用多核CPU 以 及 众 核GPU, 实 现 关 键的数据操作的快速处理，比如选择、投影、连接、聚集等。

Kim 等充分利用CPU Cache、线程级并行 (Thread-Level Parallelism)、数据级并行 (Data-Level Parallelism)、内存的高带宽 (High Memory Bandwidth) 等硬件特点，设计 了新的连接操作算法，包括Hash Join 和Sort-Merge Join, 以充分发挥多核CPU 的性能。 经过优化，Hash Join能够达到100M 元组/秒 (Tuples/Second) 的性能，比其之前的最 好结果快17倍。 Sort Merge Join 则达到47M～80M 元组/秒的性能。Albutiu 等对 Sort Merge Join进行了重新设计，考虑到内存存取的非对称性 (None Uniform Memory Ac- cess,NUMA), 使得多核CPU 的每个CPU 在进行排序操作时，尽量存取隶属于该CPU 的局部内存 (Local Memory), 而不是隶属于其他CPU 的远程内存 (Remote Memory), 于是获得更高的处理速度。

Kaldewey 等把关系数据库的Join 操作迁移到 (Offloading)GPU 上运行，他们设计 的 Hash Join算法，充分利用PCI-E 带宽(6.3GB/s) 把数据传输到GPU 上，并且利用 GPU 的众核处理器，加速Join 操作的性能，获得比多核CPU 快 3X～8X的性能提升。 Karnagel 等则把分组和聚集操作 (Grouping/Aggregation) 迁移到 GPU 上运行。他们利 用启发式规则，在运行时 (Execution Time), 选择代价最优的算法和参数，提高分组和 聚集操作的速度。

**4.** **并发控制**

并发控制是在多个事务并发执行的情况下，保证数据库状态正确的技术手段。并发控 制有两大类算法，分别是基于加锁的并发控制和基于多版本管理的并发控制。根据深入的 剖析，人们发现加锁(以及给数据结构加闩锁)占事务处理30%的指令计数。当数据完全

① 关于各级存储器的带宽、延迟、容量等特点，以及数据在不同存储器层次之间如何交换，请参考相关资料。

**驻留内存，我们有多个** **CPU 核心可用时，设计合理的并发控制方法，才能有效发挥多核**

处理的威力。

**Larson等为内存数据库设计了两个基于多版本管理** **(Multi Versioning) 的并发控制** **方法，分别使用乐观的和悲观的策略来保证事务的原子性。通过多版本管理，事务管理器** **有效地隔离读事务** **(Read Transaction) 和写事务** **(Write Transaction), 读事务读取它应** **该读取的数据的最近已经提交的** **(Committed) 版本，写事务则对数据进行拷贝** **(Copy** a new Version), 生成新版本进行操作和读事务没有读写冲突。他们通过实验发现数据库的 大部分事务都是短事务，在事务对数据存取的竞争程度较低的情况下，加锁方式获得较好 的性能，但是在大量事务竞争存取热点数据以及负载里包含大量的长事务的情况下，基于 多版本管理的并发控制技术，能够获得更高的性能。需要注意的是，多版本管理需要付出 版本管理的开销。

Jones 等设计了两个轻量级的加锁 (Light Weight Locking,LWL) 并发控制方法。 其中一个并发控制方法无须跟踪事务对数据的读/写操作是否冲突，而是在提交事务时判 别读/写冲突，如果发现此类冲突，则需要回滚 (Roll Back) 某些事务。在这样的情况下， 就会浪费一部分已经做的工作。这个方法称为Speculative LWL 方法。这个方法在TPC-C 测试基准上，获得2倍的事务吞吐量。

**5.** **恢复技术**

当发生掉电情况. 内存里的数据即刻消失(除了非易失性内存)。对于内存数据库来 讲，恢复模块是保证其可靠性的重要模块。恢复模块的目标，是在系统失败以后尽快地把 内存数据库给恢复过来。为了能够进行数据库恢复，恢复模块在正常事务处理过程中需要 做一些簿记工作 (Book Keeping), 这些簿记工作不要过多地干扰正常的事务处理。

磁盘数据库的恢复技术已经很成熟和高效，整个恢复子系统包括写日志、写检查点、 恢复三个子模块。人们并没有从零开始设计和实现内存数据库的恢复模块，而是借鉴了磁 盘数据库恢复技术的思想。当然，为了提高内存数据库的恢复效率，人们提出了一些新的 方法 。

Lee 等提出了并行恢复 (Parallel Recovery) 技术，加快内存数据库的恢复过程。他 们的恢复技术基于差分日志技术 (Differential Logging)。差分日志通过把数据修改之前的 值和修改之后的值进行异或差分 (XOR Differencing) 操作求出来。差分日志符合结合律 和交换律 (Commutative/Associative), 因此，日志记录可以按照任意顺序进行排列，写 入磁盘，加快事务的执行。进行系统恢复时，对每个数据页面 (Page), 日志记录应用① 的顺序也无须符合串行顺序 (Serialized Order), 可以乱序应用，于是每个数据页面可以 独立地、并行地进行恢复，加快恢复过程。

Malviya 则舍弃了传统的物理日志 (Physiological Logging, 记录数据的改变)方法， 提出轻量的命令日志 (Command Logging) 方法。在事务执行时，系统记录事务对应的 SQL 命令以及参数。对数据库进行恢复时，首先把数据库的上一次一致检查点 (Consist- ent Checkpoint) 装载进来，然后重放 (Replaying, 即重新执行)日志里的所有命令即可。 由于数据全部在内存中，重放过程是很快的。在TPC-C 测试基准上，命令日志方法获得 比传统的物理日志方法高1 . 5X 倍的吞吐量。



①日志本身记录了事务对数据的修改，恢复过程中，日志记录应用 (Log Record Applying)的含义是把这些修 改施加到老版本的数据上，把数据修正到最新的正确状态。

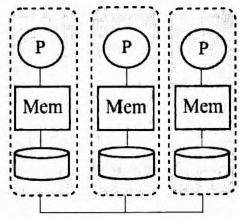
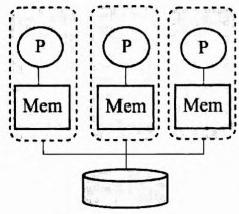


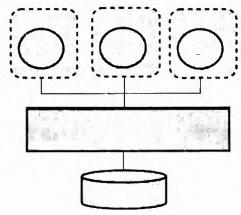
随着非易失性内存价格的下降，以及其读写性能的提高，人们在考虑如何在数据库的 恢复模块里应用非易失性内存。Schwalb 等提出了Hyrise-NV 数据库，该数据库在非易失 性内存里，利用多版本数据结构，直接修改数据和索引 (Data &.Index)。所有修改都是 事务性的，符合事务的ACID 约束。系统失败后，Hyrise-NV 几乎可以瞬间恢复整个数据 库系统，提供新的服务。它完成有10000000行记录的表格的恢复，仅仅需要100ms。

**3.2.4 MPP 并行数据库**

MPP 是 Massive Parallel Processing的缩写。 MPP 并行数据库利用专用的数据库集群 (Dedicated Cluster) 的多个节点的并行处理能力，提高数据库的查询处理性能。为了提高 并行数据库的性能，有时候MPP 数据库运行在专门的硬件上，比如厂家定制的数据库一 体机 (Database Machine) 。MPP并行数据库一般用于运行分析型负载 (Analytical Work- load) 。 分析式查询一般需要对大量数据进行扫描 (Scan) 、 连 接 (Join) 、 聚 集 (Aggrega- tion) 等操作。

在架构方面，MPP 并行数据库具有三种架构模式，分别是共享内存 (Shared Memo- ry) 架构、共享磁盘 (Shared Disk) 架构和无共享 (Shared Nothing) 架构(见图3-9)。





P P P

**memory**

(a)Shared Memory (b)Shared Disk (c)Shared Nothing (Shared Everything)

**图3** **-9** **MPP数据库架构**

注：P 表示CPU,Mem表示内存，圆柱体表示磁盘存储，虚线框表示处理单元(节点),处理单 元(节点)间的连线表示高速互联网络。

Shared Memory 架构，在单个主机上，通过多个处理器(或者多核技术)实现并行处 理。多个处理器(核心)共享主机的内存以及所有外部存储设备。每个处理器就是一个处 理单元或者一个节点 (Node), 处理器之间通过高速通信网络 (Interconnection Network) 互联。共享内存的系统，通过增加CPU 和内存，提高系统的处理能力，其扩展能力有限。

Shared Disk 架构指的是并行数据库的各个处理单元(节点)拥有自己的CPU 和 内 存，但是共享磁盘存储系统。各个节点的处理器之间没有任何直接的信息和数据的交换， 多个节点和磁盘存储由高速通信网络连接，每个节点都可以读写全部的磁盘存储。典型的 系统有Oracle 的 RAC 集群数据库系统。当系统的负载增加时，我们可以通过增加节点来 提高并行处理能力，但是由于所有节点共享外存系统，所以存储带宽有可能成为系统的瓶 颈，扩展能力有限。

Shared Nothing架构，各个处理单元(节点)拥有自己独立的CPU 、 内存和磁盘存

储，不存在共享资源，多个节点一起构成整个并行数据库系统。每个处理器使用自己的资

源，处理自己的数据，不存在内存和磁盘的争用。每个节点负责管理一部分数据，并执行 协调者 (Coordinator) 发送过来的全局事务 (Global Transaction) 的子事务 (Sub Trans- action) 的操作。典型的系统，如Oracle公司的 Exadata 数据库。无共享架构具有最好的 扩展能力。当我们为系统增加节点时，不仅提高了处理能力，而且提高了存储能力和I/O 总带宽。

为了实现数据的并行处理，需要对数据库数据进行划分，分布到各个节点。数据的划 分称为分片 (Partitioning), 分片主要有Range,Hash 等方法。Range 方法把数据库表的 记录，按照某个字段的值所属的范围进行分片。比如，当按照日期进行数据分片时，我们 可以把不同月份的数据划分到不同分片。Hash 方法则对于数据库表中的记录，根据一个 或几个属性列的取值，利用哈希 (Hash) 函数计算一个函数值，映射到具体的分区，完 成数据分片。

在 MPP 并行数据库中，数据分布在各个节点上，查询处理和集中式数据库有所不同。 我们通过实例，介绍简单的单表查询和多表连接查询。

这里的单表查询指的是对一个数据库表进行选择、分组和聚集的查询。比如查询：

|  |
| --- |
| **Select** year(sale\_date),product\_name,sum(price\*count) **From** sales  **Where** productid='101'  **Group By** year(sale\_date) |

注：从销售表中，选择产品101的销售记录，按照年度进行分组，计算销售额的汇总。

在 MPP 数据库系统里，一般有一个或者若干节点，接收用户的查询请求，并且担任 事务协调者 (Coordinator) 的身份。Coordinator 接收到用户的查询请求以后，根据数据 分布的元信息，把查询进行分析、改写和分解发送给相关的节点运行。比如对于上述单表 上的选择、分组、聚集查询，需要把该查询分发给包含相关数据的节点。

这些节点并行处理自己管理的数据，对数据进行扫描 (Scan) 、 过 滤 (Filter), 并且 对这些数据进行本地分组 (Group) 和聚集 (Aggregate), 然后把结果发送给Coordina- tor 。Coordinator 获得所有子事务的本地聚集结果以后，对结果进行合并，返回给用户即 可。在查询处理过程中，各个节点无须进行数据交换。

对于多表连接查询，我们以两个表的Hash Join作为例子。比如查询：

|  |
| --- |
| **Select** customer.name,sum(price\*count) **From** customer,sales  **Where** customer.id=sales.cid  **And** year(sale\_date)>=2015 and year(sale\_date)<=2016 **Group By** customer.id,year(sale\_date) |

注：对销售表和客户表进行连接查询，选择2015年和2016年的销售记录，按照客户和年度进行分组，汇总销售额。

对于该查询，当查询优化器选择使用Hash Join执行查询时，它把各个节点上用户表 的记录，按照customer.id 进行 Hash 操作，分发到查询节点(比如R1,R2,R3) 上，同 时把各个节点上销售记录表的记录，按照sales.cid(cid 是 sales 的外键，表示某个销售记 录隶属于某个用户，也就是把产品销售给了某个用户)进行Hash 操作，分发到各个查询 节点(比如R1,R2,R3) 上。

**由于上述** **Hash 操作的** **Hash 函数是一样的，并且** **customer.id 和** **sales.cid 的语义都** 是**指某个用户，那么符合连接条件的用户记录和销售记录，被** **Hash 分配到同样的查询节** **点。这些查询节点对收到的用户记录和销售记录进行连接操作以及后续的分组和聚集操** **作，然后把局部聚集结果返回Coordinator, 由其进行结果合并返回用户。由此可见，在** **Hash Join 的处理过程中，涉及了节点间大量的数据交换。**

**有时候，基于数据的分布特点和查询条件，优化器可以明智地对数据进行交换。比如** **对于上述实例，如果优化器了解到销售记录是按照sales.cid 进行Hash 划分的，那么执行** **查询时只需对** **customer 表根据** **customer.id 进行Hash 操作，把客户表的记录按照客户** **ID,Hash 分布到各个节点，就能保证每个节点对正确的数据进行连接操作。**

如果销售表和客户表分别是按照sales.cid和customer.id实现进行 Hash 分布的，也 就是两个将要连接的表格已经在连接的 Key上进行 Hash 划分，那么进行查询处理时无须 对数据再进行 Hash 分布，直接在每个节点上执行本地连接 (Local Join) 即可，这种情况 称为数据已经预先并置 (Co-Location) 在一起了。更为一般地，数据并置指的是其中一个 表的分片Key(Partitioning Key) 是一个外键，比如上述实例中的sales.cid, 另外一个表 的分片 Key 包含了主键，比如上述实例中的 customer.id, 而且这两个表的分片数量 (Partitions) 是一样的。

在这种情况下，主外键连接，比如上述事例中的customer.id=sales.cid, 可以在各个 节点上通过执行本地连接实现。因为匹配 (customer.id=sales.cid) 的分区已经事先保存 在各个节点上了，连接过程中无须通过网络交换数据。

在上述实例中，如果 sales.cid 上建立了聚簇索引 (Clustered Index),那么sales 表的 销售记录是按照cid 进行排序保存的。这时，我们可以在 customer 表和 sales 表之间进行 合并连接 (Merge Join)。因为当我们扫描 customer 表和 sales 表时，每在 customer 表上 扫描到一个客户，就可以在sales表上寻访到该客户的所有销售记录(因为sales的销售记 录是按照cid 进行排序的)。

**1.** **分布式事务处理**

在并行数据库中，数据分布在一系列节点上，事务处理的难度提高了。以转账事务为 例，在集中式数据库中，只有一个物理节点，它了解事务的所有状态，通过加锁机制或者 多版本并发控制机制，就能够保证事务的ACID 特性，保证数据的一致性。但是在并行数 据库里，各个节点之间在物理上相互独立，通过网络进行沟通和协调。相互独立的节点之 间，无法准确地知道其他节点中的事务执行情况。如果想让分布式部署的多台机器中的数 据保持一致性，就要保证在所有节点的数据写操作，要么全部执行，要么全部不执行。但 是，一台机器在执行本地事务时，无法知道其他机器中的本地事务的执行结果，因此，这 个节点也就不知道本次事务到底应该提交 (Commit) 还是回滚 (Rollback) 。 常规的解决 办法就是引入一个事务协调者(也称事务管理器)来统一调度各个节点的执行。

涉及多个节点的事务称为分布式事务。对涉及多个节点(数据)的分布式事务进行协 调的协议有两阶段提交协议(2 Phase Commit)、三阶段提交协议(3 Phase Commit) 等 ， 下面对这些协议进行简单介绍并分析其优点和缺点。

**2.** **两阶段提交协议**

在并行数据库里，当一个事务涉及多个节点上数据的修改，其中一个节点被指定为事 务协调者，其他节点则称为事务参与者。

两阶段提交协议的核心思想为，参与者将操作成败通知协调者，再由协调者根据所有

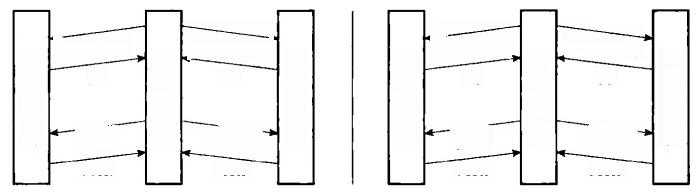
参与者的反馈情况，决定各参与者是提交操作还是中止操作。

两阶段提交协议包括两个阶段，分别是准备阶段(或投票阶段)和提交阶段(或执行 阶段)。

(1)准备阶段。事务协调者给每个参与者发送Prepare 消息.每个参与者要么直接返 回失败(如权限验证不通过).要么在本地执行事务，写本地的Redo 和 Undo 日志，但不 提交，处于一种万事俱备，只欠东风的状态。可以将准备阶段划分成三个步骤：①协调者 向所有参与者询问是否可以执行提交操作，并等待各参与者的响应。②参与者执行事务操 作，并将Undo 信息和Redo 信息写人日志。③各参与者响应协调者发起的询问。如果参 与者的事务操作实际执行成功，则返回一个“同意”消息给协调者；如果参与者的事务操 作执行失败，则返回一个“中止”消息给协调者。

(2)提交阶段。如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送 回 滚 (Rollback) 消息.否则，协调者给所有参与者发送提交 (Commit) 消息。参与者根据 协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的资源(锁资源等)。

图 3 - 1 0 (a) 和图3 - 10 (b) 分别展示了所有参与者的响应消息都为“同意”时的提 交过程或者有参与者响应消息为“中止”时的回滚过程。



请求提交 是

提交

Ack

请求提交 是

提交

Ack

请求提交 否

回滚

请求提交 是

回滚

Ack · Ack

事务参与者

事务参与者

事务参与者

事务协调者

事务参与者

事务协调者

(a) (b)

**图3-10** **两阶段提交协议**

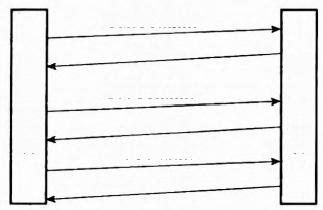
两阶段提交协议，看起来好像能够提供事务的 ACII) 特性。两阶段提交协议具有内在 的局限性：(1)同步阻塞。执行过程中，所有参与者都是事务阻塞的。也就是说，当参与 者占有资源时，其他第三方程序访问这些资源将处于阻塞等待状态。(2)单点故障。协调 者地位非常重要， 一旦协调者发生故障，参与者会一直阻塞下去，尤其在第二阶段，协调 者发生故障，那么所有的参与者还都处于锁定资源的状态中，将无法继续完成事务操作。 虽然我们可以在协调者宕机后重新选举一个协调者.但是无法解决因为协调者宕机导致的 参与者处于阻塞状态的问题。(3)数据不一致。在两阶段提交的第二个阶段中，当协调者 向参与者发送Commit 消息之后发生了局部网络异常，或者在发送 Commit 消息过程中协 调者发生了故障，将导致只有一部分参与者接收到 Commit 消息。这部分参与者接到 Commit 消息之后，就会执行Commit 操作，但是其他未接到 Commit 消息的机器则无法 执行事务提交，在这种情况下，整个系统便出现了数据不一致的状况。(4)两阶段提交协 议有一个无法解决的问题：协调者在发出Commit 消息之后宕机，而且唯一接收到这条消 息的参与者同时也宕机了，那么即使通过选举协议产生了新的协调者、这个事务的状态也 是不确定的，没人知道事务是否已经被提交。

**3.** **三阶段提交协议**

由于两阶段提交协议存在同步阻塞、单点故障、数据不一致等问题，研究人员在两阶 段提交协议的基础上提出了三阶段提交协议。三阶段提交协议是两阶段提交协议的改进版



(见图3- 11)。



CanCommit

*Yes*

PreCommit

Ack

DoCommit

Committed

事务协调者

事务参与者

**图3-** **11** **三阶段提交协议**

三阶段提交协议有两个重要改动：(1)引入超时机制，三阶段提交协议同时在协调者 和参与者中都引入超时机制。(2)三阶段提交协议把两阶段提交协议的准备阶段一分为 二，三阶段提交协议总共有三个阶段，分别是CanCommit,PreCommit,DoCommit 三个 阶段。

当事务处理进入第三阶段时，说明参与者在第二阶段已经收到了 PreCommit 请求。 协调者产生 PreCommit 消息的前提条件，是它在第二阶段开始之前，收到所有参与者的 CanCommit 响应都是Yes 。一旦参与者收到了PreCommit, 意味着其他参与者都同意对数 据进行修改。用一句话来概括，当进入第三阶段时，由于网络超时等原因，虽然参与者没 有收到Commit 或 者Abort 消息，但是它有理由相信，成功提交的概率是较大的。参与者 无法及时收到来自协调者的信息，它会默认执行Commit。

相对于两阶段提交协议，三阶段提交协议解决了单点故障问题、减少了堵塞(因为一 旦参与者无法及时收到来自协调者的信息，它会默认执行Commit, 而不会一直持有事务 资源并处于阻塞状态),但这种机制还是会导致数据一致性问题。比如，由于网络原因， 协调者发送的Abort 消息没有及时被参与者接收到，那么参与者在等待超时之后执行了 Commit 操作 · 这样就和其他接到Abort 消息并执行回滚的参与者之间存在数据不一致的 情况。

由此可见，无论是两阶段提交协议，还是三阶段提交协议，都无法彻底解决分布式系 统的一致性问题。Google 公 司 的Chubby 分布式锁服务系统的作者Mike Burrows 说过， “There is only one consensus protocol,and that's Paxos—all other approaches are just broken versions of Paxos.”。他的意思是，只有一种一致性算法即Paxos, 其他算法都是 Paxos 算法的不完整版。由此可见， Paxos 算法才是最终的解决方案。

**4.Paxos 算法**

在 Paxos 算法中，总共有四种角色，分别是提议者 (Proposer) 、 决策者 (Acceptor)、

产生议题者 (Client) 、 最终决策学习者 (Learner) 。Proposer 可以提出 (propose) 提案， Acceptor 可以接受 (accept) 提案。在分布式系统中，达到数据的一致性，就是对某个数 据的取值达成一致，比如对某个账户应该取什么值达成一致，也就是各个节点达成共识， 或者说，Proposer,Acceptor.Learner 等都认为同一个value 被选定 (chosen)。

Proposer,Acceptor,Learner 分别在什么情况下才能认为某个value 被选定呢?为了 对某个value 的取值达成共识，保证数据的一致性，Paxos 算法在执行过程中必须符合一 系列约束条件：(1) Proposer 。 只 要Proposer 发的提案被Acceptor 接受(需要半数以上 的 Acceptor 同意才行),Proposer 就认为该提案里的 value 被选定了。(2) Acceptor 。 只



要 Acceptor 接受了某个提案，Acceptor 就认为该提案里的value 被选定了。(3) Learner。 Acceptor 告诉Learner 哪个value 被选定，Learner 就认为那个 value 被选定。Learner 的 地位具有某种从属性。

Paxos 算法的执行过程分为两个阶段。

(1)第一阶段：①Proposer 选择一个提案编号N, 然后向半数以上的 Acceptor 发送 编号为N 的 Prepare 请求。②如果一个Acceptor 收到一个编号为N 的 Prepare 消息，而且 N 大于该Acceptor已经响应过的所有Prepare请求的编号，那么它就会将已经接受过的编 号最大的提案(如果有的话)作为响应反馈给 Proposer, 同时该Acceptor 承诺不再接受 任何编号小于N 的提案。

(2)第二阶段：①如果Proposer 收到半数以上Acceptor 对其发出的编号为N 的 Pre- pare 请求的响应，那么它就会发送一个针对 [N,V] 提案的Accept 消息给半数以上的 Acceptor 。注意：V 就是收到的响应中编号最大的提案的value, 如果响应中不包含任何 提案，那么V 就由Proposer 自己决定。②如果Acceptor 收到一个针对编号为N 的提案的 Accept 消息，只要该Acceptor 没有对编号大于N 的 Prepare 请求做出过响应，它就接受 该提案。

Paxos 算法执行过程见图3-12。

|  |  |  |
| --- | --- | --- |
|  | Proposer(N,V) | Acceptor(ResN,AcceptN,AcceptV) |
| 第一阶段 | Prepare(N),N递增，不重复  Prepare请求 | 如果N<=ResN.不响应或者响应error 如果N>ResN,则(a)令ResN=N,(b)  响应(Pok,AcceptN,AcceptV)或者 (Pok,Null,Null) |
| 第二阶段 | 如果收到超过半数的Pok,发出Accept(N,V) 如果响应中有提案，则V=响应中最大的Ac- ceptN对应的AcceptV;如果响应都是(Pok, Null,Null),那么V=Proposer自己定义  的值  如果Pok未超过半数，重新获取N,发起 Prepare请求(回到第一阶段) | response |
| accept请求 | 如果N>ResN,接受提案，令AcceptN= N.AcceptV=V,回复<Aok>  如果N<=ResN,不接受。不回复或者回 复error |
| 如果<Aok>数过半，确定V被选定  如果<Aok>数不过半，重新发起Prepare 请求 | response |

**图3-12** **Paxos算法执行过程示意图**

Paxos 算法是如何在分布式系统中保证数据一致性的，其证明请参考相关资料。①



①<http://www.ux.uis.no/>～meling/papers/2013-paxostutorial-opodis.pdf

**3.3** **结构化数据分析工具介绍**

结构化数据分析工具分为两大类，分别是传统的MPP 数据库和列存储数据库，以及 基于Hadoop 的结构化数据分析工具，即 SQL on Hadoop 系统。①前者包括MPP 数据库 比如Teradata 、 列存储数据库比如 Vectorwise, 后者包括 Hive,SparkSQL,Impala,

Presto 等系统。

SQL on Hadoop系统依赖于 HDFS(Hadoop Distributed File System) 作为数据存储 层。HDFS 具有高度的扩展性和容错性(通过数据的多副本存放),于是SQL on Hadoop 系统在存储层面支持超大规模数据的处理。在大数据时代，如何查询和分析TB 级别甚至 PB级别的数据是数据分析工程师不可回避的问题， SQL on Hadoop系统成为一个重要的 工具。把SQL 移植到 Hadoop 平台上的原因，是利用Hadoop 的扩展能力管理大规模的数 据，让用户能够使用他们熟悉的SQL 语言对数据进行查询和分析。

此外，对于传统的MPP 并行数据库来讲，它的容错保证需要依赖各个厂家专有的技 术，甚至需要添加特殊的硬件。MPP 并行数据库需要运行在特定的硬件平台上，各个厂 家的软件也不是免费的。SQL on Hadoop 系统则可以运行在廉价服务器构成的集群上，软 件几乎是免费的。对 MPP 并行数据库的管理，需要管理员具有该数据库的管理经验。随 着 Hadoop 技术的流行，人们可以通过各种途径获得管理 Hadoop 平台的经验，管理成本 相对降低 。

**3.3.1 MPP(Shared Nothing) 数据库、基于列存储的关系数据库**

**1.Teradata**

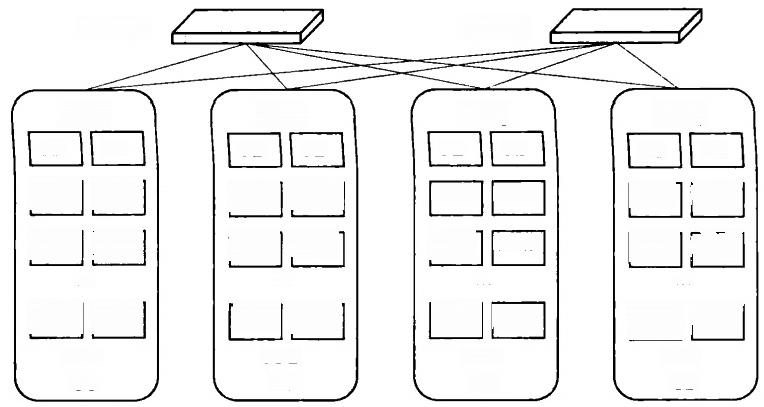
Teradata 数据库是面向大型数据仓库应用的基于Shared Nothing 架构的并行数据库系 统 。Teradata 系统由三部分构成，分别是处理节点 (Node) 、 实现节点间通信的高速互联 网络 (Interconnection) 、 数据存储介质(一般是磁盘阵列)。图3-13展示了由4个节点 构成的Teradata 数据库系统。每个节点自底向上，包括操作系统软件 (OS) 、Teradata 并 行数据库扩展 (PDE) 和其他相关模块，包括PE 和 AMP。

(1)(S 与 PDE 。Teradata 并行数据库扩展 (Parallel Database Extensions,PDE), 是直接架构在操作系统之上的一个接口层，用于为Teradata 提供并行环境，并保证这个 并行环境的可运行性和可靠性。PDE 的主要功能是管理和运行虚拟处理器、进行Teradata 并行任务调度、进行操作系统内核和Teradata 数据库的运行时故障处理等。

(2)AMP 。 存取模块处理器 (Access Module Processor,AMP), 是 Teradata 数据库 的关键进程之一，用于处理所有与数据有关的文件系统的操作任务。 一般情况下，一个节 点上有多个 AMP 在工作，每个AMP 分别负责文件系统上不同的数据的存取操作。

(3)PE 。 解析引擎 (Parsing Engine,PE), 用于实现客户端(通常是使用Teradata 数据库的应用程序的 SQL 请求)和存取模块处理器 (AMP) 之间的通信和交互。主要的

① 关 于Hadoop的详细内容，在第12章 “Hadoop 及其生态系统”介绍。



BYNET。

Node。

PE PE

AMP AMP

AMP AMP

AMP AMP

PDE

OS

BYNET

Node₂

|  |  |
| --- | --- |
| PE  AMP  AMP  AMP | PE  AMP  AMP  AMP |

PDE OS

Node,

PE PE

AMP AMP

AMP AMP

.

AMP AMP PDE

OS

Node,

PE PE

AMP AMP

AMP AMP

AMP AMP

PDE

OS

**图3-** **13** **Teradata 数据库整体架构**

功能包括会话控制 (Session Control),SQL 语句的解析、优化 . 查询步骤的生成和分发， 并行查询处理和返回查询结果。 一个节点上通常只有一两个 PE 在工作。

(4)BYNET。 在 Teradata MPP 系统中，各个节点间(即各个 AMP 之 间 ) 的 内 部 高 速互联是通过 BYNET 实现的，它由一组硬件和运行在这组硬件上的处理通信任务的软件 进程构成。 BYNET 用于节点之间的双向广播 (Bidirectional Broadcast)、多路广播 (Mul- ticast) 和点对点 (Point-to-Point) 通信。

除 了Teradata 数据库，采用Shared Nothing 架构模式的并行数据库系统还有 Exadata 数 据 库 ( 来 自(racle) 、XtremeData 数据库等。

**2.SAP HANA**

SAP HANA 是支持(LTP 和 (LAP 混合负载的内存数据库系统。由于采用内存计算

技 术 ，HANA 能够获得极快的事务响应时间和极高的事务吞吐能力。

HANA 采用列存储技术进行数据的组织，同时对列存储数据进行有效的压缩。对数 据进行压缩可以有效减少数据占用的空间。在进行查询处理时，即便考虑到解压缩的代价 也是值得的，因为CPU 比内存总线 (Memory Bus) 的速度要快很多，所以 CPU 可以及 时地对数据进行解压缩。HANA 支持数据的更新功能，其更新功能是通过添加数据实现 的。删除数据时，旧的数据(记录)没有被覆盖掉，而是在表格的末尾添加 (Append)。 当一个查询执行时，它可以看到一个记录的若干版本.选取最近的提交版本 (Committed Version) 返回即可。

通过利用大内存以及列存储和数据压缩技术，HANA 把整个数据集保存在内存中， 实现了实时的查询处理。利用HANA 的快速查询处理能力，人们可以获得秒级的响应时 间(小于10秒)。性能评测结果显示，在相同的硬件环境和数据规模下， HANA 往往能 够获得超过(Oracle(磁盘数据库)一个数量级左右的查询性能提升。①运行在 HANA 之 上的报表系统，几乎无须用户等待，返回各种统计结果，给用户更好的体验。

HANA 数据库产品是 一个完备的工具套件，包括内存数据库服务器 (MMDB Serv-

① <http://sapinsider.wispubs.con/Assets/Blogs/2013/September/Real-HANA-Perfornance-Test-Benchmarks.>



er) 、建模工具 (Studio) 以及客户端工具 (JDBC,ODBC 接口等)。在一个数据库服务器 上，HANA 支持文本 (Text) 、 空间数据 (Spatial) 、 图 (Graph) 、 流数据 (Streaming) 以及时间序列 (Time Series) 数据的处理能力，并且提供实时事务响应时间。通过集成R 软件包，HANA 方便地在数据上运行数据挖掘和机器学习算法，提供预测性分析能力 (Predictive Analysis)。

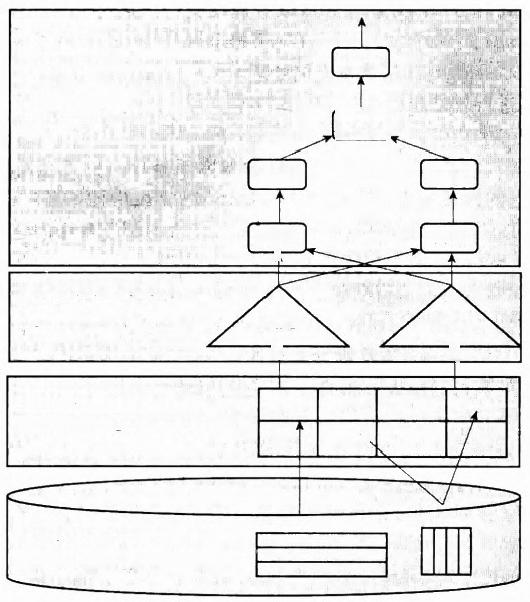
**3.MonetDB/Vectorwise/VectorH**

MonetDB 是荷兰研究机构CWI(Centrum Wiskunde &.Informatica) 于2003年开始 研发的基于列存储的面向分析型应用的内存数据库系统。2008年MonetDB 开发者基于 MonetDB创立了Vectorwise 公司，实现MonetDB 的商业化。经过两年的研发，产品趋于 成熟，并且性能优越， Vectorwise 被 Actian 公司收购。

MonetDB 和 Vectorwise 的主要区别在于其查询处理模式， MonetDB 的查询处理模式 是 Column at a time, 即一 次处理一个数据列， Vectorwise 则 是Vector at a time, 即 一 次 处理一个向量(后面详细叙述)。 一 次处理一个数据列，需要对中间结果进行物化，交给 后续的查询处理步骤继续处理。 一次处理一个向量，则有利于把若干处理步骤整合起来， 数据尽量地驻留在CPU Cache中，提高处理的效率。

为了尽快实现市场化， Vectorwise 把 MonetDB 的 X100 存储引擎、查询引擎 (Vec- torized Query Engine) 和 Ingres 数据库的上层模块结合起来。 Vectorwise 通过基于位置 的 Delta 树 (Positional Delta Tree,PDT), 维护数据的更新，支持数据的更新功能。进

行查询处理时，首先在只读的主数据集上提取相关数据，然后根据PDT 进行结果修正。 图 3 - 1 4 展 示 了Vectorwise 的查询执行过程涉及的PDT 数据结构、缓冲区管理以及数据 的存储等。



Parallel

Vectorized Query

Execution

XCHG

Aggr

Scan 不

Fast Transactional Updates

4

Scan Optimized

Buffer manager and I/O

Compressed DSM/NSM Storage

Aggr

Aggr

Scan

PDT

PDT

**图3-** **14** **Vectorwise 系统架构**

数据以列存储格式进行存放，并且保存在内存中。这时候 CPU 对内存的存取成为性 能瓶颈 (Bottlenecks) 。Vectorwise 针对这个瓶颈进行了一系列的优化，其中最重要的优 化是向量化的查询处理模式 (Vectorized Query Processing)。

向量化的查询处理模式是每个属性列的每100～1000个值构成一个向量，执行查询 时，一系列的向量以流水线的方式，流过对该属性列进行操作的一系列操作符，由其处 理，直到最后完成查询。换句话说，对某个属性列进行操作的一系列操作符，一次处理一 个向量而不是整个列。这种查询处理模式可以充分利用现代 CPU 的 SIMD 指令 (Single Instruction Multiple Data, 单指令多数据),通过并行操作，快速完成数据的处理。同时， 这种查询处理方式下已经进入CPU Cache的向量尽量长时间地驻留，以便对该向量的所 有操作符(构成操作符流水线)得到执行。此外，相关的数据属性列没有被存取，需要从 内存中调人CPU Cache的数据量减少了，于是CPU 的数据 Cache 和指令Cache 的命中率 都得到了提高。

下面以一个简单查询实例，解释向量化执行模式的查询处理过程。查询：

|  |
| --- |
| **Select Sum** (price\*quantity) **From** sales  **Where** sales.date='2011-02-02' |

在向量化执行模式下，查询处理器按照向量方式扫描日期 (date) 字段，然后获得一 系列符合查询条件 sales.date='2011-02-02 '的记录的偏移量。查询处理器继而提取对应 的单价 (price) 字段和数量 (quantity) 字段的向量，根据这些偏移量，对单价 (price) 字段取值和数量 (quantity) 字段取值进行乘法操作，计算并累加总和。

数据库的查询处理引擎的处理模式可以分成四类，分别是一次处理一个元组 (Tuple- at-a-Time Processing)、一次处理一块 (Block-Oriented Processing)、一次处理一列 (Col- umn-at-a-Time Processing) 和向量化查询处理 (Vectorized Query Processing)。下面对这 四种查询处理模式进行总结和比较。

(1)一次处理一个元组。

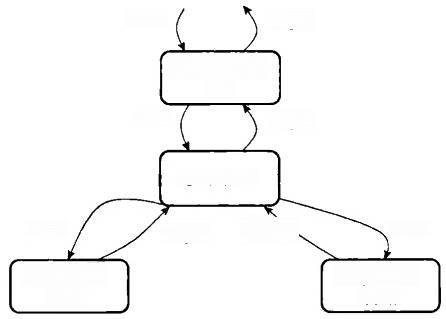
这种查询处理模式最早出现在 Volcano 数据库系统中，因此也称为 Volcano 风格的 (Volcano-Style) 查询处理模式。该模式是在RDBMS 系统中使用最广泛的查询处理模式， 包括 MySQL,SQLite,PostgreSQL,Oracle 等系统都使用这种查询处理模式。

客户端提交的SQL 查询，由查询解析器和优化器进行翻译.转换成查询执行计划 (Execution Plan), 交给查询执行引擎。查询执行计划的每个操作符 (Physical Operator) 有三个接口，分别是open(),next(),close() 。open 方法初始化资源包括初始化内存 数据结构或者打开文件，close 方法则释放资源。

整个执行计划的执行是由根节点的 next 方法拉动的，父节点的next 方法，从子节点 提取一个元组，这个元组可能是由子节点调用其子节点的next 方法获得的，直到对原始 数据库表的扫描节点 (Scan 操作符)。每个操作符的next 方法递归地调用子节点的 next 方法，最后输出一个元组，这就是 Tuple-at-a-Time Processing 查询处理模式的由来。比 如，图3-15展示了这样的一个查询执行计划，该查询扫描两个表，然后进行Hash 连接， 最后进行分组聚集。

Tuple-at-a-Time Processing查询处理模式，一般使用行存储格式。查询执行引擎装载 了不必要的字段信息，对于仅存取少量字段的分析型应用来讲，导致CPU Cache 命中率





Next Tuple

Hash Group

By

Next Tuple

Hash Join

Next Tuple Tuple Next

**Sequential Sequential**

Scan Scan

**图3-15** **一次处理一个元组**

较低。另外，整个查询语法树上的各个物理操作符交替执行， 一会儿执行Scan, 一会儿执 行Join 。 这些指令有可能无法全部放入CPU 的 一 级 (Layer One,L1) 指 令Cache, 导 致 频繁的Cache Miss, 指 令Cache 的命中率不高。

(2)一次处理一块。

Block-Oriented Processing查询处理模式是对 Tuple-at-a-Time Processing查询处理模 式的扩展。对next 函数的调用，将返回一批元组(100～1000个)而不是一个元组。这 种处理模式减少了函数调用的开销，获得比 Tuple-at-a-Time Processing 处理模式更高的 CPU 数 据Cache 和指令Cache 的命中率，但是在分析型应用中，存取每条记录不必要的 字段，仍然导致频繁的CPU 数据Cache Miss。

(3)一次处理一列。

Column-at-a-Time Processing 查询处理模式是最初实现的查询计划列式执行 (Colum- nar Execution) 模式。该查询处理模式使用列存储格式，每个数据列单独进行存放(紧密 排列的数组， Dense-Packed Array) 和后续处理。每个操作符的next 方法，处理整张表格 的某一列，并且把中间结果保存起来。为了支持这种处理模式，需要定义和实现若干操作 原 语 (Primitive) 。 这些操作原语在一个紧凑的循环中 (Tight Loop) 对数组的各个元素 进行处理。 一般来讲，对这些循环处理进行翻译后的指令，容易装入CPU 的 L1 级指令 Cache, 提高指令Cache 的命中率。此外，操作原语对数组进行操作，生成一个新的数组， 具有良好的数据Cache 的局部性，即要操作的数据可能已经在数据Cache 中。

智能编译器对 SQL 查询进行翻译时，能够利用自动向量化技术 (Auto-Vectoriza-

tion), 通过SIMD(Single Instruction Multiple Data) 指令实现数据处理。通过CPU 数据 级的并行处理能力，提高数据处理效率。

但是Column-at-a-Time Processing 查询处理模式也有局限性，因为它对一个表格 (a Whole Table) 的整个数据列进行操作，于是中间结果集可能很大，需要物化 (Material- ized) 到内存或者磁盘中，于是造成CPU 的数据Cache Miss以 及I/O 开销。

(4)向量化查询处理。

Vectorized Query Processing 查询处理模式前文已经简单介绍过，它是最先进的列式执行 模式，解决了Column-at-a-Time Processing 查询处理模式的缺点。这种处理模式建立在Col- umn-at-a-Time Processing模式之上，只不过在每个next 函数调用中，操作符处理的是某列数 据的一个部分 (Chunk), 称为一个向量，而不是表格的(某个)整个列。向量的大小设定，

考 虑 了CPU Cache的 大 小 ， 很 容 易 装 载 到CPU Cache中 ， 提 高Cache 命 中 率 。

正如上文介绍过的，这种处理模式可以充分利用CPU 的 SIMD 指令进行数据处理， CPU 的数据Cache 和指令Cache 的命中率都得到了提高。

此 外 ， 由 于 各 个 操 作 原 语 的 原 理 很 简 单 ， 我 们 可 以 使 用C 语 言 或 者 汇 编 语 言 实 现 这 些 原语 ， 使得编译器能够产生高效的执行代码 。

**4.Vectorwise** **MPP** **原** **型** **与** **VectorH**

在Vectorwise数据库基础上，Actian公司研发了MPP 并行数据库原型，这是一个采 用Shared Nothing 架构的并行数据库系统。在这个原型中，除了各个节点的 Vectorwise 进程外，为了把各个Vectorwise 进程连接起来，增加了基于 MPI(Message Passing In- terface) 通信机制的查询处理模块、在各个节点间进行数据交换的Exchange 操作符以及 查 询 改 写 模 块 (Parallel Rewriter), 如 图 3 - 1 6 所 示 。

**Client App/BI Tools**

SQLQuery result

SQL Parser

Parsed tree

INGRES

Optimizer

Query plan

Leader Node (Name Node)

Cross Compiler

Distributed rewriter上

Annotated query tree

Builder

Operator tree

X100

Execution Engine

Data request data

Buffer Manager I/O

HDFS

**MPI**

Worker Node(Data Node([1,n]

Annotated query tree

MPI

Partial resultset

rewriter

Annotated query tree

Builder

Partial operator tree

001X

Execution Engine

Data request data Buffer Manager

I/O

HDFS

**图** **3** **-** **1** **6** **VectorH 系统架构**

基于这个 MPP 数据库原型，Actian 公司继续把Vectorwise 迁移到 Hadoop 平台上， 利用HDFS 作为存储层，研发了SQL on Hadoop系统VectorH 。VectorH 扩展了Vector- wise MPP并 行 数 据 库 原 型 系 统 ， 利 用 Hadoop 平 台 的YARN 资 源 管 理 器 ， 实 现 工 作 负 载 和资源调度 。

依 赖 于 HDFS,VectorH 自 然 地 获 得 了 存 储 层 的 扩 展 性 和 容 错 性 能 。 此 外 ， ( 1 ) Vec-

torH 对 HDFS 的块复制策略 (HDFS Replication Policy) 进行了干预，控制数据块如何 在各个节点间进行复制和放置，目的是优化读操作的局部性 (Read Locality) 。(2) 虽 然 HDFS 只能进行数据添加 (Append), 但是 VectorH 通过基于位置的Delta 树 (Positional

Delta Trees.PDT) 实现了数据更新 (Update) 功 能 。PDT 是一个对数据更新进行追踪

的差分数据结构 (Differential Update Structure), 它支持快速的查询功能。

基于列存储、面向分析型应用的数据库还有Netezza,Vertica,Sybase IQ.ParAccel 等。

**3.3.2 SQL on Hadoop 系** **统**

Hive on MapReduce 和 Hive on Tez 是 Hadoop 平台上的结构化大数据分析工具， SparkSQL 则是Spark 平台上的结构化大数据分析工具。对这两个工具的具体介绍，请分 别参考关于 Hadoop 平台和 Spark 平台的相关章节。在这里，我们介绍 Impala 和 Presto 两个SQL on Hadoop 系统。

**1.Impala**

Impala 是 Cloudera 公司开发的大数据实时查询系统。它提供SQL 查询接口，能够查 询存储在 Hadoop 的 HDFS 和 HBase 中的 PB 级大数据。 Hadoop 平台上的数据仓库系统 Hive 也提供 SQL 查询支持(通过类似于SQL 的 HQL 语言) .但是由于Hive 查询需要转 换成MapReduce Job, 由 MapReduce 引擎执行.是一个批处理过程，不能满足交互式查 询的性能要求。 Impala 则试图填补这个空白，提供快速的数据查询响应时间，在 Hadoop 上实现交互式查询能力。

Impala 的实现借鉴了Google 公司的Dremel 系统。Dremel 是 Google 公司开发的交互 式数据分析系统，它构建于Google 的 GFS(Google File System) 文件系统之上，支撑了 数据分析服务Big Query 等诸多服务。 Dremel 的主要技术特色包括：(1)实现了嵌套的列 存储数据结构。(2)使用多层查询树，使得任务可以在上千节点上并行执行，不断对结果 进行聚集。其中， Dremel 的多层查询树的根节点负责接收查询，并将查询分发到下一层 节点……底层节点负责具体的数据读取和查询执行，然后将结果返回上层节点。中间层次 可能有多层，逐层进行数据汇总，最后由根节点把结果返回给客户端，如图3-17的左侧 子图所示。每个节点内部有一棵查询执行树 (Query Execution Tree), 是整个查询执行计 划的一部分，如图3-17的右侧子图所示。

Client Query Execution Tree



Root server

Intermediate

servers

Leaf servers

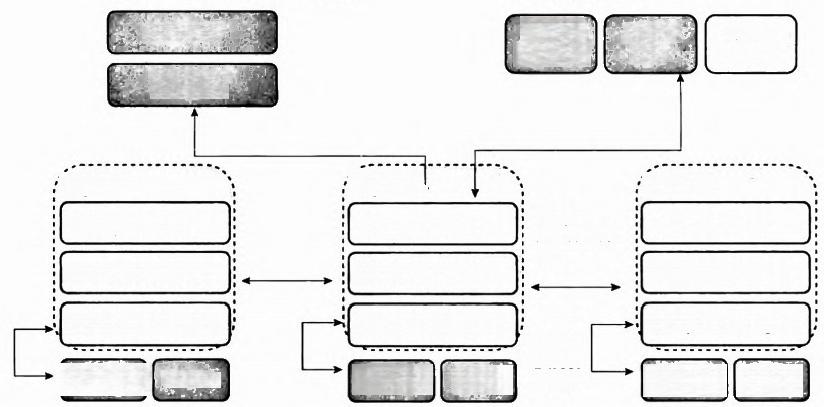
(with local storage)

Storage Layer(e.g.GFS)

图3- 17 Dremel的多层查询树

Impala 的系统架构如图3- 18所示。 Impala 支持主流的SQL 语言功能，包括Select, Insert,Join 等操作。表格的元数据信息存储在Hive 的 MetaStore 中 。State Store是 Im- pala 的一个服务，用来监控集群中各个节点的健康状况，提供节点注册、错误检测等功 能 。Impala 在每个节点运行了一个后台服务进程 Impalad, 用来响应外部请求，并完成实 际的查询处理。

Common Hive SQL&Interface Unified meta data



**Hive HDFS State**

**metastor NN Store**

|  |  |
| --- | --- |
| Fully  distributed MPP  Local Direct Read | impalad  Query Planner  Query Coordinator  Query Exec Engine  HDESDN **HBase** |

**SQLApp**

**ODBC**

impalad

Query Planner

Query Coordinator

Query Exec Engine

HDFSDN HBase

impalad↓

Query Planner

Query Coordinator

Query Exec Engine

HDESDN HBase

**图3-18** **Impala 系统架构**

Impalad 包含 Query Planner,Query Coordinator 和 Query Exec Engine 三个模块。 Query Planner接收来自客户端的SQL 查询请求，然后将其查询转换为许多子查询。 Que- ry Coordinator将这些子查询分发到各个节点上，由各个节点上的Query Exec Engine负 责子查询的执行，最后返回子查询的结果。这些中间结果经过合并之后，最终返回给用 户。查询处理过程中，中间结果集不写入磁盘，直接在不同的操作符和节点之间通过流水 线 (Pipeline) 方式进行传递。

Impala 的查询效率相比Hive 有数量级的提升。之所以能有这么好的性能，主要有如 下几方面的原因：(1) Impala 不需要把中间结果写入磁盘，节省了大量的I/O 开销。(2) Impala 像 Dremel 一样，借鉴了MPP 并行数据库查询处理技术的思想，抛弃MapReduce 计算模型，对查询进行深入的优化，省略不必要的 Shuffle,Sort 等操作。(3) MapRe- duce 启动任务 (task) 的速度是很慢的(默认每个心跳间隔是3秒钟),Impala 直接通过 服务进程来进行任务调度，节省了MapReduce 任 务 (task) 启动的开销，查询处理速度快 了很多。(4) Impala 使 用LLVM(Low Level Virtual Machine) 来编译运行时代码，提高 了查询计划的执行效率。(5) Impala 使 用C++ 语言实现，针对硬件进行了针对性的优 化，包括使用SSE 指令集提高数据处理效率等。(6) Impala 基于数据的局部性 (Data Lo- cality) 进 行I/O 调度，尽量把数据和计算分配在同一个节点上，减少了网络开销。此外， Impala 使 用Parquet 列存储格式，提高分析负载的执行效率。

虽然 Impala 的查询处理性能比 Hive(Hive on MapReduce) 要高很多，但是 Impala 并不能取代已有的MapReduce 数据处理系统，而是作为MapReduce 数据处理系统的一个 补充。Impala 适用于处理数据规模适中的 SQL 分析查询，对于大量数据的批量处理任务，



MapReduce 以 及 Hive on MapReduce 仍然是可信赖的选择。

**2.Presto**

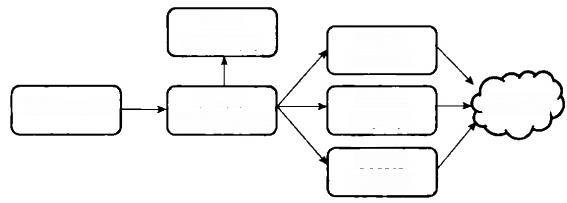
**Presto 是由** **Facebook 公司贡献出来的开源SQL on Hadoop项目(2012年开发，2013** 年**开源)。它的目标和** **Impala** **是类似的，即在** **Hadoop** **平台上提供对结构化数据的交互式** **(Interactive) 查询能力。Teradata 公司除了维护其** **Teradata 数据库的市场领导地位之外，** 也在 Hadoop 平台以及SQL on Hadoop工具等方面大量投入。自从该公司收购了Hadapt 以后，它安排其中的部分工程师跟进Presto项目，并对该软件做出贡献。

由于提供了交互式查询能力， Presto 在查询处理层不提供容错保证。当使用Presto 执 行查询失败时，用户重新提交该查询即可。如果数据存放在HDFS 中，我们可以在数据上 同时运行Presto 和 Hive 。Presto 主要用于执行交互式查询(响应时间在秒级，即小于10 秒),这类查询扫描的数据量一般较少，换句话说，这类查询只是扫描大数据集的一小部 分 . Hive 则主要用于执行长查询 (Long Query),这类查询扫描的数据量较大，比如 PB 级别，执行时间较长。

对于交互式查询， Presto 选择不予提供查询容错保证，降低软件的复杂度，提高执行 效率。Hive on MapReduce 在执行层面，通过中间结果存盘，提供了容错保证，保证长查 询可以最终执行完毕，无论查询执行过程中是否发生各种失败状况(比如节点失败)。

Presto 的主要特点包括：(1)数据通过流水线方式，从一个操作符传送到后续操作 符，在没有必要的情况下，中间结果不用写磁盘。(2) Presto 的查询处理模式是向量化 (Vectorized) 的执行模式。(3) Presto 的执行计划(以及操作符),被动态编译成字节码 (Byte Code, 一种通用的机器指令),然后执行。(4) Presto 虽然用Java 语言编写，但是 它不使用Java 的内存管理接口，而是直接进行内存管理 (Direct Memory Management), 因此它的内存管理是高效的。

Presto的系统架构如图3- 19所示。 Presto 采 用Master-Slave 架构，它由一个Coor- dinator 节点、多个Worker 节点组成。Coordinator 节点中通常内嵌一个Discovery Server。 Coordinator 负责解析 SQL 语句，生成执行计划，分发执行任务给 Worker 节点执行。 Worker 节点负责实际执行查询任务。 Worker 节点启动后，向Discovery Server服务注册， Coordinator 从 Discovery Server 获得可以正常工作的Worker 节点列表。



**Hive**

**Metastore**

**Presto**

Coordinator

**Presto Worker**

**Presto Worker**

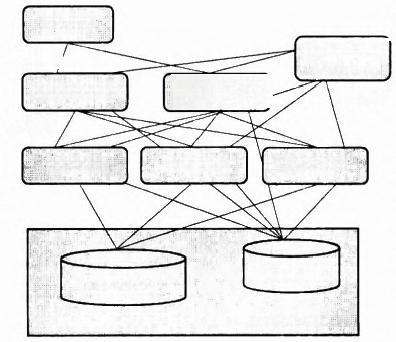
**Presto Worker**

Presto CLI

HDFS

**图3-** **19** **Presto系统架构**

Presto通过插件方式 (Connector, 插件，如图3 - 20 (a) 所示),支持不同的存储引 擎。比如，Presto 通 过Hive Connector 可以存取 Hive 的数据。 Presto 通 过 Hive Connec- tor 的元信息接口 (Meta data API), 存取Hive MetaStore的元信息。调度器 (Scheduler) 通过Hive Connector的数据位置接口 (Data Location API), 了解数据的位置信息。每个 Worker 节点通过 Hive Connector 的数据流接口 (Data Steam API), 与 HDFS 交互读取 数据。如图3- 20 (b) 所示。



Client

<HTTP/>

Coordinator

<HTTP>

**Worker** Worker Worker

Storage Meta data

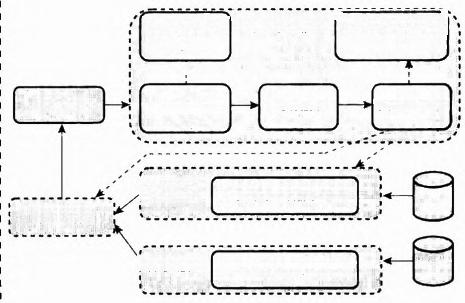
Connector(Pluggable Storage Engine)

Discovery

Server

Coordinator

<HTTP>



Meta Data Coordinator API

不

Client Parser/ Planner Scheduler

Analyzer

Data Stream API

**Worker**

Data Stream API

Data Location API

Worker

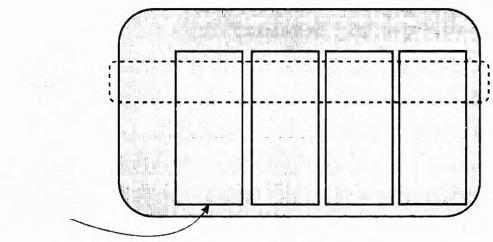
Worker

(a) 系统架构里的存储引擎 (b) 查询处理流程中的存储引擎

**图3-20** **Presto通过插件方式支持不同的后端存储引擎** **(Backend Storage Engine**)

Presto 处理的最小数据单元是一个Page 对 象 ，Page 对象的内部结构如图3- 21所示。

一 个Page 对象包含多个Block 对象，每个Block 对象是一个字节数组，存储一个字段的若 干 Value (即对应若干行)。从逻辑上看，多个 Block 的相应的偏移量上的值构成表格的一 行数据。



**page**

Row

block block

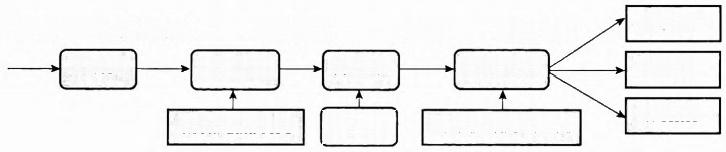
Slice

Byte Array

block block

**图3-21** **Presto的数据模型**

Presto 服务器对其接收到的SQL 查询，大致经过如下几个步骤进行解析和执行(见 图3 - 22)。首先Parser 对 SQL 语句进行词法和语法解析，生成statement 结构。然后An- alyzer 利用元信息对语法树进行检查，生成 analysis 结构。 Planner 利 用Optimizer 的 帮 助 进行优化，生成逻辑计划 plan 。Scheduler 根据数据的位置信息，把逻辑计划划分成物理 计划片段sub plan, 分发给若干 Worker 节点，由其执行。



SQL statement analysis plan sub plan

Parser Analyzer Planner Scheduler

**Worker**

Meta data API Opimizer

**Worker**

**Worker**

Data Location API

**图3-22** **SQL 语句的解析和执行**

下面通过一个SQL 查询实例，解释如何由逻辑计划转换成物理计划，以及物理计划 片段如何在Worker 节点上调度。比如查询：

**Select** c1.rank,count(\*)

**From** city cl join city c2

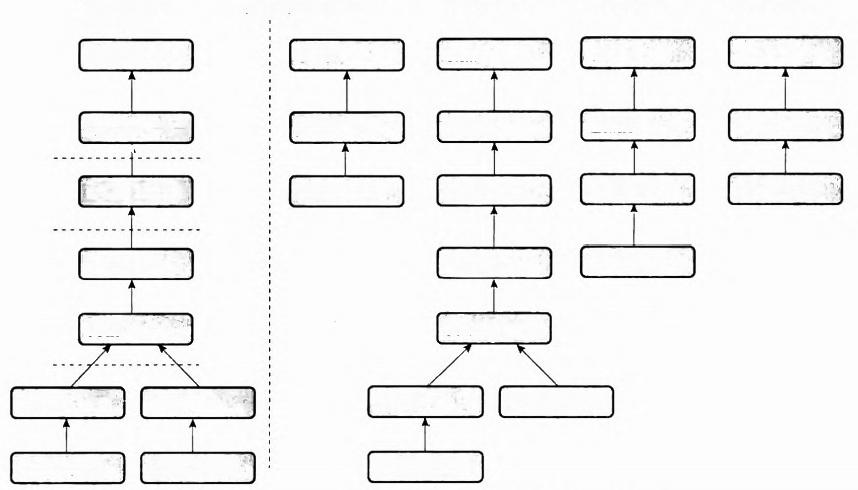
**On** c1.id=c2.id

**Where** c1.id>10

**Group By** c1.rank

**Limit 10**

由 该SQL 查询语句生成的逻辑执行计划 (Logical Execution Plan) 如 图 3 - 2 3 (a) 所示。逻辑执行计划图中的虚线是Presto 对逻辑执行计划进行切分的切分点。这些切分点 把逻辑计划划分成四个子计划 (sub plan), 每 个sub plan 交给 一 个或者多个 Worker 节 点 运行。先导和后续sub plan 之间需要进行数据交换，如图3-23 (b) 所示。



query plan (Final)sub plan 1 sub plan 0 sub plan 2 (partial)

Output sink sink sink

Limit filter Limit Limit

A

Aggregate Table scan Aggregate Aggregate

Project Project exchange

Join Join

**Filter** Filter Filter Exchange

Table Scan Table Scan Table Scan

sub plan 3

Output

Exchange

limit

(a) 查询执行计划 (b) 查询执行计划的子计划

**图3-23** **一个SQL的逻辑执行计划**

该 SQL 语句的各个 sub plan的实际执行过程，请参考图3- 24,具体说明如下：

(1)Coordinator 通 过<HTTP> 协议，调用Worker 节点的任务 (task) 接口，将执行计 划分配给 Worker 节点(图3 - 24中虚线箭头)。

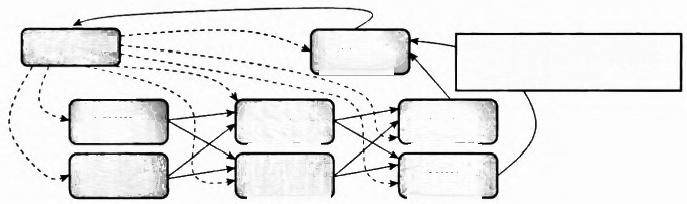
( 2 ) 执 行sub plan 1的 每 个Worker 节点，读取一个Split 的数据并进行过滤后，将数 据分发给每个 sub plan 0 节点，进行连接 (Join) 操作和局部聚集 (Partial Aggregation) 操作。

( 3 ) 执 行sub plan 0 的 每 个 Worker 节点，计算完成后，按Group By Key 的 Hash 值，将数据分发到不同的sub plan 2 节 点 (Group By 是分组操作， Group By Key 是 分 组 操作的Key)。

( 4 ) 执 行sub plan 2 的 每 个 Worker 节点，进行全局聚集，计算完成后，将数据分发 到执行sub plan 3的 Worker 节点，执行limit 操作。



(5)执行sub plan 3的 Worker 节点计算完成后，通知Coordinator 结束查询，并将数 据发送给Coordinator, 由其返回客户端程序。



虚线表示查询计划分配 实线表示数据流

**Worker**

**sub plan 2**

**Worker**

**sub plan 2**

**Worker**

**sub plan 0**

**Worker**

sub plan 0J

**Worker**

sub plan 1

Worker

sub plan 1

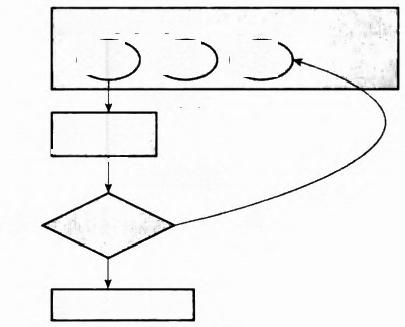
**Worker**

**sub plan 3**

Coordinator

**图** **3** **-** **2** **4** **Sub Plan的分配、依赖关系和执行过程**

Presto 支持节点内部的流水线计算和节点间的流水线计算。图3-25 (a) 表示多个任 务的执行流程，图3-25 (b) 表示每个任务的执行流程。

Pending split pool

Split Split

Split

Take one

Execute for

1 second

Time is up

Done

Y

N

**Remove the split**

Op1

0p2

Op3

Page=op1.getOutput () Op2.addlnput(page)

Page =op2.getOutput()

Op3.addlnput(page)

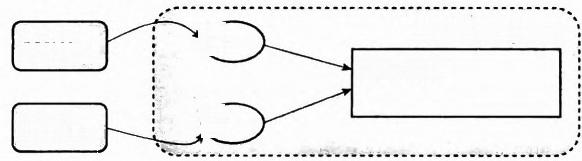
(a) 任务间的调度流程 (b) 任务内的各个Operator的执行流程

**图3** **-** **25** **Worker** **节点内部的计算流程**

在一个Worker节点内部，Worker 节点将最细粒度的任务封装成一个 Prioritized Split Runner 对象，放入pending splitpool优先级队列中。Worker 节点启动一定数目的线程进行计 算。每个空闲的线程，从队列中取出 一 个Prioritized Split Runner对象，然后执行。如果超过 最大执行时间即1秒钟，判断任务是否已经完成，如果已经完成，就从split 队列中删除该 split, 否 则 放 回pending splitpool 队列，以便后续继续调度，执行未完成的工作。

每个任务的执行流程是，依次遍历各个操作符 (Operator), 尝试从上一个操作符取 得一个 Page 对象，如果该Page 对象不为空，那么交给下一个操作符执行。

节点间的流水线计算，通过Exchange 操作符(Operator) 实现(见图3 - 26)。 Ex- change Operator为每一个Split 启动一个<HttpPageBufferClient> 对象，主动向上一个阶段



Exchange Operator

Pages队 列

**Source**

Worker B

C

C

Source

Worker A

**图3** **-** **26** **节点间的流水线计算(C:**[**HttpPageBufferClient)**](HttpPageBufferClient))



(Stage, 即 sub plan) 的 Worker 节点的sink 操作符拉数据。数据的最小单位是一个Page 对象，取到数据后放入Pages 队列中。这个Pages 队列，由本地的下一个阶段 (Stage, 即 sub plan) 的 操 作 符 进 行 消 费 。

**3.3.3** **性能比较**

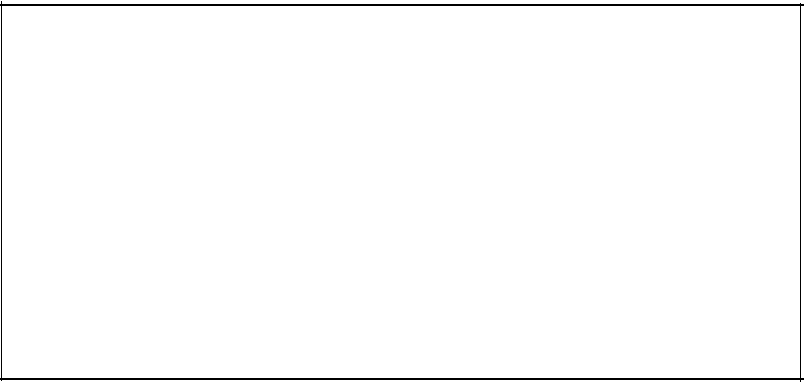
我们从参考文献中选取了部分第三方性能评测结果呈现出来，让读者可以了解到主流 的结构化数据分析系统的相对性能。需要注意的是，这些评测结果是在一定的软硬件条件 和数据规模下，在某个时间获得的结果。考虑到各个开源社区对开源软件以及各个厂家对 其产品的不断改进，这些系统的性能也在不断提高，其相对性能也会发生变化。

**1.AMP Lab Benchmark**

加州大学伯克利分校的 AMP Lab 于2014年初对 Redshift,Hive on MapReduce (v0.12),Shark(v0.8.1),Impala(v1.2.3),Hive on Tez(v0.2.0) 等结构化数据分析 系统进行了性能评测。其中，Redshift 是一个来自Amazon的基于ParAccel 数据仓库技术 的 MPP 数据库。Shark 为SparkSQL 的前身，它是Spark 平台上与Hive 兼容的SQL 查询 引 擎 。

所使用的硬件平台是在Amazon 云平台 (Amazon Elastic Compute Cloud,EC2) 上创 建的虚拟集群。测试数据集包括三张数据库表。Ranking 表保存各个页面和它们的 Page Rank,UserVisits 表保存对每个Web 页面的用户存取日志，Documents 表保存每个页面 的非结构化的HTML 页面内容。

所使用扫描查询 (Scan Query)、聚集查询 (Aggregation Query)、连接查询 (Join Query) 等不同查询负载，对目标系统进行了性能测试。其中Join Query查询对应的SQL 语 句 如 下 ：



**Select** sourceIP,totalRevenue,avgPageRank **From**

(

**Select** sourceIP, **Avg** (pageRank)asavgPageRank, **Sum** (adRevenue)astotal-

Revenue

**From** RankingsASR,UserVisitsASUV

**Where** R.pageURL=UV.destURL

**And** UV.visitDate **Between** Date('1980-01-01') **And** Date('X')

**GroupBy** UV.sourceIP

)

**OrderBy** totalRevenue

**DescLimit 1**

注：上述查询包含一个子查询。子查询对Ranking表和UserVisits 表进行连接，然后以 sourceIP作为分组条件，统计 1980年1月1日以来到某个时间为止(用户提供该参数)的每个sourceIP的平均的PageRank 和广告营收(adRevenue)的 总和。外围查询在上述查询结果基础上，把广告营收最高的sourceIP 选择出来。

Join 查询在各个系统(配置)上的实验结果如表3-5和图3-27所示。当选择率比较 低时，Spark(In-Memory, 表示数据驻留在内存中)获得和Redshift 及Impala(In-Mem-

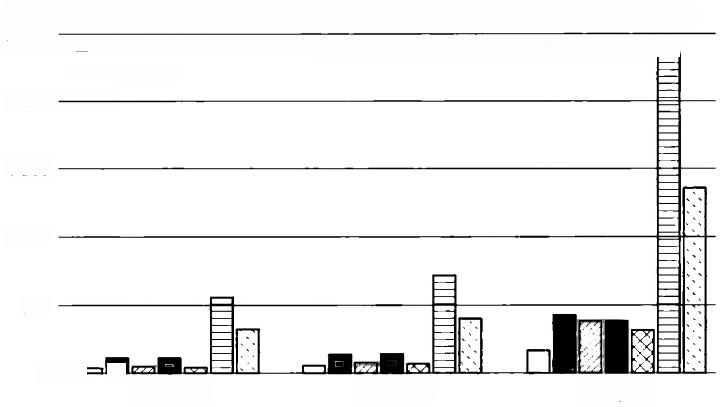
ory) 类似的响应时间，并且比其他系统(配置)的性能都要好。当选择率比较高时， Spark(In-Memory) 比其他系统(配置)的性能都要好，除了Redshift。Redshift总是获 得比Spark 更高的性能。

从这个结果我们可以看出，Redshift 的性能是最好的，Impala 和 Shark 性能接近，两 者的性能都比 Hive 高出很多。Hive(on MapReduce) 的性能最差，Hive on Tez 的性能 相对于Hive 有所改进。

**表3-5** **各个系统(配置** **·)执行Query** **3(不同选择率)的响应时间的中位数**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| System(Setting)  (seconds) | Version | Query 3A | Query 3B | Query 3C |
| 485312 rows  (low selectivity) | 53332015 rows  (median selectivity) | 533287121 rows  (high selectivity) |
| Redshift(HDD) | Current | 33.29 | 46.08 | 168.25 |
| Impala-Disk | 1.2.3 | 108.68 | 129.815 | 431.26 |
| Impala-Mem | 1.2.3 | 41.21 | 76.005 | 386.6 |
| Shark-Disk | 0.8.1 | 111.7 | 135.6 | 382.6 |
| Shark-Mem | 0.8.1 | 44.7 | 67.3 | 318 |
| Hive-YARN | 0.12 | 561.14 | 717.56 | 2374.17 |
| Tez | 0.2.0 | 323.06 | 402.33 | 1361.9 |

\*这里的配置，指的是数据驻留在内存中还是磁盘中。



-Disk-1.2.30Impala-Mem-1.2.3

-Mem-0.8.1 □Hive-0.12YARN

2000 -

1500

1000

500

0-

Query 3A Query 3B Query 3C

2500□Shdski H-D)8-Curent

□Tez-0.2.0

■Impala

囚 Shark

**图3** **-** **27** **各个系统(配置)执行Query 3 (不同选择率)的响应时间的中位数**

需要指出的是，这几个软件，包括 Impala 和 Hive 以及Shark (现在是Spark SQL), 都在不断改进之中，当它们使用了更多的优化技术以后，其性能的相对关系将发生变化。 关于其他查询的实验结果以及最新的实验结果，读者可以参考 AMP Lab网站。①

**2.Actian Benchmark**

Actian 公司于2016年使用TPC-H 数据集 (Scale Factor 为1000),对 VectorH (5.0)进行了性能评测，并且和 Impala(2.3),Hive(1.2.1),HAWQ(1.3.1) 以及



①<https://amplab.cs.berkeley.edu/benchmark/.>



SparkSQL(1.5.2) 进行了对比。 Actian 公司使用的集群由10个节点组成，运行 Hadoop 2.6.0,其中一个节点作为Name Node,其余9个节点用于进行SQL on Hadoop系统的性 能试验。每个节点有2个Intel Xeon E5-2690v2 CPU(3GHz),256GB内存，整个集群总 共有20个CPU 核心(40个超线程)。节点间通过10Gbit Ethernet 网进行连接。每个节点 有24个600GB 的磁盘，其中一个安装操作系统，其余用于存储数据(使用HDFS 分布式 文件系统)。

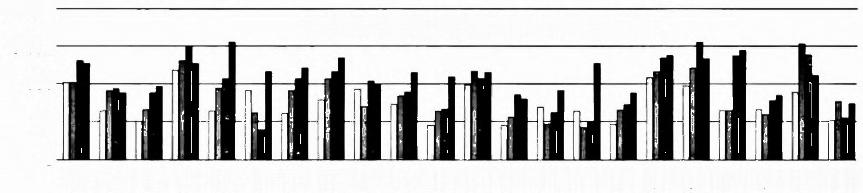
Actian 公司使用了Hive 所有可用的优化技术，包括 Tez 执行引擎 (Tez Execution Engine) 、 向量化执行模式 (Vectorization) 、 谓词下推 (Predicate Pushdown)、基于代价 的优化器 (Cost-Based Optimizer)。数据采用ORC 格式( 一 种列存储格式)进行存储， 采用snappy 压缩技术进行压缩。

HAWQ 是来自EMC Green plum公司的SQL on Hadoop系统。给 HAWQ 的每个节点设 定了21GB的 vmem 参数，其中12GB 作 为work memory,6GB用于暂存查询语句 (Query Statement) 。 每个节点开辟256MB的共享缓存 (Shared Buffer) 。Orders表和Lineltem 表采用 Parquet 格式(另外一种列存储格式)进行存储，采用snappy 压缩技术进行压缩。

在 SparkSQL 配置中，采用了snappy 压缩方式的Parquet 格式进行数据存储。试验结果 如图3 - 28和表3 - 6所示。 VectorH 系统获得了秒级的响应时间，比 HAWQ,SparkSQL,

Impala,Hive 等系统快得多，超过一个数量级(甚至达到2～3个数量级)。

How Many Times are VectorH Faster than These Systems?



10000-

□HAWQ ■SparkSQL ■Impala ■Hive

1000-

100-

10-

1

Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10Q11Q12Q13Q14Q15Q16Q17Q18Q19Q20Q21 Q22

**图3-28** **VectorH比各个系统快多少倍**

**表3-6** **TPC-H(Scale** **Factor** **1000)数据集上各个查询的响应时间(秒)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | VectorH | HAWQ | SparkSQL | Impala | Hive |
| Q1 | 1.5 | 158.2 | 155.4 | 585.4 | 490.1 |
| Q2 | 1.14 | 21.46 | 74.98 | 81.81 | 63.57 |
| Q3 | 3.16 | 32.06 | 62.38 | 167.7 | 266.6 |
| Q4 | 0.17 | 38.21 | 68.27 | 163.18 | 59.08 |
| Q5 | 1.94 | 36.38 | 146.5 | 242.5 | DNF |
| Q6 | 0.31 | 20.19 | 5.1 | 1.81 | 63.63 |
| Q7 | 2.75 | 44.74 | 180.2 | 369.0 | 721.8 |
| Q8 | 1.31 | 48.38 | 174.6 | 276.2 | 625.6 |
| Q9 | 11.11 | 766.4 | 264.0 | 1242.9 | 1077 |
| Q10 | 1.21 | 32.97 | 56.62 | 69.97 | 230.5 |
| Q11 | 1.69 | 12.48 | 30.28 | 35.04 | 246.1 |
| Q12 | 0.34 | 31.75 | 66.97 | 45.67 | 65.78 |
| Q13 | 3.66 | 27.97 | 47.65 | 180.8 | 140.7 |
| Q14 | 0.83 | 19.47 | 6.92 | 13.95 | 53.23 |
| Q15 | 1.63 | 31.58 | 11.16 | 15.19 | 556.5 |
| Q16 | 1.68 | 14.17 | 33.81 | 47.52 | 92.51 |



续前表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | VectorH | **HAWQ** | **SparkSQL** | Impala | Hive |
| Q17 | 1.24 | 173.2 | 244.9 | 581.53 | 711.7 |
| Q18 | 0.99 | 87.08 | 254.7 | 1234 | 454.5 |
| Q19 | 1.32 | 24.82 | 24.89 | 714.7 | 1010 |
| Q20 | 2.15 | 42.84 | 31.56 | 74.25 | 100.5 |
| Q21 | 1.48 | 84.7 | 1614 | 880.8 | 247.7 |
| Q22 | 2.84 | 29.44 | 91.18 | 34.81 | 81.11 |

**3.AtScale Benchmark**

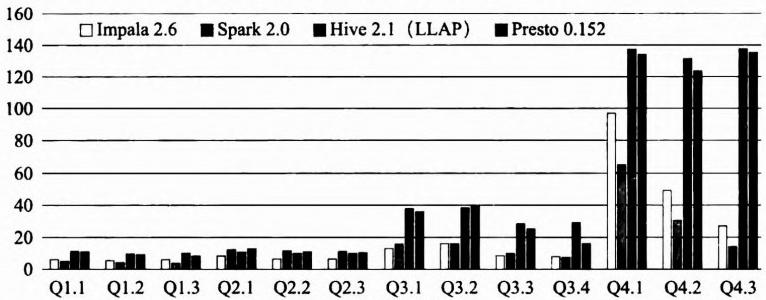
2016年底， AtScale 公司发布了针对主流的SQL on Hadoop结构化大数据分析平台的 性能评测结果。所评测的系统包括 SparkSQL(2.0.1),Impala(2.6),Hive/Tez(2.1.

Tez 版本为0.8.5)和Presto(0.152)。

AtScale 公司进行测试的集群包括12个节点，其中1个节点为Master Node,1个节 点作为Gateway Node, 其余10个节点作为 Data Node。每个节点有128GB 内存，Data Node 有32个CPU 核心以及2个512MB 的 SSD (固态硬盘)。SparkSQL 和 Impala 使用 Parquet 列存储格式，Hive 和 Presto 使用ORC 列存储格式。

AtScale 公司采用SSB(Star Schema Benchmark) 评测基准，对这些系统进行测试。 在其中的一个测试中，LineOrders 的记录数达到60亿条。实验结果如图3-29和表3-7 所示。

Large Query Performance,Response Time(Sec)



Query ID

**图3-29** **采用SSB测试基准对四个系统进行评测的结果**

**表3-7** **采用SSB测试基准对四个系统进行评测的结果**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Query | Query Execution Time(秒) | | | |
| Impala 2.6 | Spark 2.0 | Hive 2.1(LLAP) | Presto 0.152 |
| Q1.1 | 5.6 | 4.8 | 10.5 | 10.3 |
| Q1.2 | 5.0 | 3.8 | 9.1 | 8.7 |
| Q1.3 | 5.6 | 3.3 | 9.3 | 7.9 |
| Q2.1 | 8.0 | 11.8 | 10.0 | 12.4 |
| Q2.2 | 6.2 | 11.0 | 9.3 | 10.4 |
| Q2.3 | 6.0 | 10.6 | 9.3 | 9.8 |
| Q3.1 | 12.6 | 15.5 | 37.5 | 35.5 |

续前表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Query | **Query** **Execution** **Time(秒)** | | | |
| Impala 2.6 | Spark 2.0 | **Hive** **2.1(LLAP)** | Presto 0.152 |
| Q3.2 | 15.5 | 15.6 | 38.1 | 39.9 |
| Q3.3 | 8.0 | 9.2 | 28.1 | 24.9 |
| Q3.4 | 7.4 | 6.8 | 28.7 | 15.7 |
| Q4.1 | 97.3 | 64.9 | 137.6 | 134.4 |
| Q4.2 | 49.1 | 30.3 | 131.9 | 123.8 |
| Q4.3 | 26.8 | 13.2 | 137.9 | 135.5 |

新版本的 Impala,SparkSQL 和 Hive 都相对于其各自早先版本有了较大的性能提升。 基于该评测结果，得出的结论是：

(1)没有一个SQL on Hadoop 在所有的查询上都获得最好的性能。SparkSQL 和 Im- pala比 Hive 要快得多。在多数查询上，Impala 和 SparkSQL 的性能差异很小。Presto 获 得了和 Hive 相似的性能。

(2)随着一个查询里Join操作数量的增加，查询处理时间也相应增加。当Join操作 的数量从1增加到3时， Hive 和 Presto的查询响应时间的变化较大。Impala 和SparkSQL 则比Presto 和 Hive 获得更好的性能。

(3)随着查询选择率的提高，查询响应时间也相应增加。 Hive 和 Presto 对查询选择 率的敏感度相对较弱，也就是随着选择率提高，响应时间的变化没有那么大。

(4)在所有的系统上，在两张大表之间进行Join 操作，查询的响应时间都会变得很 慢。一般来讲，对两张大表进行Join 操作，比如Customer 表达到10亿条记录，是一个代 价极大的操作。

**3** **.** **4** **思** **考** **题**

(1)数据仓库、星型模型。

(2)联机分析处理的主要操作。

(3)联机分析处理的三种实现技术。

(4)列存储技术。

(5)位图索引技术。

(6)内存数据库技术。

(7)并行数据库架构、数据分片、查询处理。

(8)Teradata 数据库及其特点。

(9)SAP HANA数据库及其特点。

(10)MonetDB/Vectorwise/VectorH 系统及其特点。

(11)Impala 系统及其特点。

(12)Presto 系统及其特点。